Extended pointers for memory protection in single address space 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 single address space paradigm of memory reference, we identify a set of mechanisms aimed at preserving the integrity of the virtual space of a given process from erroneous or fraudulent access attempts originated from other processes. These mechanisms guarantee a level of protection that is, in many respect, superior to that of a traditional multiple address space environment. We introduce extended pointers as a generalization of the concept of a memory address, which includes a password and an access authorization. A universally known, parametric one-way function is used to assign passwords. An *ad hoc* circuitry for address translation supports memory reference and protection at the hardware level. A small set of protection primitives forms the process interface of the protection system. The resulting protection environment is evaluated from a number of viewpoints, which include extended pointer forging, and the review and revocation of access authorizations. An indication of the flexibility of the proposed protection paradigm is given by applying extended pointers to the solution of a variety of protection problems.

Keywords: Access authorization, memory addressing, parametric one-way function, password, protection.

1 INTRODUCTION

In a traditional memory management approach, a separate virtual space is assigned to each process. The address space of a given process contains the information items relevant to the execution of that process. Separation enforces protection; each process has no virtual address that maps into a physical address belonging to another process. In a system of this type, if an information item is placed at different virtual addresses for different processes, complex synonym problems arise, in the circuitry for virtual to physical address translation, and in the translation lookaside buffer, for instance [1], [2], [3]. Code and data sharing is a relevant issue. A solution is to place the shared item at the same address in the virtual spaces of all the processes involved in the sharing activities, in the present as well as in the future [4]. This solution is prone to place severe constraints on address space management.

In a different approach, all the processes share a *single address space* [5], [6], [7], [8]. In this approach, two or more processes aimed at accessing the same given information item simply use the virtual address of this item. The virtual address is unique, and independent of both the process, and the item position in the physical memory. Information sharing is facilitated, but mechanisms must be provided to preserve the integrity of the virtual space areas reserved for a given process from erroneous or malevolent access attempts originated from the other processes. These mechanisms should guarantee a level of protection comparable to that of a traditional multiple virtual space environment. An essential requirement is that each process should have complete control over any form of access that involves its own memory areas. This means that the distribution and revocation of access privileges should be facilitated.

With reference to single address space systems and forms of segmentation with paging, this paper presents a generalization of the concept of a pointer (memory address), called *extended pointer*, which includes the specification of a memory segment, an access authorization, and a password. If the password is valid, the extended pointer grants the specified access authorization for the named segment. The access authorization can be expressed in terms of any combination of the *read*, *write*, and *execute* access rights. We shall show that a single password, the *master password*, is sufficient to define extended pointers for an arbitrary number of segments. Extended pointers do not need to be segregated into special memory regions. Instead, they can be mixed in memory with ordinary information items. A subject that holds a given extended pointer can *reduce* this pointer to reference a fraction of the original memory area, and can *weaken* the pointer to include less access rights. The review and revocation of access authorizations is supported. Protection from forgery is guaranteed by master passwords.

The rest of this paper is organized as follows. Section 2 introduces a classical protection model based on subjects, objects and an access matrix. Preeminent implementations of this model are analyzed. The accent is on the inherent problems and limitations of each solution. Section 3 introduces our protection paradigm based on extended pointers. Extended pointer validation and memory addressing are analyzed in special depth. An *ad hoc* circuitry for address translation is presented, supporting a mechanism of memory reference and protection based on extended pointers. A set of primitives is introduced, the *protection primitives*, which form the process interface of the protection system. Section 4 gives an indication of the flexibility of the proposed protection paradigm. Extended pointers 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 extended pointer forging, the revocation of access authorizations, and the relation of our work to previous work. Section 6 gives concluding remarks. Appendix A details the actions involved in the execution of each protection primitive.

2 THE PROTECTION MODEL

In a classical protection model, active entities $S_0, S_1, \ldots$, called *subjects*, generate access attempts to passive entities $B_0, B_1, \ldots$, called *objects* [9], [10], [11]. Objects are typed. The definition of the type of each object states the set of operations that can be applied to this object, and a set of access rights. Each operation is associated with one or more access rights. Execution of a given operation terminates successfully only if the subject issuing this operation holds the corresponding access rights.

In this model, the protection system can be represented in the form of a matrix, called the *access matrix* AM, featuring a row for each subject and a column for each object [12], [13], [14]. The contents of element $AM_{i,j}$ in row i and column j specifies the *access authorization* held by subject S_i on object B_j. An access authorization is a collection of access rights. A central issue in the implementation of a protection system is how to represent the access matrix in memory. Capability-based addressing is a multi-decade old solution to this problem.

2.1 Capabilities

A *capability* is a protected pointer having the form (B_j, z), where B_j is the identifier of a protected object, and z specifies an access authorization for this object [15], [16], [17]. Typically, the z field is encoded as a sequence of bits, one bit for each access right that may be included in an access authorization. If the bit in a given position of z is asserted, then the capability grants the access right for object B_j that corresponds to that position.

The set of capabilities held by a given subject specifies the access authorizations held by that subject on the protected objects. A subject S_i aimed at executing a given operation on object B_j is required to exhibit a capability (B_j, z), and the z field of this capability must specify the access authorization which is necessary for successful execution of that operation. In the access matrix model, the capability-based approach corresponds to a representation of the access matrix that is *by rows.* If S_i holds capability (B_j, z), then z specifies the contents of element $M_{i,j}$ of the access matrix.

In a classical implementation of a capability-based addressing system, the objects are memory segments, i.e. sets of adjacent memory cells. The processor hardware is augmented by an array of special registers, the *capability registers*, where capabilities are stored for memory reference [18], [19]. A subject aimed at accessing a given segment must have previously loaded a capability for this segment into a capability register. For each memory address, the instruction formats incorporate the index of a capability register in the capability register array. The access terminates successfully only if the capability in the named register includes the necessary access authorization, e.g. access right *write* if a data segment should be accessed to modify its contents.

2.1.1 Segregation

A basic problem in capability systems is capability *segregation.* We must prevent an erroneous or fraudulent process from altering a capability, for instance, by adding new access rights, or even modifying the object identifier to forge a new capability for a different object. In a segmented memory system, a viable solution is to reserve special segments for capability storage, the *capability segments* [15] (in contrast, the *data segments* will be reserved to store ordinary information items). In this approach, the instruction set of the processor includes a few special instructions, the *capability instructions*, aimed at manipulating capabilities in a strictly controlled fashion. Only the capability instructions can be used to access a capability segment; if an ordinary instruction is used, a protection exception is raised, and execution fails. This approach is prone to segment proliferation. Processes are forced to adhere to a complex model for data structuring, which usually takes the form of a tree. The root and the nodes at the intermediate levels of the tree are reserved for capability storage, whereas ordinary information items are contained in the leaf nodes. Consider a simple data item supported by two data segments, for instance. A capability segment is necessary to contain the capabilities for these data segments. This is an undesirable complication of the whole memory management process.

In an alternative approach, a one-bit *tag* is associated with each memory cell. If asserted, the tag of a given cell specifies that this cell contains a capability [19], [20], [21]. If an ordinary instruction is issued on a cell whose tag is asserted, a protection exception is raised, and execution of the instruction fails; alternatively, the tag is cleared to invalidate the capability [22]. The tag based approach must be supported by *ad hoc* memory systems, e.g. the cells in the memory banks of a 64-bit system will be 65 bits wide. This is in contrast with the requisite of hardware standardization. Complications ensue in the caches, which have to store the tags, and in memory management, owing to the need to save and then restore the tags as part of the usual page swapping activities between the primary memory and the secondary memory.

2.1.2 Review and revocation

The main advantage of capability systems is simplicity in object sharing. A subject that holds a capability referencing a given object is free to transfer a copy of this capability to another subject. In this way, the recipient acquires the access authorization specified by that capability. The recipient can also transmit the capability further. As a result, it is hard to keep track of the memory location of every copy of a given capability. The original capability holder should be given the ability to review the distribution and revoke the capability copies from the recipients.

2.2 Passwords

2.2.1 Segregation

In a different, password-based protection model, a collection of *passwords* is associated with each given object, a password for each access authorization defined for that object. A subject that holds one of these passwords is allowed to access the object to carry out the operations permitted by the corresponding access authorization. If passwords are large, sparse and chosen at random, the processing time cost for a fraudulent subject to guess a valid password by a brute force attack can be prohibitive. It follows that passwords can be mixed in memory with ordinary information items; this is an effective solution to the segregation problem.

2.2.2 Proliferation

A drawback of password-based approaches is password proliferation. The internal representation of a given object should contain the passwords corresponding to all the significant access authorizations for this object. High memory costs are connected with the necessity to store several passwords for each object. The resulting complexity in access right management can be inappropriate, especially if protection should be exercised at a high granularity level, for small-sized objects and many different access authorizations. Ease of access right management is especially important if we are aimed at supporting the *principle of least privilege*, i.e. each subject should be granted least possible privileges, and a privilege should be granted to least possible subjects [23], [24].

For instance, for memory segments and the usual access rights, *read*, *write*, and *execute*, a complete coverage of all possible access authorizations would require a total of seven passwords for each segment, one password for each combination of the three access rights. Alternatively, we can take advantage of three passwords, one password for each access right. In this case, a subject that holds full access rights for a given segment should possess the three passwords of this segment. Successful execution of an action requiring both to read and to modify the segment contents implies that two passwords are presented and validated.

2.2.3 Weakening and reduction

Further issues are password weakening and reduction. A subject that holds a given password is free to transfer this password to another subject. Consequently, the recipient acquires the whole access authorization connected with the password. It is impossible for the original subject to *weaken* the access authorization expressed by the password. Consider a password for a given memory segment, for instance, and suppose that this password corresponds to the three access rights, *read*, *write* and *execute*. The process that holds this password may well be aimed at transmitting an access authorization to read only. Intervention of a password manager is necessary, which is part of the protection system. The process sends the original password to the password manager, and receives the password corresponding to the weakened authorization. If one such password does not exist, the password manager should create this password. If this is indeed impossible, a negative response is sent to the original process. The entire procedure is much more complicated than implied by the required effect. A different solution is desirable, whereby the process is given the ability to weaken the password autonomously.

Furthermore, it is impossible to *reduce* a password to reference an object fraction. Consider a process that holds the password for a given memory area, and is aimed at transforming this password to reference a segment in that area. Once again, a mechanism is desirable whereby the process can carry out the transformation autonomously.

3 THE PROTECTION SYSTEM

This paper presents an overall solution to the problems, outlined above. We refer to single address space systems featuring a virtual space that is partitioned into fixed-size *pages*. A virtual memory *area* is a sequence of adjacent pages. Areas can overlap, partially or totally. This means that a page can be part of two or more areas. Areas cannot be used for effective memory accesses. Instead, *segments* are the units of virtual memory reference. A segment is a sequence of adjacent pages entirely contained within the boundaries of an area. Thus, a segment is a fraction of an area. In our protection model, objects can be areas and segments, and subjects can be processes or, in an event driven environment, activities caused by events, e.g. hardware interrupts [25]. For an area, a single action is defined, to create segments in this area. For a segment, the three actions are to read, to modify and to execute the segment contents.

An extended pointer can be an *area pointer* or a *segment pointer*. An area pointer specifies an area and a password. If the password is valid, the area pointer grants the authorization to create segments in this area. A segment pointer specifies an area, a segment in this area, an access authorization and a password. The access authorization can be expressed in terms of any combination of the *read*, *write* and *execute* access rights. If the password is valid, the segment pointer grants the specified authorization for the named segment.

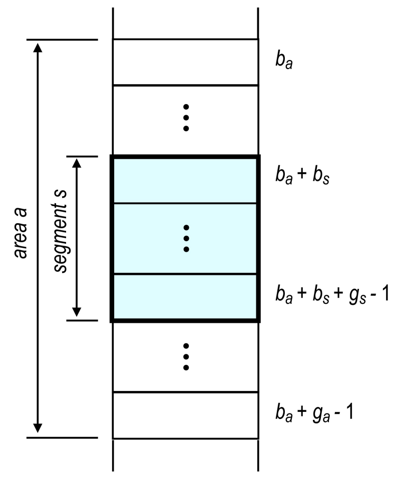

Figure 1: An area a consisting of g_a adjacent pages. The area starts at page b_a, the area base. It includes a segment s consisting of g_s adjacent pages. The segment starts at page $b_a + b_s$, where b_s is the segment base.

3.1 Areas and Segments

An area a is completely defined by an *area base* b_a, equal to the order number of the first page of the area, and an *area length* g_a, equal to the number of pages that form the area (Figure 1). A segment s included in area a is completely defined by the *segment base* b_s, equal to the number of pages between the first page of the area and the first page of the segment, and a *segment length* g_s, equal to the number of pages that form the segment. Thus, the *absolute* numbers of the first and last page of segment s are given by $b_a + b_s$ and $b_a + b_s + g_s - 1$, respectively. The *inclusion condition* is $b_s + g_s \leq g_a$, that is, the entire segment must be contained within the area boundaries.

When a process is generated, one or more master passwords are created, and one or more virtual memory areas are allocated for this process. Several master passwords for the same process permit forms of selective revocation of access authorizations; this issue will be discussed later. For each area, an *area pointer* is created, and is granted to the process. The area pointer includes an *area descriptor*, which identifies the area, and an *area password.* The area pointer is valid only if the area password is valid, that is, it descends from a master password by application of a *parametric one-way function*, the *global function* f, which is unique for the whole system, and is universally known. Function $f_p(x)$ is parametric one-way if, given a parameter p and a value y, it is computationally unfeasible to determine x such that $y = f_p(x)$ [26]. Thus, a parametric one-way function is a family of one-way functions, one for each value of the parameter [27]. The design and implementation efforts can be reduced by taking advantage of a good cryptosystem, e.g., if E_x is a symmetric cipher, we have $f_p(x) = E_x(p)$ [28].

A process that holds a valid pointer for a given area can allocate segments in this area. For each segment, the process receives a segment pointer, which can be used to

Table 1: Extended pointers.

Area pointer $P_a = (M, d_a, descr_a)$
- M: identifier of a master password
- d_a: value of an area password
- $d_a = f_{descr_a}(m)$
- m: value of master password M
- $descr_a = (b_a, g_a)$: area descriptor
- b_a: area base
- g_a: area length

Segment pointer $P_s = (M, d_s, descr_a, descr_s)$
- d_s: value of a segment password
- $d_s = f_{descr_s}(d_a) = f_{descr_s}(f_{descr_a}(m))$
- $descr_s = (b_s, g_s, z_s)$: segment descriptor
- b_s: segment base
- g_s: segment length
- z_s: access authorization

access the segment. The segment pointer is a form of extended pointer that includes an *area descriptor* identifying the area, and a *segment descriptor* identifying the segment. In turn, the segment descriptor includes an *access authorization* specifying the actions that can be successfully accomplished on the segment contents. The access authorization can be expressed in terms of any combination of the three access rights, *read*, *write* and *execute*. The segment pointer also includes a *segment password*. The pointer is valid only if the segment password is valid, that is, it descends from the password of the area of that segment by application of global function f.

3.2 Extended Pointers

The descriptor of area a is denoted by $descr_a$. It has the form (b_a, g_a), where b_a is the area base and g_a is the area length. An area pointer P_a that references a has the form $P_a = (M, d_a, descr_a)$, where M is the *identifier* of a master password, and d_a is the *value* of an area password (Table 1). We have $d_a = f_{descr_a}(m)$, where quantity m is the *value* of the master password whose identifier is M, and the parameter of global function f is the result of the *concatenation* (joining) of quantities b_a and g_a that form $descr_a$.

The descriptor of segment s is denoted by $descr_s$; it has the form (b_s, g_s, z_s), where b_s is the segment base, g_s is the segment length, and z_s is an access authorization. Quantity z_s is encoded in three bits, one bit for each of the three access rights, *read*, *write* and *execute*. A given access right is included in z_s if the corresponding bit is asserted. A segment pointer P_s that references s has the form $P_s = (M, d_s, descr_a, descr_s)$, where M is the identifier of a master password, d_s is the value of a segment password, $descr_a$ is the descriptor of the area a including s, and $descr_s$ is the segment descriptor. We have $d_s = f_{descr_s}(d_a)$, where the parameter of global function f is the result of the concatenation of the three quantities

b_s, g_s, and z_s that form $descr_s$. Thus, the password of a given segment is the result of a double application of function f, to produce d_a and d_s, respectively. In the second application, z_s is part of the parameter, and consequently, different access authorizations for the same segment correspond to different segment passwords.

Let us now consider a subject S that holds pointer $P_a = (M, d_a, descr_a)$ referencing area a, as is specified by area descriptor $descr_a = (b_a, g_a)$. Suppose that S is aimed at generating a pointer for a segment s in a, as is specified by segment descriptor $descr_s = (b_s, g_s, z_s)$. S is in the position to generate the segment pointer by applying global function f to d_a. We have $P_s = (M, d_s, descr_a, descr_s)$, where $d_s = f_{descr_s}(d_a)$. We have obtained this important result by taking advantage of global function f, which is universally known.

3.3 Extended Pointer Validation

The protection system maintains a table, the *master password table*, featuring an entry for each master password. The entry for a given master password contains both the identifier M and the value m of this master password. The table is contained in a memory region reserved for the protection system. Thus, ordinary processes are prevented from accessing the values of the master passwords.

An extended pointer defined in terms of a given master password is valid if the area or segment password it contains can be generated by application of global function f, starting from the value of that master password. More specifically, let us consider area pointer $P_a = (M, d_a, descr_a)$, where $descr_a = (b_a, g_a)$. P_a is valid if area password d_a can be generated starting from the value m of master password M, according to relation $d_a = f_{descr_a}(m)$. Quantity m is only contained in the password table, and consequently, the area pointer can only be validated by the protection system. Similarly, let us consider segment pointer $P_s = (M, d_s, descr_a, descr_s)$, where $descr_s = (b_s, g_s, z_s)$. P_s can be validated by applying global function f twice, i.e. P_s is valid if $d_s = f_{descr_s}(f_{descr_a}(m))$. In this case, too, validation requires quantity m, and consequently, it is a prerogative of the protection system.

3.4 Memory Addressing

The protection model, described so far, is conceived to be integrated at the hardware level with an *ad hoc* circuitry for address translation. This circuitry supports a pointer-based mechanism of memory reference and protection. It includes an array of registers, the *pointer registers* $PR_0, PR_1, \ldots$, which are loaded and possibly cleared under control of user programs. Let us refer to the segment referenced by segment pointer $P_s = (M, d_s, descr_a, descr_s)$, where $descr_a = (b_a, g_a)$ and $descr_s = (b_s, g_s, z_s)$. Each pointer register can contain an *internal descriptor*, derived from a segment pointer. The internal

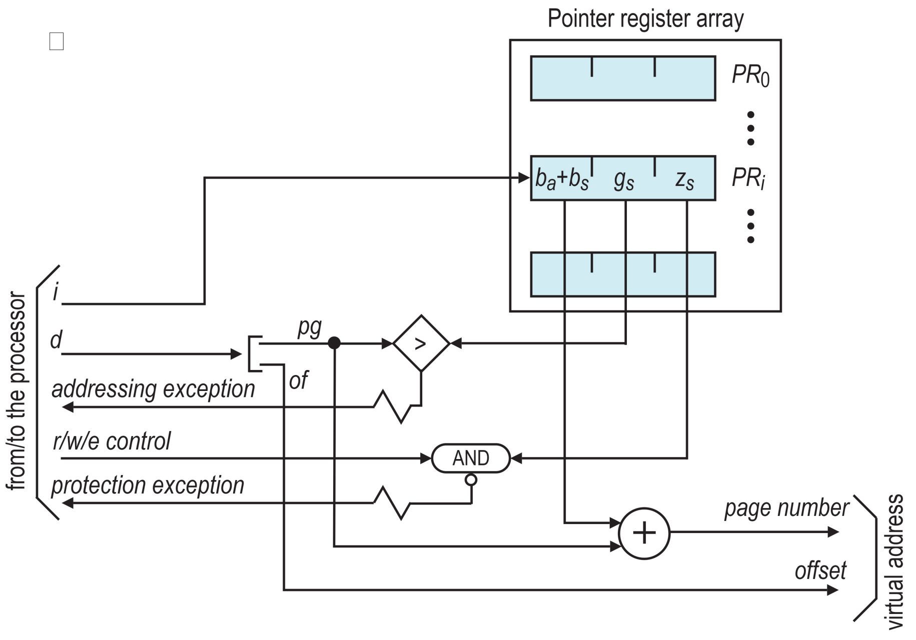

Figure 2: Translation of a memory address generated by the processor into a virtual address consisting of a virtual page number and an offset. The processor address has the form (i, d), where quantity i selects pointer register PR_i in the pointer register array, and quantity d is a displacement in the segment referenced by this pointer register.

descriptor is a triple $(b_a + b_s, g_s, z_s)$, where quantity $b_a + b_s$ is the *absolute* number of the first page of a memory segment (see Figure 1).

A memory address generated by the processor has the form (i, d), where quantity i identifies a pointer register, and quantity d is a displacement. The address translation circuitry converts this *processor address* into a *virtual address* consisting of a page number and an offset. The translation proceeds as follows (Figure 2):

1. Quantity i is used to select pointer register PR_i in the pointer register array. Let $(b_a + b_s, g_s, z_s)$ be the internal descriptor contained in this pointer register.
2. Displacement d is partitioned into a page number *pg* and an offset of. The page number is compared with quantity g_s, contained in PR_i. If *pg* $> g_s$, an addressing exception is generated to the processor. Otherwise,
3. The bitwise AND of the values of the *read/write/execute* control lines from the processor and the corresponding three bits that form the z_s field of PR_i is evaluated. If the result is 0, the access right necessary to accomplish the access is lacking (e.g. the access is to write and the *write* bit in z_s is cleared). In this case, a protection exception is sent to the processor. Otherwise,
4. The base $b_a + b_s$ of the memory segment specified by PR_i is added to page number *pg* and is paired with offset of to obtain the virtual address of the referenced information item. This virtual address is sent to the circuitry for virtual-to-physical address

translation.

3.5 Special Passwords

The protection system defines three *special access rights*, namely *create*, *delete*, and *new.* Access rights *create* and *delete* make it possible to create new master passwords, and to delete the existing master passwords, respectively. Access right *new* makes it possible to allocate new virtual memory areas. When the system is initialized, three passwords are generated, called the *special passwords*, one password for each special access right. We shall denote the *value* of a special password by w, e.g. w_{new} is the value of the special password granting access right *new.*

The special passwords are assigned to selected system components. Examples are the process allocator, the process deallocator, and the area manager, which are assigned w_{create}, w_{delete} and w_{new}, respectively. When a new process is generated, the process allocator uses w_{create} to create one or more master passwords; these master passwords are assigned to the process. When a process asks for a memory area, the area manager determines whether the request can be accepted, and then, it allocates a new area by using w_{new} and the master passwords indicated by the process. A pointer for this area is returned to the process. When a process terminates, the process deallocator uses w_{delete} to delete the master passwords assigned to that process.

A process that holds a pointer for a given area is in the position to allocate segments in that area. Afterwards, these segments can be used for memory reference. No special password is necessary for segment allocation. The process can interact with other processes via shared segments; to this aim, pointers for these segments should be preventively copied to these other processes.

3.6 Protection Primitives

We hypothesize that the processor supports a separation between a system mode and a user mode with memory access limitations. In the system mode, the processor can access the pointer registers to modify their contents, as is necessary to load a pointer register with an internal descriptor resulting from translation of a segment pointer (see Section 3.4). Furthermore, in the system mode the address translation circuitry, illustrated in Figure 2, is disabled. In a processor address, the index i of the pointer register is ignored; the displacement d is partitioned into a page number and an offset to form the virtual address. No page number relocation takes place, and no access right check is carried out.

In contrast to traditional protection models using the system mode for operating system activities, in our model the system mode is restricted to the execution of a set of primitives, the *protection primitives*, which form the process interface of the protection

Table 2: The protection primitives.

$M \leftarrow createMasterPassword(w_{create})$	Creates a new master password, inserts the identifier M and the value m of this master password into a free entry of the master password table, and returns M. Requires special password w_{create}.
$deleteMasterPassword(w_{delete}, M)$	Deletes the master password whose identifier is M from the master password table. Requires special password w_{delete}.
$P_a \leftarrow newArea(w_{new}, M, descr_a)$	Allocates the area specified by $descr_a$. Returns a pointer $P_a = (M, d_a, descr_a)$ referencing the new area, where $d_a = f_{descr_a}(m)$ is the area password. Requires special password w_{new}.
$P_s \leftarrow newSegment(P_a, descr_s)$	Allocates the segment specified by $descr_s$ in the area referenced by area pointer $P_a = (M, d_a, descr_a)$. Returns a pointer $P_s = (M, d_s, descr_a, descr_s)$ referencing this segment, where $d_s = f_{descr_s}(d_a)$ is the segment password corresponding to the access authorization specified by $descr_s$. Requires a valid P_a.
$loadPointerRegister(P_s, i, mask)$	Transforms the segment descriptor $descr_s$, contained in segment pointer P_s, into an internal descriptor, and loads the result into pointer register PR_i. Uses $mask$ to weaken the access authorization z_s in $descr_s$. Requires a valid P_s.
$clearPointerRegister(i)$	Clears pointer register PR_i.

system (Table 2). These primitives can be implemented at software level by system routines. A call to a protection primitive takes the form of a system call that traps into the system mode. This is necessary to access the pointer registers and the memory regions reserved for the protection system, in particular the master password table (see Section 3.3).

In the rest of this section, we shall illustrate the effects of each protection primitive from the point of view of a process that issues a call to this primitive. Appendix A contains a more detailed description of the actions involved in the execution of each primitive.

3.6.1 Master passwords

A first example of a protection primitive is the $M \leftarrow createMasterPassword(w_{create})$ primitive. Its execution generates the identifier M and the value m of a new master password, and returns M. Pair (M, m) is inserted into a free entry of the master password table. Execution requires special password w_{create}. This primitive is used by the process allocator while generating a new process, to create the master passwords for this process (see Section 3.5).

Protection primitive $deleteMasterPassword(w_{delete}, M)$ deletes the master password whose identifier is M. Execution of this primitive eliminates this master password from the master password table. Execution requires special password w_{delete}. This primitive can be used by the process deallocator when a process terminates, to delete the master passwords associated with this process. Furthermore, the ability to delete the master passwords supports the review and revocation of access authorizations. This issue will be discussed in depth in forthcoming Section 5.4.

3.6.2 Extended pointers

Let $descr_a = (b_a, g_a)$ be an area descriptor, where b_a is the area base and g_a is the area length. Protection primitive $P_a \leftarrow newArea(w_{new}, M, descr_a)$ allocates the memory area specified by $descr_a$, and returns a pointer $P_a = (M, d_a, descr_a)$ referencing this area. Area password d_a in P_a is generated by using relation $d_a = f_{descr_a}(m)$, where m is the value of the master password identified by argument M (see Section 3.2). Execution requires special password w_{new}.

Let $descr_s = (b_s, g_s, z_s)$ be a segment descriptor, where b_s is the segment base, g_s is the segment length, and z_s is an access authorization. Protection primitive $P_s \leftarrow newSegment(P_a, descr_s)$ allocates the segment specified by $descr_s$ in the memory area referenced by area pointer P_a, and returns a pointer $P_s = (M, d_s, descr_a, descr_s)$ referencing this segment, where $d_s = f_{descr_s}(d_a)$. As seen in Section 3.5, no special password is required to generate segment pointers. In fact, a process that holds a valid pointer for a given area is free to allocate segments in this area. For the given segment, access authorization z_s in $descr_s$ is configured according to the intended purpose, e.g., if the segment corresponds to a portion of an area storing executable code, the access authorization will include a single access right, *execute*.

3.6.3 Pointer registers

As seen in Section 3.4, a memory segment can be accessed only after the segment descriptor in a valid pointer referencing this segment has been transformed into an internal descriptor, and the result of this transformation has been loaded into a pointer register. An effect of this type can be obtained by executing protection primitive $loadPointerRegister(P_s, i, mask)$, where P_s is a segment pointer, i identifies pointer register PR_i, and $mask$ can be used to remove unnecessary access rights to weaken the access authorization in P_s, according to the principle of least privilege.

In terms of memory resources, least privilege implies that a pointer register referencing a given segment should be cleared on termination of the actions involving this segment. To this aim, protection primitive $clearPointerRegister(i)$ can be used, which invalidates pointer register PR_i by loading it with a null internal descriptor.

Let us consider the case of a subject S that holds pointer P_a referencing area a, and is aimed at accessing the entire area. To this aim, S generates a segment whose boundaries are the same as the area. Let $descr_a = (b_a, g_a)$ be the area descriptor. The descriptor of the new segment is $descr_s = (0, g_a, z_s)$, where z_s is the desired access authorization. S uses protection primitive *newSegment* to forge a pointer P_s for the new segment. Then, S issues protection primitive *loadPointerRegister* to transform P_s into an internal descriptor, and to load the result into a pointer register.

4 EXAMPLES OF APPLICATIONS

Extended pointers can be used to solve a variety of protection problems efficiently. In this section, we consider a few significant examples of these problems, where extended pointers are used to implement a bounded buffer, hierarchical ports, an access control list paradigm of access control, and *pointer repositories*, which are segments reserved to contain segment pointers. These examples are by no means exhaustive; they are only aimed at giving an indication of the flexibility of the extended pointer concept. In these examples, an access authorization is denoted by curly brackets, e.g. $\{rwx\}$ stands for an access authorization that includes all the three access rights *read*, *write* and *execute*, and $\{r\}$ stands for an access authorization that includes a single access right, *read*.

4.1 Bounded Buffer

Let us consider the classical problem of a bounded buffer. A producer process inserts data items into the buffer, and a consumer process extracts data items from the buffer. To simplify the presentation, we shall not consider the well-known aspects of this problem, which are related to process synchronization and mutual exclusion. Instead, we shall concentrate on memory sharing.

In a possible solution, the producer uses protection primitive *newSegment* to create two segments for the buffer, say s_1 and s_2. These segments are perfectly overlapped, that is, they have identical bases and lengths. However, the access authorizations are different. In fact, the producer writes data into s_1, and the consumer reads data from s_2. We have $descr_1 = (b, g, \{w\})$ and $descr_2 = (b, g, \{r\})$, where b and g denote the base and the length of the buffer, and the access authorizations are to write only in $descr_1$ for the producer, and to read only in $descr_2$ for the consumer. The corresponding segment pointers, as returned by *newSegment*, have the form $P_1 = (M, f_{descr_1}(d_a), descr_a, descr_1)$ and $P_2 = (M, f_{descr_2}(d_a), descr_a, descr_2)$, where a is the area containing the two segments, M is the name of the master password used to create a, and d_a is the password of a. The producer will grant P_2 to the consumer. Afterwards, the producer will use P_1 to write into the buffer, and the consumer will use P_2 to read from the buffer.

4.2 Hierarchical Ports

Let us consider a communication system featuring a server process that can receive data from client processes on a priority basis. The system features n communication ports. Each port is assigned a priority in the range from 0 (the highest priority) to $n-1$. Each client has a priority, and can transmit data to the server using the ports at the same or a lower priority. This means that a client at priority i can use the ports at priority $i, i+1, \ldots, n-1$.

Let us hypothesize that the size of a port is one page. The server process allocates a segment s aimed at containing all the ports. We have $descr_s = (b, n, \{r\})$, where b denotes the segment base, the segment length is n pages, one page for each port, and the access authorization is to read only. Furthermore, the server process creates a segment for each priority level. The descriptor of segment c_i reserved for priority level i is $(b+i, n-i+1, \{w\})$, that is, c_i has a size of $n-i+1$ pages and it includes ports $i, i+1, \ldots, n-1$. When a new client enters the communication system at priority level i, it is assigned an extended pointer for segment c_i, and the access authorization is $\{w\}$.

4.3 Access Control Lists

The access matrix, introduced in Section 2, can be represented in memory *by columns.* In this case, a list, called the *access control list*, is associated with each object [29]. The access control list ACL_j of object B_j consists of a collection of entries having the form (S_i, z), where S_i is the name of a subject, and z is an access authorization. When subject S_i attempts to access object B_j to execute a given operation *op*, the request must include a certification of the subject identity. Execution of *op* inspects ACL_j to find the entry for S_i, to verify that the z field of this entry includes the access rights necessary to execute *op*. If this is not the case, a negative acknowledgement is returned to S_i, and the execution of the operation fails.

Of course, a crucial requirement is that process identities cannot be counterfeited. We can obtain this result by using *null pointers.* A null pointer is an extended pointer that references a segment of size one page, positioned at the base of the area assigned to the given process to store the process code. The null pointer features no access authorization. Thus, if b denotes the area base, the descriptor of the null pointer has the form $(b, 1, \{\})$. Of course, the null pointer cannot be used to access the corresponding segment, as it includes no access right. Instead, in our example, it is transmitted to operation *op* to certify the identity of the issuing process.

4.4 Pointer Repositories

A *pointer repository* is a segment reserved to contain extended pointers. The segments referenced by the pointers in a pointer repository may be pointer repositories, or ordinary data segments. It follows that pointer repositories and data segments can be organized into a hierarchical tree structure, whereby the root and the intermediate nodes are pointer repositories, and the terminal nodes are data segments.

An interesting observation is that a segment pointer referencing a pointer repository may grant access authorizations stronger than that included in the segment pointer itself. In fact, if the segment pointer specifies access right *read* for the pointer repository, a subject that holds the segment pointer is in the position to extract the extended pointers contained in the pointer repository, to use them to access the segments they reference. In contrast, if the access right is *write*, the subject is only allowed to access the segment repository to add new extended pointers, to overwrite the existing extended pointers, or to delete them. *Write* does not permit any form of access to the segments referenced by these extended pointers.

5 DISCUSSION

5.1 Pointer Registers

As seen in Section 3.4, an address generated by the processor has the form (i, d), where quantity i selects a pointer register PR_i, and quantity d is a displacement in the segment referenced by the internal descriptor in PR_i. The contents of the pointer registers cannot be altered freely. Instead, these contents can only be modified by protection primitives *loadPointerRegister* and *clearPointerRegister*. A process that holds a given segment pointer and is aimed at accessing the segment referenced by this pointer issues *loadPointerRegister* preventively, to translate the segment pointer into an internal descriptor, and to load the result into a specific pointer register. Afterwards, this pointer register can be used for any sequence of accesses to the corresponding segment, until the contents of the pointer register are cleared, or are replaced with a new internal descriptor. If the pointer register array is dimensioned adequately, the necessity of a replacement will be comparatively rare. In a situation of this type, at any given time, the segments referenced by the registers in the pointer register array are a good approximation of the working set of the process running at that time.

5.2 Master Passwords

As seen in Section 3.5, special passwords w_{create} and w_{delete} are intended to be held by system components only, e.g. the process allocator and the process deallocator. It follows that an ordinary process, which does not possess w_{create}, is not entitled to create new master passwords. When a new process is started up, the process allocator creates one or more master passwords, and assigns these master passwords to this process. When a process terminates, the process deallocator deletes the master passwords that were assigned to this process. This means that all the area and segment pointers generated by using these master passwords are invalidated; it will no longer possible to use these extended pointers for successful memory accesses. In fact, as seen in Section 3.6.3, the segment referenced by a given segment pointer can be accessed only after the pointer has been transformed into an internal descriptor, and the result has been loaded into a pointer register. These actions are carried out by protection primitive *loadPointerRegister*, which validates the pointer by using the master password that was used to create that pointer. If the master password has been deleted, the validation is destined to fail.

5.3 Forging Extended Pointers

Let us consider a fraudulent subject running in the user mode, and aimed at forging an area pointer from scratch. The area pointer includes the name M of a master password, the area descriptor, and the value d_a of the area password. Quantity M can be copied from a valid extended pointer at little effort. The configuration of the area descriptor will be set in relation to the area that the area pointer should reference. The area password is given by relation $d_a = f_{descr_a}(m)$. This relation uses global function f, which is universally known. However, the value m of master password M is stored in the master table, which can only be accessed in the system mode. Of course, if master passwords are large, sparse, and chosen at random, the effort required to forge a valid area pointer by a brute force attack can be extremely high.

Let us now consider a subject that possesses a valid pointer for a given segment, and is aimed at forging a pointer for an area enclosing this segment. The segment pointer includes the value d_s of the segment password, which is expressed in terms of the value d_a of the area password, according to relation $d_s = f_{descr_a}(d_a)$. But global function f is one-way. This means that is computationally unfeasible to invert f to evaluate d_a starting from d_s. In this case, too, a solution is to use a value chosen at random, but the probability of success is vanishingly low.

5.4 The Revocation Problem

A subject that holds an area or segment pointer is free to grant a copy of this pointer to another subject. In turn, the recipient can transfer the extended pointer further. As a result, it may be hard to keep track of the position in memory of all the copies of the extended pointer. The revocation of access authorizations is a related issue. The original subject should be given the ability to review the pointer distribution, and revoke the access authorizations from the recipients.

Several solutions to this review and revocation problem have been conceived in the past with special reference to capability systems. A propagation graph can be constructed, which links all the access authorizations for the same given object [30], [31]. Temporary access authorizations can be used to force renewal, or automatic revocation [32]. A centralized reference monitor can be associated with each given object to keep track of all the subjects that hold an access authorization for this object [33]. All these solutions tend to adversely affect simplicity in the distribution of access rights.

In our system, the revocation of access authorizations is supported by the ability to delete the master passwords. In fact, if we delete a given master password, all the area and segment pointers defined in terms of this master password are invalidated; it will be no longer possible to use the area pointers to allocate new segments, and the segment pointers to access the corresponding memory pages. This revocation mechanism is *transitive* [30], that is, the effects of a revocation of an access authorization propagates automatically to all the subjects that hold this access authorization. In fact, a copy of an extended pointer is indistinguishable from the original.

As seen in Section 3.6.2, the arguments of protection primitive *newArea* include an area descriptor. This primitive places no limitations on the contents of this descriptor. Consequently, it is always possible to allocate two or more areas that overlap in memory, partially or totally. In a situation of this type, two or more segments belonging to different memory areas can share a common set of memory pages. If these areas were allocated by using different master passwords, and we delete one of these master passwords, all the pointers for the segments in the corresponding area are revoked; it will be no longer possible to take advantage of these pointers to access the segments. However, the validity of the pointers referencing the other overlapped segments is not affected, and the corresponding memory pages remain accessible by using these pointers. Thus, we are the presence of a form of *independent* [30] revocation of access authorizations, whereby access rights received from independent sources can be revoked independently of each other.

Suppose that primitive *loadPointerRegister* has been used to transform a pointer into an internal descriptor, and to load the result into a pointer register, as was illustrated in Section 3.6.3. If the pointer was created by using a given master password, and this master password is subsequently deleted, the contents of the pointer register are not affected by the deletion. This means that the pointer register can be used to access the segment even after deletion of the master password, until the register contents are replaced by a new execution of primitive *loadPointerRegister*, or are cleared by execution of primitive *clearPointerRegister*. An attractive property of this form of *delayed* revocation is that it never causes object inconsistencies, as may be the case for *immediate* revocation mechanisms, if an access authorization for a given object is cancelled while an operation is being executed on that object [30].

If we delete a master password that was used to allocate a given memory area, we invalidate all the pointers referencing this area. If we subsequently use a different master password to allocate a new area including the same memory pages, the validity the previous pointers, which referenced the old area, is not renewed. Similar considerations can be made for segment pointers.

5.5 Extended pointers and password capabilities

Password capabilities are an application of the password concept that received much attention in the past [25], [34], [35], [36], [37]. A password capability is a protected pointer having the form (B_j, w), where B_j is the identifier of a protected object, and w is a password. A set of passwords is associated with each object, one password for each access authorization defined for this object. A subject that is aimed at executing an operation on a given object must present a password capability referencing this object. If the password in that password capability matches one of the password associated with the object, and the access rights in the corresponding access authorization permit execution of the intended operation, the operation is actually executed, otherwise execution fails.

If passwords are large, sparse and chosen at random, the processing time cost for a fraudulent process to guess a valid password to forge a password capability from scratch by a brute force attack can be extremely high. Thus, password capabilities are a solution to the segregation problem, and they constitute an important improvement on the concept of a capability. Furthermore, if one of the password associated with a given object is changed, all the password capabilities defined in terms of that password are no longer valid. This is an effective solution to the problem of the review and revocation of access authorizations.

A drawback of password capability systems is password proliferation. Many passwords should be associated with the given object to exercise protection at a high level of granularity, for many different access authorizations. Inappropriate complexity follows in access right management, and the resulting memory cost for password storage can be a high fraction of the total, especially for small objects and large passwords.

Furthermore, in their original formulation, password capabilities suffer from the lack of mechanisms for weakening and reduction. A subject that holds a given password capability has no means to weaken this password capability to include less access rights. Similarly, it is impossible for a given subject to reduce a password capability to reference a fraction of the original object, e.g., for a password capability referencing a given memory area, a segment in this area.

The extended pointer concept, introduced in this paper, can be seen as a revisitation of the password capability concept that applies to memory areas and segments. We solve the password proliferation problem by taking advantage of a parametric one-way function, the global function. By executing protection primitive *newArea*, a single master password allows the area manager to allocate one or more memory areas for a given subject. By executing protection primitives *newSegment*, the subject can allocate an unlimited number of segments in each of these areas. As seen in Section 3.6.2, the arguments of *newSegment* include a segment descriptor, which specifies the base and the length of the new segment, and an access authorization. A subject that possesses the pointer for a given area and is aimed at a weak authorization for a segment in this area simply issues *newSegment* to create the new segment. If the subject is aimed at a segment reduction, it creates a new segment with the same base and authorization, and a reduced length.

6 CONCLUDING REMARKS

With reference to a single address space paradigm of memory reference, we have identified a set of mechanisms aimed at preserving the integrity of the memory space reserved for a given process from erroneous or malevolent access attempts originated from other processes. These mechanisms guarantee a level of protection that is, in many respect, superior to that of a traditional multiple virtual space environment. An essential requirement has been that each process should have a complete control over any form of access that involves its own memory space. In our approach:

- A universally known, parametric one-way function, the global function, is used to assign passwords to areas and segments.
- The protection system maintains a single table for the administration of access authorizations, the master password table, where the names and the values of all master passwords are recorded.
- An *ad hoc* circuitry for address translation supports a pointer-based mechanism for memory reference and protection at the hardware level. This circuitry includes a set of special registers, the pointer registers, aimed at storing segment pointers in an internal form.
- A small set of primitives, the protection primitives, forms the process interface of the protection system. These primitives make it possible to create new master passwords, to delete the existing master passwords, to allocates memory areas and segments, and to load segment pointers into the pointer registers.

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

- Taking advantage of the global function, a single master password is sufficient to generate the passwords for the areas of a given subject, and for an unlimited number of segments in these areas, with the desired access authorizations. This is a solution to both the password proliferation and the password weakening problems.

- A subject that holds a pointer for a given area can generate pointers for segments of arbitrary lengths within the boundaries of this area. This is a solution to the password reduction problem.
- If master passwords are large, sparse, and chosen at random, the processing time cost for a fraudulent subject to guess a valid master password value to forge an area pointer by a brute force attack can be prohibitive. Transformation of a pointer for a given segment into a valid pointer for an area enclosing this segment is prevented by the non-invertibility property of the global function.
- The ability to delete the master passwords supports the review and revocation of access authorizations. The resulting revocation mechanism possesses interesting properties; it is transitive, independent, and delayed.

APPENDIX A

This appendix is aimed at illustrating the actions caused by the execution of each protection primitive. To simplify the presentation, we shall omit the details concerning well known activities, e.g. the generation of area and segment pointers (see Section 3.2), and extended pointer validation (see Section 3.3).

$M \leftarrow createMasterPassword(w_{create})$

1. Quantity w_{create} is validated; it should be the value of the special password that grants special access right *create*. If this is not the case, execution generates a protection exception, and fails.
2. The identifier M and the value m of a new master password are generated, and are inserted into a free entry of the master password table. Quantity M is returned to the caller.

In step 2, a simple method to generate master password identifiers is a sequential generation. A master password counter is set to 0 as part of the activities related to system initialization. The identifier of the new master password is taken from this counter, and then, the value of the counter is incremented by 1. Master password values should be generated at random, sparse, and large, according to the security requirements of the system.

$deleteMasterPassword(w_{delete}, M)$

1. Quantity w_{delete} is validated; it should be the value of the special password that grants special access right *delete*. If this is not the case, execution generates a protection exception, and fails.

2. The master password table is accessed, and the entry reserved for master password M is cleared.

$P_a \leftarrow newArea(w_{new}, M, descr_a)$

1. Quantity w_{new} is validated; it should be the value of the special password that grants special access right *new*. If this is not the case, execution generates a protection exception, and fails.
2. The master password table is accessed to find the entry reserved for primary password M. The value m of this primary password is extracted from this entry.
3. Area pointer $P_a = (M, d_a, descr_a)$ referencing the new area a is assembled by using relation $d_a = f_{descr_a}(m)$ (see Section 3.2). This area pointer is returned to the caller.

$P_s \leftarrow newSegment(P_a, descr_s)$

1. The master password table is accessed to find the entry reserved for master password M specified by area pointer $P_a = (M, d_a, descr_a)$. The value m of this primary password is extracted from this entry.
2. Relation $d_a = f_{descr_a}(m)$ is used to validate P_a (see Section 3.3). If validation is unsuccessful, execution generates a protection exception, and fails.
3. Area descriptor $descr_a = (b_a, g_a)$ and segment descriptor $descr_s = (b_s, g_s, z_s)$ are considered to verify that inclusion condition $b_s + g_s \leq g_a$ is verified, that is, the new segment is completely contained within the boundaries of area a (see Section 3.1 and Figure 1). If this is not the case, execution generates an addressing exception, and fails.
4. Segment pointer $P_s = (M, d_s, descr_a, descr_s)$ referencing the new segment s is assembled by using relation $d_s = f_{descr_s}(d_a)$ (see Section 3.2). This segment pointer is returned to the caller.

$loadPointerRegister(P_s, i, mask)$

1. Segment pointer $P_s = (M, d_s, descr_a, descr_s)$ is considered, where $descr_a = (b_a, g_a)$ and $descr_s = (b_s, g_s, z_s)$. The master password table is accessed to find the entry reserved for master password M. The value m of this master password is extracted from this entry.
2. Relation $d_s = f_{descr_s}(f_{descr_a}(m))$ is used to validate segment password d_s in P_s (see Section 3.3). If validation is unsuccessful, execution generates a protection exception, and fails.
3. Internal descriptor $(b_a + b_s, g_s, z_s \wedge mask)$ is constructed, and is finally loaded into pointer register P_i.

ACKNOWLEDGEMENT

This work has been partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the *CrossLab* project (Departments of Excellence).

REFERENCES

[1] L. Lopriore, "Protection structures in multithreaded systems," *The Computer Journal*, vol. 56, no. 4, pp. 478–496, 2012.

[2] C. H. Park, T. Heo, and J. Huh, "Efficient synonym filtering and scalable delayed translation for hybrid virtual caching," in *Proceedings of the 43rd International Symposium on Computer Architecture*, (Seoul, Republic of Korea), pp. 217–229, IEEE Press, June 2016.

[3] X. Qiu and M. Dubois, "The synonym lookaside buffer: a solution to the synonym problem in virtual caches," *IEEE Transactions on Computers*, vol. 57, no. 12, pp. 1585–1599, 2008.

[4] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," *Operating Systems Review*, vol. 36, no. 5, pp. 304–316, 2002.

[5] B. Leslie, N. FitzRoy-Dale, and G. Heiser, "Encapsulated user-level device drivers in the Mungi operating system," in *Proceedings of the Workshop on Object Systems and Software Architectures*, (Victor Harbor, Australia), pp. 16–30, January 2004.

[6] L. Lopriore, "Memory protection in embedded systems," *Journal of Systems Architecture*, vol. 63, pp. 61–69, February 2016.

[7] D. S. Miller, D. B. White, A. C. Skousen, and R. Tcherepov, "Lower level architecture of the Sombrero single address space distributed operating system," in *Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing and Systems*, (Dallas, Texas, USA), November 2006.

[8] C. Van Schaik and G. Heiser, "High-performance microkernels and virtualisation on ARM and segmented architectures," in *Proceedings of the First International Workshop on Microkernels for Embedded Systems*, (Sydney, Australia), March 2007.

[9] S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "Access control: principles and solutions," *Software – Practice and Experience*, vol. 33, no. 5, pp. 397–421, 2003.

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

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

[12] G. Saunders, M. Hitchens, and V. Varadharajan, "Role-based access control and the access control matrix," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 4, pp. 6–20, 2001.

[13] W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong, "Access control in collaborative systems," *ACM Computing Surveys*, vol. 37, no. 1, pp. 29–41, 2005.

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

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

[16] P. G. Neumann and R. J. Feiertag, "PSOS revisited," in *Proceedings of the 19th Annual Computer Security Applications Conference*, (Las Vegas, NV, USA), pp. 208–216, IEEE, December 2003.

[17] J. D. Woodruff, "CHERI: a RISC capability machine for practical memory safety," Tech. Rep. UCAM-CL-TR-858, University of Cambridge, Computer Laboratory, July 2014; http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-858.pdf.

[18] L. Vilanova, M. Ben-Yehuda, N. Navarro, Y. Etsion, and M. Valero, "CODOMs: protecting software with code-centric memory domains," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 469–480, 2014.

[19] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. Murdoch, R. Norton, M. Roe, S. Son, and V. Munraj, "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), IEEE, May 2015.

[20] J. Brown, J. Grossman, A. Huang, and T. F. Knight Jr, "A capability representation with embedded address and nearly-exact object bounds," tech. rep., Project Aries, ARIES-TM-005, Artificial Intelligence Laboratory, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 2000; http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-05.pdf.

[21] 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*, (Minneapolis, Minnesota, USA), pp. 341–348, IEEE, May 1981.

[22] 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*, (Minneapolis, MN, USA), pp. 457–468, IEEE, June 2014.

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

[24] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, September 1975.

[25] 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*, (Sydney, Australia), pp. 25–34, ACS, January 2015.

[26] W. Trappe, J. Song, R. Poovendran, and K. J. Liu, "Key management and distribution for secure multimedia multicast," *IEEE Transactions on Multimedia*, vol. 5, no. 4, pp. 544–557, 2003.

[27] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, pp. 770–772, November 1981.

[28] R. S. Sandhu, "Cryptographic implementation of a tree hierarchy for access control," *Information Processing Letters*, vol. 27, no. 2, pp. 95–98, 1988.

[29] L. Lopriore, "Access control lists in password capability environments," *Computers & Security*, vol. 62, pp. 317–327, September 2016.

[30] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, pp. 575–586, November 1979.

[31] 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*, (Miami Beach, Florida, USA), pp. 341–352, IEEE, December 2007.

[32] 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*, (Reno, NV, USA), pp. 1–12, IEEE, November 2007.

[33] J. S. Shapiro and N. Hardy, "EROS: a principle-driven operating system from the ground up," *IEEE Software*, vol. 19, no. 1, pp. 26–33, 2002.

[34] M. D. Castro, R. D. Pose, and C. Kopp, "Password-capabilities and the Walnut kernel," *The Computer Journal*, vol. 51, no. 5, pp. 595–607, 2008.

[35] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey, “Lightweight shared objects in a 64-bit operating system,” in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, (Vancouver, British Columbia, Canada), pp. 397–413, ACM, October 1992.

[36] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, and J. Liedtke, “The Mungi single-address-space operating system,” *Software – Practice and Experience*, vol. 28, pp. 901–928, July 1998.

[37] L. Lopriore, “Password capabilities revisited,” *The Computer Journal*, vol. 58, pp. 782–791, April 2015.