Protection structures in multithreaded systems

Lanfranco Lopriore

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

Abstract — We consider a single-address-space system which implements a form of segmentation with paging within the framework of the multithreaded model of program execution. A salient problem of a system of this type is the definition of the set of mechanisms enforcing memory protection. We present a paradigm for protection system design that is based on the well-known concepts of protection domains and access rights. The resulting environment guarantees an effective separation of the memory resources of the different processes, whose loosely-coupled interactions correspond to explicit actions of information sharing. Within the boundaries of a single multithreaded process, a less stringent protection requirement is to confine the consequences of a programming error in the thread that originated the error. These results are obtained by taking advantage of techniques of symmetric-key cryptography to represent access privileges in memory at the level of the single pages that form a segment.

Keywords — process, protection, revocation, single address space, symmetric-key cryptography, thread.

1.   INTRODUCTION

In a traditional virtual memory environment, each process references its own virtual space. An address generated by a given process corresponds to an information item which is different from that identified by the same address in the virtual space of a different process. This virtual space separation enforces protection; a process has no means of accessing the private information items of any other process. On the other hand, if an object is shared between two or more processes, complex synonym problems arise, for instance, in a virtual-address cache (as multiple copies of the same object may be stored in different virtual cache positions) and in the circuitry for virtual-to-physical address translation (e.g., the translation lookaside buffer) [1], [2], [3]. A solution is to allocate the shared object at the same address in the virtual space of each process involved in the sharing (presently, and in the future) [4]. Alternatively, it is possible to map the physical memory frame that contains the shared object to the respective local pages; in this case, the shared object must be located at the same address within each page [2]. Both solutions must be supported by explicit intervention of the operating system.

In a different approach, all processes share a single, virtual address space, and no separation exists between the virtual spaces of different processes [5], [6], [7]. In this approach, information sharing between processes is straightforward – a given process will be able to access a shared object by simply using the object address – but provision of mechanisms for private object protection is mandatory.

We shall refer to a multithreaded environment in which each process gives rise to concurrent, independent execution flows called *threads* [8], [9]. An salient feature of an environment

of this type is that a switch of the execution flow between two threads of the same process is not overwhelmed by the high time costs that are usually connected with context switches between different processes, to save the processor state into the descriptor of the original process and then restore the processor state with quantities taken from the descriptor of the new process. The threads of the same process operate on a common pool of memory resources; interactions between these threads are frequent, and should be supported by information sharing mechanisms at low cost in terms of processing time. On the other hand, interactions between different processes are comparatively rare, and efficiency in information sharing between them is not a stringent requirement. Thus, the two different interaction levels, between the threads of the same process and between different processes, correspond to dissimilar efficiency requirements.

As far as protection is concerned, the protection system should be able to limit the consequences of malevolent attacks to the private information items of a given process that originate from a different process. Any illegitimate access should be blocked, and any attempt to protection privilege stealing should be prevented. This means that the protection system should guarantee an effective separation of the memory resources of the different processes, whose loosely-coupled interactions should correspond to explicit actions of information sharing. On the other hand, within the boundaries of the same multithreaded process, a less stringent protection requirement is to confine the consequences of a programming error in the thread that originated the error. Consider a process that supports dynamic extensibility in the form of plug-ins implementing new functionalities, for instance [10]. Plug-ins are usually considered trustworthy, but not free from errors [11]. If the main program and a plug-in share access to a common pool of objects, we are in the presence of a security vulnerability.

With reference to a single-address-space environment that supports the notion of segmentation with paging, we shall present a solution to the problems outlined above that is based on the application of techniques of symmetric-key cryptography [12], [13]. The rest of this paper is organized as follows. Section 2 presents our paradigm of memory reference and protection, which is based on the well-known concepts of protection domains and access rights [14], [15], [16]. Section 3 describes an implementation scheme for the protection system that relies on *ad-hoc* hardware inside the processor and the memory management unit, and takes advantage of techniques of symmetric-key cryptography to represent access privileges in memory at the level of the single pages that form a segment. We shall show that in our approach cryptography is hidden in the implementation level, behind the application program interface as defined by a set of system primitives, the *protection primitives*. Section 4 outlines the relation of our work to previous work and discusses the proposed system from the point of view of the design goals introduced above. Section 5 gives concluding remarks.

2.   THE PROTECTION SYSTEM

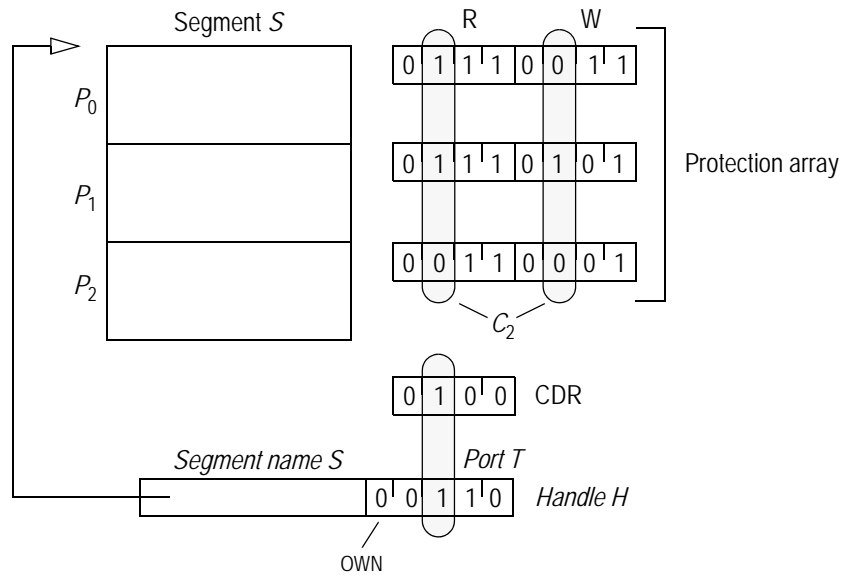We shall refer to a virtual memory system that combines segmentation and paging. The vir-

Figure 1. A three-page segment *S*, its protection array, current domain register CDR and a handle *H* referencing *S*.

tual space is partitioned into fixed-size *pages*. The primary memory is partitioned into frames, and the size of a frame is equal to the page size. The secondary memory is partitioned into blocks, and the size of a block is equal to the page size. A page is the elementary unit of information transfer between the primary memory and the secondary memory. A *segment* is a collection of pages that are contiguous in the virtual space. The number of pages that form a given segment is called the *segment size,* and is segment specific. Segments are aimed at containing information items that are logically correlated at the level of the program structure, e.g., a program module.

2.1 Protection contexts and handles

In our protection paradigm, access control is exercised at the level of a single page. Two access rights are defined on a page, the READ access right that makes it possible to read the page contents and the WRITE access right that makes it possible to modify these contents. The protection system allows us to define a limited number of *protection contexts* C_0, C_1,..., C_w, where quantity w is protection system specific and is the same for all segments. For each given segment *S*, a protection context specifies a collection of access rights for the pages that form this segment. An exception is the w-th protection context C_w, called the OWN protection context, which corresponds to a single access right, the OWN access right defined at the segment level. Possession of the OWN access right for a given segment implies possession of both the READ and the WRITE access rights for all the pages that form this segment.

Let us refer to the example illustrated in Figure 1. Segment *S* is formed by three pages. The specification of the protection contexts for *S* takes the form of a *protection array* featuring one element for each page. The element for the generic page P_j is partitioned into two *protection*

fields, the R field and the W field. Each protection field consists of $w - 1$ bits, which are numbered starting from the rightmost position. If asserted, the i-th bit of the R field, $i = 0, 1,..., w - 2$, specifies that the i-th protection context C_i contains the READ access right for page P_j, and similarly for the W field and the WRITE access right. It follows that the extent of the i-th protection context of segment S is specified by the i-th column of the R and the W fields of the protection array.[1,2] In the example of Figure 1, we have a three-page segment S and five protection contexts ($w = 5$). The first of them, C_0, contains both the READ and the WRITE access rights for all the pages. Protection context C_1 contains the READ access right for all the pages and the WRITE access right for page P_0. Protection context C_2 contains the READ access right for pages P_0 and P_1, and the WRITE access right for page P_1. Protection context C_3 specifies no access rights at all. Being defined at segment level rather than at page level, protection context C_4 is not included in the protection array. This protection context implicitly contains the OWN access right for segment S, corresponding to both read and write access permissions for all the pages that form S.[3]

A *port* specifies an access privilege in terms of one or more of protection contexts. We shall use a square bracket notation to indicate the protection contexts that form the given port, e.g., $T = [1, 2]$ indicates that port T is formed by protection contexts C_1 and C_2. Possession of a port for a given segment is certified by possession of a *handle* referencing this segment. Handle H is a pair $\{S, T\}$, where S is a segment name and T is a port. In a handle, the port takes the form of a w-bit natural number that is interpreted as a bit vector. If bit i of port T is asserted, $i = 0, 1, ..., w - 1$, then the port includes the i-th protection context C_i. In this implementation, port $T = [1, 2]$ takes the form of a natural number featuring bits 1 and 2 asserted and all the other bits cleared. A port featuring bit $w - 1$ asserted grants the OWN access right on the corresponding segment.

2.2 Protection domains

A *protection domain* is the specification of a subset of all protection contexts that acts as a limitation on the protection privileges held by a thread associated with that domain. At any given time, the protection domain associated with the thread running at that time is called the

---

1. A single page size is a simplifying assumption that does not affect generality. Even if the hardware supports a single page size, the protection system may well support multiple page sizes. A result of this type can be easily obtained by designing the hardware for small pages, which means a protection array with many elements. The protection system will consider consecutive pages as forming a single, large page, by replicating the bit configuration of the corresponding elements of the protection array. Our system does not provide protection at a granularity level smaller than a single page. On the other hand, we may wish to take advantage of small pages to support forms of fine-grained data protection.

2. In a well-known protection model, the protection state is depicted in the form of an *access matrix* featuring a row for each protected object and a column for each protection context [14], [17]. Element in column i and row j of the access matrix specifies the access rights included in the i-th protection context on the j-th object. The protection array is a hardware implementation of an access matrix in which the protected objects are the segment pages. The element of the access matrix for the i-th protection context and j-th page corresponds to the i-th bit of the two protection fields, the R field and the W field, of the j-th element of the protection array.

current domain. Suppose that thread q holds handle $H = \{S, T\}$, and let D be its current domain. The possibility to take advantage of the access privileges granted by H is restricted to those contexts specified by port T that are also part of the current domain D. Thus, q may take full advantage of H only if D includes all the protection contexts that form T. A single exception is the OWN context, which is implicitly included in the current domain; this means that a port including the OWN context always grants full access rights to the corresponding segment.

At any given time, a register of the central processor, the *current domain register* CDR, specifies the composition of the current domain. The size of this register is w - 1 bits; if the i-th bit is asserted, then the current domain contains the i-th protection context, C_i. In the example of Figure 1, the current domain contains a single protection context, C_2. The port of handle H includes protection contexts C_1 and C_2, however C_1 is not part of the current domain (bit 1 of CDR is cleared) and is masked out. It follows that the running thread can only take advantage of the access rights in protection context C_2, i.e., the READ access right for pages P_0 and P_1, and the WRITE access right for page P_1.

2.3 Encrypting handles

Handles are never stored in memory in plaintext. Instead, they are always stored in the form of unintelligible ciphertext. In the transformation, we take advantage of a double encryption and symmetric-key ciphers. In detail, we associate a *segment encryption key* k_S with each segment S. A further key, called the *process encryption key* k_Q, is associated with each process Q, and is shared by all the threads that form this process. At any given time, a register of the processor, the *process key register* PKR, contains the key of the father process of the running thread.

From now on, we shall use the * symbol to denote a ciphertext. Let $H = \{S, T\}$ be a handle, q be the thread holding this handle, and Q be the father process of q. Furthermore, let S^* be the result of encrypting quantity S by using a symmetric-key cipher with key k_Q, and let T^* be the result of encrypting quantity $\{S^*, T\}$ by using a symmetric-key cipher with key k_S. Quantity H^* corresponding in ciphertext to handle H is given by relation $H^* = \{S^*, T, T^*\}$ (Figure 2). Quantity T^* is called the *validation field*, and is aimed at validating H^*.

Figure 3 shows the reverse conversion of handle $H^* = \{S^*, T, T^*\}$ into the corresponding plaintext H. Process encryption key k_Q is used to decrypt quantity S^* into segment name S. Key k_S associated with S is then used to encrypt pair $\{S^*, T\}$, and the result is compared with

---

3. The idea of limiting the set of accessible pages to a subset of all pages on a per-process basis is certainly not new. For instance, in the Itanium processor architecture, each entry of the translation lookaside buffer (TLB) is tagged with a *protection key* (not cryptographic) that makes it possible to control the access rights granted by that entry [18]. The running process is associated with a set of registers, the *protection key registers*. When the virtual address selects a TLB entry, the protection key of this entry is compared with each protection key register. If a match is found, the access is permitted and the access rights are specified by the matching register, otherwise an exception is raised. Protection keys are especially important as far as page sharing is concerned. They allow an easy implementation of the protection domain concept. Processes with different permissions are allowed to access shared objects while using the same TLB entries.
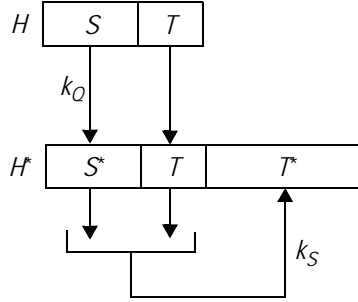
Figure 2. Translation of handle $H = \{S, T\}$ into the ciphertext form $H^* = \{S^*, T, T^*\}$.
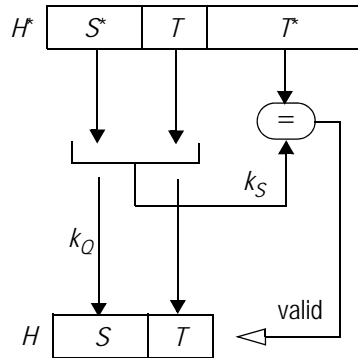

Figure 3. Validation of handle $H^* = \{S^*, T, T^*\}$ and subsequent translation into the plaintext form $H = \{S, T\}$.

T^* to validate H^*; validation is successful only if a match is found. If validation is successful, quantity H is given by pair $\{S, T\}$.

Thus, quantity S^* is obtained by using an encryption key, k_Q, which is *process* specific and is independent of the segment, whereas quantity T^* is the result of a double encryption involving both k_Q and an encryption key, k_S, which is *segment* specific and is independent of the process.

3. PROTECTION SYSTEM IMPLEMENTATION

The protection system is conceived to be implemented by a hardware/software complex that includes *ad-hoc* hardware inside the processor and the memory management unit (MMU), and a set of *protection primitives* that form the process interface of the protection system (Table I). In the following, we shall show an implementation scheme that hypothesizes a possible division of the protection system functionalities between the hardware level and the software level. Different implementations may well be devised; however, every implementation should encapsulate the protection system so that processes are prevented from tampering the system and altering its intended behavior, even if the underlying algorithms and implementation details are publicly known. An effective solution relies on the usual concept of the two control modes, user and privileged. The protection primitives are executed as privileged operations. In this ap-

Table I. Protection primitives.

---

hReg ← hLoad(*addr*)

Converts the ciphertext handle stored in memory location *addr* into a plaintext, and loads this plaintext into handle register *hReg*.

---

addr ← hStore(*hReg*)

Converts the plaintext handle stored in handle register *hReg* into a ciphertext, and stores this ciphertext into memory location *addr*.

$addr_2$ ← *newSegment*(*size*, $addr_1$)

Allocates a new segment of the given *size* and stores a ciphertext handle for this segment into memory location $addr_2$; the port in this handle includes all the protection contexts. The initial configuration of the protection array of the new segment is taken from the memory region at address $addr_1$.

---

deleteSegment(*hReg*)

Deletes the segment referenced by *hReg*. Requires access right OWN in *hReg*.

---

addr ← hReduce(*hReg*, *msk*)

Eliminates protection contexts from the handle in *hReg* as specified by *msk*, converts the result into a ciphertext, and stores this ciphertext into memory location *addr*.

---

addr ← hTranscode($hReg_2$, *msk*, $hReg_1$)

Eliminates protection contexts from the handle in $hReg_1$ as specified by *msk*, converts the result into a ciphertext by using the key of the process identified by the process descriptor referenced by $hReg_2$, and stores this ciphertext into memory location *addr*. Requires access right OWN in both $hReg_1$ and $hReg_2$.

---

readProtection(*addr*, *hReg*)

Copies the contents of the protection array of the segment referenced by *hReg* from the page table of this segment into the memory region at address *addr*.

---

writeProtection(*hReg*, *addr*)

Replaces the contents of the protection array of the segment referenced by *hReg*, stored in the page table of this segment, with quantities taken from the memory region at address *addr*. Requires access right OWN in *hReg*.

---

addr ← newSegmentKey(*hReg*)

Generates a new segment key and associates this key with the segment referenced by *hReg*. Uses the new key to convert the plaintext handle stored in *hReg* into a ciphertext, and stores this ciphertext into memory location *addr*. Requires access right OWN in *hReg*.

---

newProcessKey(*hReg*)

Generates a new process key and associates this key with the process corresponding to the process descriptor referenced by *hReg*. Requires access right OWN in *hReg*.

---

proach, a call to a protection primitive produces a system call that traps into the operating system kernel and invokes the protection system; the system call switches from user mode to privileged mode.

3.1 System tables

The information items relevant to process protection are gathered in a system table called the *process table*. This table features one entry for each process. The entry for a given process contains the process encryption key k_Q and the identifier S_Q of a special segment, called the *process descriptor*, which is associated with Q and identifies Q to the other processes. The size of a process descriptor is a single page, and physical memory space is not allocated for this segment. A single access right is defined on a process descriptor, i.e., the OWN access right. As will be illustrated later, process descriptors are used for handle transmission between different

processes.

In our hypothesis of a single address space, a single *segment table* ST is shared by all processes. The segment table features an entry for each segment allocated in the virtual space. The entry for a given segment S contains: i) the segment encryption key k_S; ii) the segment size, expressed in terms of the number of pages that form the segment; and iii) the memory address of the *page table* of the segment. The page table is aimed at storing information concerning both allocation of the virtual pages in the physical memory and page protection.

Inside the MMU, a *segment table cache* (STK) is aimed at containing the segment table entries relevant to recently-accessed segments, for fast access to these entries. STK is accessed by using a segment name S as the access key. If no STK entry matching S is found, STK resolves the miss by accessing the segment table and loading the contents of the segment table entry reserved for S into STK. If the segment table contains no entry for S, then quantity S does not identify an existing segment, and an exception of addressing violation is raised to the processor.

The page table of segment S features one entry for each page that forms S. The entry for page P contains: i) the two protection fields, R and W; ii) a presence bit V that specifies whether space has been allocated for P in the primary memory, or the page is stored only in the secondary memory; iii) the number F of the primary memory frame reserved for P; iv) the number B of the secondary memory block reserved for P. Inside the MMU, a *page table cache* (PTK) contains the page table entries relevant to recently-accessed pages. PTK is accessed by using a pair $\{S, P\}$ as the access key, i.e., the name of a segment and the name of a page in this segment.

It is worth noting that when the running process releases the processor and a new process is assigned the processor, the contents of both the segment table cache STK and the page table cache PTK are not invalidated. This is a consequence of the single address space paradigm: memory mapping information is independent of the process.[4]

3.2 Memory addressing

Inside the processor, a set of registers, the *handle registers*, are used for both memory addressing and protection. Each handle register is aimed at storing a handle in plaintext. Let *hReg* denote a handle register, and $H = \{S, T\}$ be the handle contained in *hReg*. We say that *hReg ref-*

---

4. In a typical organization of a translation lookaside buffer (TLB), a process identifier field is used in each TLB entry for storage of the memory mapping information relevant to more than a single process at a time [1], [19]. In this approach, a TLB register, which we shall call the *process identifier register* (PIDR), contains the identifier of process being executed at that time. When a process switch takes place, the identifier of the new process is written into PIDR. When the TLB is accessed to find the TLB entry corresponding to the present address, only those entries are considered whose process identifier field matches the contents of PIDR. In contrast, in our approach, the address translation information contained in both the segment table cache STK and the page table cache PTK is independent of the process. If two or more processes share an address space portion, the corresponding memory mapping information is not replicated. Consequently, the use of the overall cache capacity is significantly enhanced, and more cache space is available for the active processes.
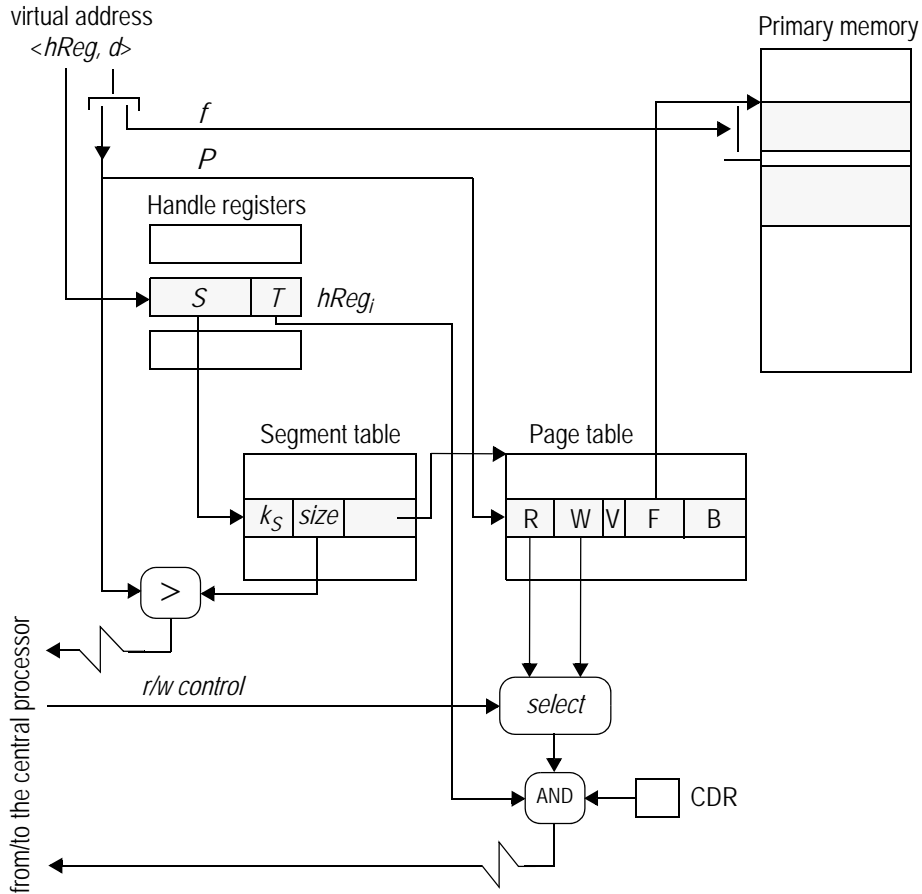
Figure 4. Translation of a virtual address into a physical address in the primary memory.

erences segment *S*. A virtual memory address *addr* consists of a pair *<hReg, d>*, where quantity *d* is a displacement in the segment referenced by *hReg* (Figure 4). The most significant bits of the displacement *d* identify a page *P* within *S*, and the least significant bits specify the offset *f* of the referenced information item in this page.

Let *q* be the running thread and suppose that *q* issues virtual address *addr* = *<hReg, d>*. Translation of *addr* into the corresponding physical address in the primary memory is preceded by the necessary validation of the access privileges held by *q*. In detail, the actions involved in the address translation are as follows:

1. The segment table ST is accessed and the entry ST_S corresponding to segment *S* is selected. The segment size is extracted from this entry and is compared with page number *P*. If *P* is greater than the segment size, an exception of addressing violation is raised to the processor; otherwise,

2. The page table PT of segment *S*, identified by a field of ST_S, is accessed to find the entry PT_P corresponding to page *P*. The contents of the R protection field of this entry (or, if the access is for write, the contents of the W protection field) are extracted from the table.

3. The bitwise AND of the protection field, CDR and port T is evaluated[5]. If the result is 0, and the most significant bit of T (corresponding to the OWN protection context) is cleared[6], the access right required to accomplish the access successfully is lacking, and an exception of violated protection is raised; otherwise,

— 9 —

4. Frame number F is paired with offset f to obtain the address of the referenced information item in the primary memory, and the memory access is finally accomplished.

3.3 Protection primitives

We shall now introduce a number of protection primitives, and we shall delineate the activities involved in the execution of each of them. We shall hypothesize that the given primitive is executed by thread q of process Q. To simplify presentation, we shall not mention self-explaining or obvious actions, such as the check of the access rights required to access a given memory location, and the activities of the two caches, the segment table cache STK and the page table cache PTK.

A first example of a protection primitive is the load primitive *hLoad*, which has the form $hReg \leftarrow hLoad(addr)$. This primitive allows us to load handle $H^* = \{S^*, T, T^*)$ from memory location *addr,* where it is stored in ciphertext form, into handle register *hReg*. Execution includes the actions, illustrated in Figure 3, which are necessary to validate H^* and convert it into the corresponding plaintext H. In detail, execution is as follows:

1. The primary memory is accessed and handle H^* is loaded from memory location *addr* into the processor.
2. The encryption key k_Q of the running process Q is read from the process key register PKR. This key is used to decrypt quantity S^* and obtain the name S of the segment referenced by H^*.
3. A search is made in the segment table to find the table entry reserved for segment S and extract the encryption key k_S of this segment.
4. Key k_S is used to convert pair $\{S^*, T\}$ into ciphertext, and the result is compared with quantity T^*. If a match is found, port T is validated; otherwise, an exception of violated protection is raised and execution fails.
5. Segment name S and port T are paired to form handle $H = \{S, T\}$. This handle is finally stored into handle register *hReg*.

The store primitive $addr \leftarrow hStore(hReg)$ allows us to store handle $H = \{S, T\}$ from handle register *hReg*, where it is contained in plaintext form, into memory location *addr*. Execution includes the actions, illustrated in Figure 2, which are necessary to convert H into the corresponding ciphertext H^*. In detail, execution is as follows:

1. The encryption key k_Q of the running process Q is read from the process key register PKR. This key is used to encrypt segment name S and obtain quantity S^*.
2. A search is made in the segment table to find the table entry reserved for segment S and extract the encryption key k_S of this segment.

---

5. As pointed out previously (see Figure 1), a memory access performed by taking advantage of a given port terminates successfully only if it is permitted by an access right in one of the protection contexts contained in the port, provided that this protection context is also part of the current domain, as specified by CDR.

6. It should be recalled that the OWN protection context in the port for a given segment grants both the READ and the WRITE access rights for all the pages that form the segment, and this protection context is always contained in the current domain (independently of CDR). It follows that if the most significant bit of port T is asserted, then the access right check is always successful.

3. Key k_S is used to convert pair $\{S^*, T\}$ into ciphertext quantity T^*.

4. Quantity $H^* = \{S^*, T, T^*\}$ is assembled and is finally stored into memory location *addr*.

Segment creation and deletion

The segment allocation primitive $addr_2 \leftarrow newSegment(size, addr_1)$ can be used to allocate a new segment. Argument *size* expresses the size (in pages) of the new segment. Argument $addr_1$ is the address of a memory region containing the initial configuration of the protection array of the new segment (i.e., the R and W protection fields of each page). Execution of this primitive stores a ciphertext handle for the new segment into memory location $addr_2$; the port in this handle will contain all the protection contexts, including the OWN context. The actions involved in the execution of *newSegment* are as follows:

1. The virtual memory manager reserves a virtual space area of the given *size* for a new segment and returns the name S of this segment.

2. The physical memory manager reserves a new entry for S in the segment table, and allocates a page table featuring one entry for each page that forms S. In the page table, the protection array is initialized with quantities taken from the memory region starting at address $addr_1$.

3. The encryption key k_Q of the running process Q is read from the process key register PKR. This key is used to encrypt segment name S and obtain quantity S^*.

4. A new segment key k_S is generated and is associated with segment S in the segment table.

5. Let T be a port containing a bit pattern of all 1's to indicate all protection contexts. Key k_S is used to convert pair $\{S^*, T\}$ into ciphertext quantity T^*.

6. Quantity $H^* = \{S^*, T, T^*\}$ is assembled and is finally stored into memory location $addr_2$.

In point 1 above, a simple strategy for virtual space allocation is a sequential allocation: the identifier of a given segment is equal to the identifier of the previous segment incremented by the size of the previous segment. This strategy can be effectively implemented by using a counter that at any given time contains the virtual address of the segment to be allocated next. After allocation of a new segment, the counter is incremented by the size of the new segment.

The segment deallocation primitive *deleteSegment(hReg)* deletes the segment referenced by handle register *hReg*. Execution of this primitive causes the physical memory manager to delete both the page table of S and the entry reserved for S in the segment table. This primitive requires the OWN access right in *hReg*

Physical memory allocation, deallocation and management are outside the scope of this paper, and will not be discussed.

Handle reduction

An access privilege can be transferred between two threads of the same process by a simple action of a handle copy. Suppose that q and q' are both threads of process Q, and q' holds handle H^*: q' will copy H^* into a memory segment shared with q, and q will read the handle from this segment. In fact, both these threads are associated with the same process key, k_Q. H^* was encrypted by using k_Q, and consequently, q will be able to execute the *hLoad* protection primi-

tive successfully, as *hLoad* will use k_Q to decrypt H^*. Handle *reduction* is the action of limiting the extent of a handle by eliminating part of the protection contexts from the port. An action of this type may be necessary before transferring the handle, to limit the access privileges of the recipient thread to a subset of the original privileges.

Let $H = \{S, T\}$ be the handle contained in handle register *hReg*. Protection primitive $addr \leftarrow hReduce(hReg, msk)$ eliminates protection contexts from port T as specified by mask *msk*, and saves the resulting ciphertext handle into memory location *addr*. *msk* is a bit pattern of the same size as a port; for each bit that is cleared in *msk*, execution of *hReduce* clears the corresponding bit in the port. In detail, execution is as follows:

1. The encryption key k_Q of the running process Q is read from the process key register PKR. This key is used to encrypt segment name S and obtain quantity S^*.

2. A search is made in the segment table to find the table entry reserved for segment S and extract the encryption key k_S of this segment.

3. Quantity T_1 equal to the bitwise AND of port T and mask *msk* is evaluated, and key k_S is used to convert pair $\{S^*, T_1\}$ into ciphertext quantity T_1^*.

4. Quantity $H^* = \{S^*, T_1, T_1^*\}$ is assembled and is finally stored into memory location *addr*.

Transcoding handles

Let q' be a thread of process Q' and q be a thread of process Q. If q' holds handle H^* and transmits a copy of this handle to q, no actual transfer of access privilege takes place. In fact, H^* was encrypted by using the encryption key $k_{Q'}$ of process Q', and consequently q will not be in the position to execute *hLoad* successfully, as this primitive will use the key k_Q of Q to decrypt the handle. Instead, H^* should be *transcoded* before being transferred to q, that is, it should be transformed into plaintext by using $k_{Q'}$, and then converted back to ciphertext by using k_Q.

The handle transcoding mechanism takes advantage of process descriptors. As has been anticipated in Subsection 3.1, a process descriptor is a special segment associated with each process and aimed at identifying the process; the entry of the process table reserved for process Q contains the identifier S_Q of the corresponding process descriptor.

Handle transcode proceeds as follows. Thread q' of process Q' uses *hLoad* to convert H^* into plaintext and load the result $H = \{S, T\}$ into a handle register $hReg_1$. Then, q' issues protection primitive $addr \leftarrow hTranscode(hReg_2, msk, hReg_1)$. The arguments of this primitive include a mask *msk* and the name $hReg_2$ of the handle register containing a handle $H_Q = \{S_Q, T_Q\}$ referencing process descriptor S_Q. Execution reduces the access privileges contained in handle H according to *msk*, converts the result into a ciphertext by using the key of the process Q identified by S_Q and stores the ciphertext into memory location *addr*. In detail, execution is as follows:

1. Port T of handle H and port T_Q of handle H_Q are inspected to ascertain whether both of them include the OWN protection context. If this is not the case, an exception of violated protection is raised and execution terminates; otherwise

2. A search is made in the process table for an entry containing quantity S_Q. The encryption key k_Q of process Q is extracted from this entry.

3. Key k_Q is used to encrypt segment name S and obtain quantity S^*.

4. A search is made in the segment table to find the table entry reserved for segment S and extract the encryption key k_S of this segment.

5. Quantity T_1 equal to the bitwise AND of port T and mask *msk* is evaluated, and key k_S is used to convert pair $\{S^*, T_1\}$ into ciphertext quantity T_1^*.

6. Quantity $H^* = \{S^*, T_1, T_1^*\}$ is assembled and is finally stored into memory location *addr*.

Thread *q'* will use *hTranscode* to store the resulting ciphertext handle H^* into a memory segment for which thread *q* holds the READ access right. Thread *q* will be now in the position to load H^* into a handle register successfully, as this handle is encrypted by using the key k_Q of its own father process, *Q*.

Of course, after receipt of handle H^*, thread *q* may well transcode H^* and transmit it further. On the other hand, the original owner of the handle, *q'*, can prevent any subsequent action of handle transmission by taking advantage of the mask in the *hTranscode* primitive and eliminating the OWN access right from the handle.

Access right revocation

Let *S* be the segment referenced by handle register *hReg*. Protection primitives *readProtection*(*addr, hReg*) and *writeProtection*(*hReg, addr*) make it possible to inspect and modify the protection array of *S*. *readProtection* copies the contents of the protection array from the page table of segment *S* into the memory region starting at address *addr*. *writeProtection* replaces the contents of the protection array in the page table of segment *S* with quantities taken from the memory region starting at address *addr*; this primitive requires the OWN access right in *hReg*. *readProtection* and *writeProtection* can be used to exercise a form of access right review and revocation at the level of the single pages of a given segment. By eliminating the access rights from one or more protection contexts, we revoke these access rights from all threads that hold handles including these protection contexts.

Indeed, our protection system supports further techniques for the revocation of access permissions. A process that holds a handle for a given segment with the OWN access right can change the key of this segment by issuing protection primitive *addr* ← *newSegmentKey*(*hReg*). Let *S* be the segment referenced by handle register *hReg*. Execution of this primitive generates a new segment key and associates this key with *S* in the segment table entry reserved for *S*. Then, the new key is used to convert the plaintext handle stored in *hReg* into a ciphertext, which is finally saved into memory location *addr*. This is now the only valid handle referencing *S* (indeed, any attempt to issue *hLoad* and load a handle generated by using the old segment password into a handle register is destined to fail).

Handle invalidation obtained by changing the key of a given segment is a process-independent action that affects any handle referencing this segment. In a different approach, we change the key of a given process to invalidate all the handles held by this process. Let S_Q be

the descriptor of process Q, and suppose that handle register $hReg$ references S_Q. Execution of the *newProcessKey*($hReg$) primitive generates a new process key and associates this key to Q. To this aim, the new key is inserted into the process table entry reserved for Q. Execution terminates successfully only if the handle in $hReg$ includes the access right OWN for S_Q.

4.   DISCUSSION, AND RELATION TO PREVIOUS WORK

4.1   Forging handles

A handle can be successfully used to access an information item in memory only after it has been loaded into a handle register. Processes cannot read or modify the contents of handle registers freely; instead, these registers can only be accessed in a protected fashion using the protection primitives. This aspect of handle register security is especially important as the contents of these registers are in plaintext (not encrypted).

Ciphertext handles are mixed in memory with ordinary data. It follows that a process is free to read and modify an existing ciphertext handle, or even to forge a new ciphertext handle from scratch. Suppose that process Q assembles handle $H^* = \{S^*, T, T^*\}$ by using arbitrary bit patterns for both the S^* and the T^* fields, and filling the T field with the bit pattern of all 1's that corresponds to full access privileges. In order to take advantage of H^* for memory access, Q will have to issue the *hLoad* protection primitive to load this handle into a handle register. Execution of *hLoad* uses the encryption key k_Q of process Q to decrypt quantity S^* and obtain segment name S. This action always terminates with success, however, if the virtual address space is large and sparsely allocated, the probability that quantity S be a valid segment name is negligible. In a situation of this type, the subsequent search in the segment table for the entry reserved for segment S is destined to fail, and an exception of addressing violation is generated. Even in the improbable case that S is valid, execution of *hLoad* proceeds by encrypting pair $\{S^*, T\}$ using the key k_S of segment S. Then, the result is compared with quantity T^*; in this comparison, the probability of a casual match is negligible. Thus, *hLoad* is destined to fail. In fact, the two events, the validity of S^* and the congruence of T^*, are largely independent. The probability that both these events occur is evanescently low. We may conclude that handle forging is virtually impossible.

An aspect of handle forging is a modification of the port of an existing, valid handle aimed at producing an undue amplification of access privileges. Let $H^* = \{S^*, T, T^*\}$ be a handle, T_1 be a port stronger than the original port T, e.g., a bit pattern of all 1's, and $H_1^* = \{S^*, T_1, T^*\}$ be the resulting, forged handle. The utilization of H_1^* for memory reference requires that H_1^* is loaded into a handle register. In the execution of the *hLoad* primitive, T_1 will be paired with S^* and the result will be encrypted by using the segment key for subsequent comparison with the validation field T^*. Of course, this validation is destined to fail.

We wish to remark that knowledge of a valid segment name does not represent a security hole. A segment can be successfully accessed in the primary memory only after a handle referencing this segment has been loaded into a handle register. The forging of a handle for a given

segment requires knowledge of both the process key and the segment key, but these keys are hidden to processes.

4.2 Stealing handles

Validity of a given handle is restricted to the process on behalf of which this handle was enciphered in memory. Let $H = \{S^*, T, T^*\}$ be a ciphertext handle generated by using the encryption key k_Q of process Q. Suppose that a second process Q' steals H from Q and tries to take advantage of H. To this aim, Q' issues the *hLoad* primitive to load H into a handle register. In the execution of *hLoad*, transformation of S^* into S will use the encryption key of the process issuing *hLoad*, that is, key $k_{Q'}$; whereas S^* was encrypted by using key k_Q. As result, *hLoad* will produce an arbitrary segment identifier S'. Even in the improbable case that segment S' is presently allocated in the virtual space, the subsequent action of handle validation is destined to fail.

It is worth to note that the three components of ciphertext handle $H^* = \{S^*, T, T^*\}$ do not need to be stored in contiguous memory locations. In fact, a process holding several ports for the same segment, and the corresponding validations, needs to maintain a single copy of quantity S^*, while keeping track of the associations between S^*, the ports and the validations in program logic. This is a consequence of the fact that a validation includes the indication of the corresponding segment; if a $\{T, T^*\}$ pair is associated with the wrong segment name, any attempt to load the resulting ciphertext handle into a handle register is destined to fail.

4.3 Thread-level protection

As seen in the Introduction, our protection model hypothesizes a set of loosely coupled processes that share access to a single address space. Each process holds access permissions for a fraction of the segments in the virtual space, and these access permissions are certified by handle possession. The sharing of a given segment between processes is obtained by means of the transmission of a handle for this segment, which must be preceded by an action of handle transcode (see Subsection 3.3), from the key of the process that grants the handle to the key of the process that receives the handle. The related costs in terms of execution times are mitigated by the low frequency of these actions, as follows from the loose degree of process coupling. On the other hand, between two threads of the same process no transcode activity is necessary, as both threads share the process key. In fact, from the point of view of protection, all the threads of the same process can be considered as sharing a common pool of handles, which is associated with the process rather than with the single thread.

We may conclude that the process keys guarantee an effective separation of the memory resources of the different processes. Between the threads of the same process, mechanisms are necessary to protect the memory regions of each thread, in application of the *principle of least privilege* [15], [20] according to which at any given time each software component should be given the smallest set of access privileges that is required at that time by that software component to carry out its job.

We associate a protection domain with each thread. The protection domain of the running thread is defined as a subset of all protection contexts whose composition is determined by the contents of the current domain register CDR. When a new thread is generated, its father process assigns this thread a suitable configuration for CDR. When a thread switch takes place, the current thread relinquishes the processor and a new thread is assigned the processor, the contents of CDR are replaced with the configuration corresponding to the domain of the new thread, as part of the actions connected with the thread switch.

We do not force application programs to adhere to a ubiquitous, pre-existing protection paradigm that is fixed for all applications. We shall now consider a variety of common protection paradigms, and we shall show that these paradigms are well supported by our protection system, where they can be implemented at little effort.

Hierarchical domains

In a hierarchical domain organization, the domains form a tree structure in which each domain at a directly higher level to one or more child domains inherits all the access rights of the child domains, recursively. We implement a structure of this type by reserving a protection context for each given domain: protection context C_i contains the access privileges associated with domain D_i that are not inherited from the child domains. For a parent domain, the import of the access privileges from the child domains is obtained by taking advantage of the current domain register CDR, as follows: if the running thread is assigned a domain D_i that has child domains, then in CDR we set the i-th bit (corresponding to the protection context C_i) and the bits that are asserted in the CDR configuration for each child domain; whereas if D_i is a leaf in the tree, then in CDR we set only the i-th bit.

Let us refer to domains D_0 and D_1 sharing a parent domain D_2, for instance. Suppose that D_0 contains the READ access right for page P_0, D_1 contains the READ access right for page P_1, and D_2 contains the WRITE access right for both pages P_0 and P_1 and inherits the access rights of the child domains D_0 and D_1. Figure 5a shows the corresponding configuration of the protection array and CDR for the three domains. Protection context C_2 is reserved for domain D_2, and similarly, contexts C_1 and C_0 are reserved for domains D_1 and D_0, respectively. In the configuration of CDR for domain D_0 we set only bit 0 (all the other bits are cleared); similarly, for domain D_1 we set only bit 1. For the parent domain D_2, we set bit 2 corresponding to C_2, and bits 0 and 1 corresponding to the CDR configuration for the child domains D_0 and D_1. So doing, the access rights of the child domains D_0 and D_1 are inherited by D_2.

A salient feature of a hierarchical domain organization is that if an access right is revoked from a child domain, revocation extends to the parent domain; this feature is well supported by the proposed implementation.

Protection rings

A protection ring is a special case of a hierarchical organization in which each parent domain has a single child. Thus, a domain at a given level in the hierarchy imports the access
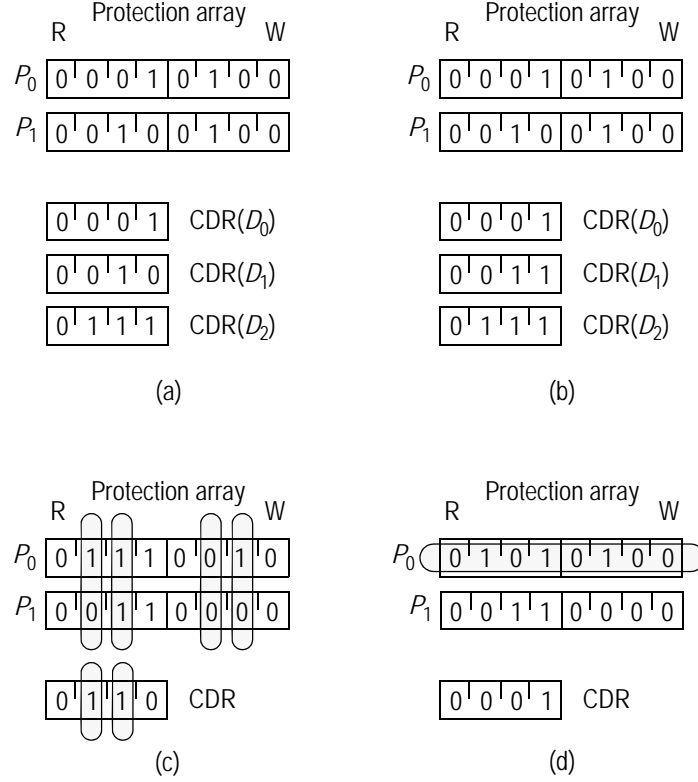
Figure 5. Configuration of the protection array and the current domain register CDR for a variety of protection paradigms: (a) hierarchical domains; (b) protection rings; (c) capability lists; (d) access control lists. The most significant bit of CDR, corresponding to the OWN protection context, is not shown.

privileges of all the domains at the lower levels. An organization of this type is usually depicted as a set of concentric rings; an inner ring corresponds to a domain at a higher privilege level than the outer rings.

In our protection system, a structure of this type will be implemented by reserving a protection context for each domain. Protection context C_i contains the access privileges associated with domain D_i that are not inherited from the outer domains. If the running thread is assigned domain D_i, then in CDR we set bit i corresponding to protection context C_i, and bits $0, 1,..., i - 1$ corresponding to the protection contexts reserved for the outer domains. So doing, D_i inherits the access rights of the outer domains.

Let us refer to an organization featuring three rings, for instance. Ring D_0 (the least privileged) contains the READ access right for page P_0. Ring D_1 contains the READ access right for page P_1 and imports the READ access right for page P_0 from D_0. Finally, ring D_2 (the most privileged) contains the WRITE access right for both P_0 and P_1 and imports the READ access right for these pages from D_0 and D_1. Figure 5b shows the corresponding configuration of the protection array and CDR for the three domains. For domain D_1, we set bits 0 and 1, for instance. So doing, the access rights of the outer domain D_0 (as included in protection context C_0) will be inherited by D_1.

Capability lists

A *capability* [17] is a form of protected pointer taking the form of a pair {object name, access rights}. The holder of a capability can access the object referenced by this capability and perform the actions that are made possible by the access rights specified by the capability. A *capability list* is a collection of capabilities. In the capability paradigm, a protection domain takes the form of a capability list; a thread running in a given domain can take advantage of all the capabilities in the list for object access.

As will be shown in the next subsection, at the interprocess interaction level, handles are a thorough replacement of capabilities. Within the boundaries of a single process, as far as interactions between threads are concerned, we support the capability model at page level by taking advantage of the protection fields and CDR, as follows: protection context C_i includes a capability for page P_j if the i-th bit of the R and/or the W fields of the protection array element for this page are asserted. Thus, in the protection array, the capabilities in C_i correspond to the two columns at the i-th position of the R and the W protection fields. The capability list associated with the current protection domain is formed by the columns of the protection array corresponding to the bits that are asserted in CDR.

In the configuration of Figure 5c, the current domain includes protection contexts C_1 and C_2. The capability list for the current domain is formed by the columns of the protection array at position 1 (corresponding to protection context C_1) and at position 2 (corresponding to protection context C_2) of the R and the W fields.

Access control lists

An access control list is a collection of pairs {domain name, access rights} that is associated with a protected object and states the domains that hold access permissions for this object [16]. In a multithreaded, page-oriented protection environment, when a thread performs an access attempt to a given page the access control list of this page is inspected to ascertain whether the domain of this thread includes the access right permitting the access. In our protection system, the access control list of a given page is specified by the element of the protection array that corresponds to this page. In this element, the access rights of the current domain are those in the positions of the R and the W fields corresponding to the bits of CDR that are asserted.

In the configuration of Figure 5d, the access control list for page P_0 is specified by the first element of the protection array. The current domain includes a single protection context, C_0. For P_0, the corresponding access right is READ.

4.4 Capability-based addressing

In the previous subsection, we have seen how protection contexts and the current domain register make it possible to implement forms of capability-based page protection at the level of the threads of the same process. Now we shall consider the capability protection paradigm from the point of view of segment protection between processes.

In a segment-oriented, capability-based protection environment, a capability is a pair {S,

AR}, where *S* is a segment identifier and *AR* is the specification of a set of access rights for *S*. In order to access a given segment, a process submits the protection system a valid capability for this segment, and the access rights specified by this capability must permit the access. The main advantage of capability-based addressing is simplicity in segment sharing. A process holding a capability for a given segment can allow another process to access this segment by a simple action of a capability copy.

Capabilities are sensitive objects that must be segregated from ordinary data, so that processes are prevented from forging a new capability, or modifying an existing capability and extending the access rights it contains. Several approaches have been devised to segregate capabilities in memory [21]. In the *tagged memory* approach, a 1-bit tag is associated with each memory cell. The tag of a given cell specifies whether this cell contains a capability, or an ordinary data item. The processor features a set of machine instructions, the *capability instructions*, aimed at capability treatment. When a capability instruction is executed on a memory cell that is tagged to contain an ordinary data item, an exception of violated protection is raised and execution of the instruction fails [22], [23]. In sharp contrast to memory bank standardization, the tagged memory approach relies on specialized main memory banks (e.g., the memory banks of a 64-bit system are 65 bits wide). Time and space overheads follow from the need to save and then restore the tags as part of the activities of page swapping between the primary memory and the secondary memory. In order to find all capabilities, a garbage collector must examine every single memory location [24].

In an alternative approach, capabilities are segregated in memory into *capability segments* (in contrast, *data segments* are reserved for storage of ordinary data items) [25], [26]. In this approach, processes are forced to adhere to a ubiquitous protection paradigm relying on complex structures consisting of capability segments reserved for storage of capabilities for the data segments. Typically, capability segments are organized hierarchically into tree structures, and the data segments are the leaf nodes.

A classical implementation of capability addressing uses a set of registers inside the processor, the *capability registers*, each aimed at storing a capability [25], [27]. In order to access a given segment, a process must have loaded a capability for this segment into a capability register. An address in memory consists of the name of a capability register and an offset in the segment referenced by the capability in this capability register. The access can be accomplished successfully only if the capability contains the access right permitting the access. When a process relinquishes the processor and a new process is assigned the processor, the contents of the capability registers are saved into the descriptor of the original process and these registers are filled with quantities taken from the descriptor of the new process. These actions are not necessary when a switch occurs between two threads of the same father processes. In fact, all the threads of the same process share a common set of access privileges, and these include the capabilities in the capability registers. A situation of this type is in sharp contrast with the principle of least privilege; a thread inherits the whole set of access permissions of its father process

instead of being restricted to only those access rights that are necessary for this thread to carry out its job.

Capabilities are effective in protecting the private objects of a given process from illegal accesses originating from the other processes; the protection system guarantees that a process can access a segment only if it holds a valid capability for this segment. On the other hand, capability-based protection is unable to contrast attacks to the private segments of a given process produced by a program extension taking the form of a trusted routine that is accepted within the process boundaries, e.g., a device driver or a multimedia codec. In a situation of this type, the added code will be in the position to take advantage of all the access privileges granted by the capabilities stored in the capability registers, thereby gaining access to the private segments of the hosting process. Owning to the inherent tree-structured nature of capability segment organizations, a single capability for a capability segment at a high position in the tree allows a harmful code to gain access to large portions of the private data of the victim process. This problem is especially important as extensions have been pointed out to be a major cause of operating system failures [28], [29], [30].

Our protection system supports valid solutions to the problems, delineated above. Handles can be mixed in memory with ordinary data items; the double encryption of handles in memory guarantees that every attempt to forge a handle or take advantage of a stolen handle is destined to fail. Furthermore, protection contexts and the current domain register allow us to restrict the set of access privileges of each thread of a multithreaded process to a subset of the access privileges of the father process, thereby implementing the principle of least privilege effectively. When a switch occurs between two threads of the same process, we modify the extent of the access privileges of the new thread with respect to those of the previous thread by simply changing the contents of the current domain register; whereas the contents of the handle registers do not change. We can limit the access privileges of a plugin to a subset of the privileges of the main application. In a protection paradigm using protection rings, we shall reserve an outer ring to plugins, for instance. Before starting up execution of a plugin, we shall change the contents of the current domain register to a configuration restricting the extent of the current domain to that outer ring; on termination of the execution of the plugin, the current domain register will be restored to the previous value.

4.5   Password capabilities

Password capabilities are an important improvement to the original capability concept [5], [6], [31], [32]. In a segment-oriented, password-capability architecture, the protection system associates one or more passwords with each segment, and each password corresponds to a subset of all access rights defined for this segment. A password capability is a pair {S, W}, where S is a segment identifier and W is a password. If a match exists between W and one of the passwords associated with S, a process that presents the password-capability to the protection system will be allowed to access segment S and perform the type of access permitted by the access

rights associated with the password.

Password capabilities can be freely mixed in memory with ordinary data; thus, they are an effective solution to the problem of capability segregation. A related problem is to prevent processes from forging valid password capabilities. If the virtual space is wide and scarcely populated, an arbitrary segment name has a low probability of corresponding to an existing segment. A further requirement is that passwords are large. It was argued that 64-bit passwords guarantee that the probability of guessing a valid password is vanishingly low even if the segment identifier is known [31].

The validity of a password capability extends system-wide, and is process independent. Suppose that process Q holds a password capability for a given segment S and a malevolent process Q' succeeds in accessing the memory area of Q and copying this password capability into its own address space. So doing, Q' acquires the access permissions for S originally held by Q. With respect to capability systems, in a password-capability environment this problem is complicated by the fact that password capabilities are not segregated in memory; this facilitate successful accomplishment of fraudulent actions of a password-capability copy. Storage of password capabilities in the stack and heap memory areas results in occasions for application of well-known techniques for data stealing [33], [34], for instance.

In our protection system, validity of a given handle is restricted to the process for which that handle was assembled in memory. As pointed out in Subsection 4.2, this is a consequence of the fact that each given handle is always stored in memory in the ciphertext form that follows from application of an encryption key, the process key, that is private to the process holding this handle. Consequently, any handle stealing action is ineffective. Furthermore, as seen in Subsection 4.1, we make no hypothesis on the size of the address space, and sparsity of segment allocation in the virtual space is not a requisite, as protection from handle forging is guaranteed by the double encryption and the handle validation field T^*.

4.6 Access right revocation

Suppose that process Q granted an access permission for a given segment S to process Q'. The protection system should include mechanisms allowing Q to review its own decision and retract the access permission from Q' as well as from any process that received the access permission from Q', recursively at any depth.

In capability environments, ease of access right distribution is a serious obstacle to access right revocation; the recipient of a capability is free to transmit this capability, and this makes it hard, if not impossible, to maintain track of all subsequent redistributions. Several proposals were made in the past, aimed at alleviating the revocation problem. These include a propagation graph for each capability recording all successive transfers of this capability throughout the system [35]; a reference monitor acting as a repository of the access permissions for a protected object [27]; short-lived capabilities, so that an access permission, once granted, can be held by the recipient only if the lifetime of the corresponding capability is periodically re-

newed [36]. Solutions of this type tend to subvert the salient characteristic of capability-based protection, i.e., simplicity of access right distribution. Space and time costs are added to the original access right management mechanism, in sharp contrast with the main reason for the introduction of the capability concept.

In our protection system, access right revocation is supported by three low-level mechanisms, i.e., process encryption keys, segment encryption keys, and, for the given segment, the protection fields of the page table of this segment. A modification of the encryption key k_Q of process Q has the effect of invalidating all the handles encrypted on behalf of this process, thereby revoking all the access permissions held by this process. Of course, this is a drastic decision that must be followed by a new distribution of handles, encrypted by using the new process key. It is important to note that the process key change does not affect the handles contained in plaintext in the handle registers. This is essential, as process execution should be allowed to continue, in particular, for the acquisition of the renewed handles. After the process key change, the process can even use the *hStore* protection primitive to save the handles contained in the handle register and encrypt them by using the new key. In spite of its simplicity, this revocation mechanism possesses a number of interesting properties [35]. Revocation results to be *temporal*, that is, its effects can be reversed by restoring the original process key using the same mechanism used for the revocation; *selective*, that is, we can exercise revocation from any subset of all processes, by changing the key of each of them; and *immediate*, that is, a process holding a given access privilege is prevented from taking advantage of this privilege starting from the time when the key of this process is changed.

A modification of the encryption key k_S of segment S has the effect of invalidating all the handles referencing S (in this case, too, the handles contained in plaintext in the handle registers are not affected by the revocation). This form of revocation results to be *transitive*, that is, if a process grants a handle for a given segment to other processes and these in turn transmit the handle to further recipients, the effect of a revocation propagates to all recipients at any transition depth; *temporal*, as its effects can be reversed by restoring the original segment key; *immediate*, as its effects start from the time of the segment key change. The two revocation strategies are orthogonal: a process key change is a segment-independent action that spreads on all the handles held by the given process, whereas a segment key change is a process-independent action that spreads on all the handles referencing the given segment, independently of the processes that hold these handles.

Finally, a modification of the contents of the R and W protection fields of one or more entries of the page table of a given segment has the effect of modifying the extent of the protection contexts defined for this segment. If we access the j-th entry of the page table of a given segment and clear the i-th bit of the R field of this entry, we eliminate the READ access right for the j-th page from the i-th protection context, for instance. This revocation mechanism results to be *partial*, that is, the extent of the revocation can be limited to any subset of all protection contexts; *transitive*, and indeed, if revocation involves a given handle, the effects of the revo-

cation propagate to all the copies of this handle; *temporal*, as its effects can be reversed by restoring the original contents of the protection fields; *immediate*, and in this case the effects of the revocation extends even to the handles stored in plaintext in the handle registers.

4.7 Single address space

As seen in the Introduction, in a single address space environment the meaning of an address extends to all processes and is independent of the process generating the address. This facilitates information sharing between processes. On the other hand, in the absence of mechanisms for private data protection, an erroneous or deliberately harmful process could access the private data items of a different process for both read and write, by simply using the addresses of these data items; the address translation circuitry is unable to prevent or ever reveal illegitimate accesses of this type. Thus, the addition of private data protection features is mandatory.

Several single address space systems have been designed and actually implemented in the past, and solutions to the protection problem have been devised at both the hardware and the software levels. In the rest of this subsection, we shall consider three important examples of single address space systems, namely Mungi, Arias and Opal. For each of them, the protection environment embedded in the system design will be discussed in some depth.

Mungi

In the Mungi single address space operating system [6], [32], a protected object takes the form of a collection of contiguous pages. Protection is enforced by taking advantage of password capabilities. A password capability consists of an object address and a password. When an object is created, an *owner password* is associated with that object, and a capability containing the owner password, called *owner capability*, is returned to the creator process. The owner password is a system-generated random number. The owner capability allows successful accomplishment of any operation on the object it references. The protection system defines four access modes, *read*, *write*, *execute* and *destroy*. A password derivation scheme based on one-way (hard to invert) functions makes it possible to derive less powerful capabilities (with restricted access rights) from the owner capability.

In the Mungi design, the password capability protection scheme was chosen instead of other sparse pointer schemes taking advantage of cryptography. This design choice was motivated by the need to avoid the costs connected with decryption in the validation of each object access. In contrast, in our system we avoid these costs by taking advantage of handle registers. Handle decryption and validation take place in the execution of the *hLoad* protection primitive, when a handle is loaded into a handle register. Afterwards, the handle register will be used for object access. No further actions of handle decryption is necessary unless the handle register is reused to contain a different handle; in the presence of an adequate set of handle registers, reuse will be arguably rare.

Arias

In the Arias Distributed Shared Memory System [37], protection is aimed at controlling accesses of system entities called *agents* to system controlled entities called *objects*. Each object is associated with a set of operations. For each agent, the protection mechanisms control the objects this agent can access and the operations it can execute on these objects. Protection takes advantages of capabilities. A capability identifies an object and grants a set of access rights on this object. Arias was actually implemented on top of an existing UNIX system. In this implementation, capability segregation was obtained by storing capabilities in kernel structures maintained by a kernel extension, thereby making them inaccessible to processes being executed with user privileges.

In Arias, when an agent creates an object, a capability referencing this object is returned to the agent including full access rights. The agent can then pass the capability to other agents, possibly with reduced access rights. A protection domain is itself an object; it consists of a collection of capabilities for other objects. At any given time, each agent is executed in a protection domain that states the objects that can be accessed by that agent at that time and the operations the agent may perform on these objects. An agent may move from its present domain to a different domain by performing a cross-domain invocation. To this aim, the domain of the agent must include a capability for an entry point of the new domain.

Opal

In the Opal operating system [5], [38], segments are the basic unit of storage allocation and protection. Each thread is executed in a protection domain that restricts the possible accesses of that thread to a subset of all segments. Access control is based on an implementation of the password capability concept. A thread wishing to access a given segment uses a password capability referencing that segment to explicitly attach the segment to its own protection domain; an action of this type allows all threads being executed in that domain to access the segment.

In Opal, protection is coarse-grained at the operating system level. A segment may well include several objects and even a heap for allocation of new objects. If a segment is attached to the domain of a given process, the process is free to access this segment and modify (and possibly corrupt) the objects it contains and even the heap. If two or more threads share a given segment, they can access all the objects allocated in that segment, and it is impossible to restrict sharing to a subset of these objects. Fine-grained protection relies on strongly-typed languages and the compiler. In contrast, our system supports a form of fine-grained protection at the level of the single pages that form a segment. This is made possible by protection contexts and the protection array. Being supported at the hardware level, the concept of a protection domain defined in terms of a subset of all protection contexts makes it possible to associate a protection domain with each thread while preserving efficiency in thread switching.

4.8   External protected memory

Trusted computing [39] is a recent design technology aimed at increasing data protection

and security in a wide class of computing platforms that includes, in particular, embedded systems designed to resist malicious attacks. At the hardware level, an essential component of a trusted execution environment is an authenticated non-volatile memory integrating storage protection and typically implemented as a cryptographic chip [39], [40]. An example of one such device is the Intel Authenticated Flash memory [41], which includes a standard flash memory and integrates security circuitry aimed at controlling memory accesses at the hardware level. This result is obtained by relying on the HMAC and the RSA signature protocols. A set of authenticated key management commands makes it possible to associate address ranges with RSA keys, so that all subsequent commands involving a range will require a valid signature to terminate successfully. This is the case, for instance, for the authenticated modify commands (write, replace, erase), and for the authenticated read that can be used to read enable or read disable a range (in the read enabled state, the contents of the range can be read by the host processor). If the host system is able to generate signatures in a secure fashion, Intel recommends utilization of a single protection range and the HMAC protocol, for better performance and simplified configuration.

The protection approach presented in this paper is well suited for systems that require a strong protection of all memory references that go off-chip. In a possible implementation of an external protected memory, a single segment, the *external segment*, will be assigned to this memory. This segment is divided into pages, and the page table is supported by the external memory hardware. The page table includes the protection array of the external segment, which specifies the read and write access rights for each page and each context (see Figure 1). A process is allowed to access the external memory only if it possesses a handle for the external segments. The port of this handle will specify the access privileges in terms of one or more protection contexts, as has been illustrated in Subsection 2.1. In a configuration of this type, let *ES* denote the identifier of the external segment. Let us suppose that a handle $H = \{ES, T\}$ referencing the external segment has been loaded into handle register *hReg*, and address *<hReg, d>* is generated by the processor. A modified port value obtained as the result of the bitwise AND of port *T* and the current domain register CDC (or, if the OWN bit of port *T* is set, an all-1's bit pattern) is sent to the external protected memory together with displacement *d*. In the external memory, the most significant bits of the displacement specify a page *P* in the external segment, and the least significant bits specify the offset *f* of the referenced information item in this page. Quantity *P* is used to identify an entry of the external page table. The bitwise AND of the modified port value and the R protection field of this entry (or, if the access is for write, the W protection field) is evaluated; if the result is 0 an exception of violated protection is raised, otherwise the memory access is finally accomplished.

4.9   HP's Precision Architecture

The Hewlett-Packard's Precision RISC Architecture (PA-RISC) [42], [43] incorporates a sophisticated set of hardware mechanisms for access protection into the storage unit, as part of

the address translation mechanisms. The processor architecture defines four privilege levels; at any given time, a process is assigned a privilege level, which can change as a consequence of the process activity (the privilege level is increased by ad-hoc instructions called *gateways* and is decreased by several branch instructions). Three access types are defined for pages, read, write, and execute. Each page is associated with an access right specification encoded in 7 bits that are divided into three sub-fields, a *type* sub-field that specifies the type of the accesses that can be successfully accomplished on this page, and two *privilege level* sub-fields PL_1 and PL_2 that specify bounds for the required privilege levels (a read access must be at least as privileged as PL_1, a write access must be at least as privileged as PL_2, and an execute access must be at least as privileged as PL_1 and no more privileged than PL_2). The *type* field can also specify promotion to a new privilege level, as occurs in the execution of a *gateway* instruction.

Furthermore, each page has a form of protection key called the *access identifier*. The processor includes four control registers each aimed at storing a protection key associated with the running process. A memory access to a given page can be successfully accomplished only if a match occurs between the contents of one of these control registers and the access identifier of that page. The union of the protection keys associated with the given process forms its current protection domain.

The protection system proposed in this paper defines two access rights for pages, READ and WRITE. Of course, the system may well be extended to support the EXECUTE access right; to this aim, the protection array will be enlarged to include a third protection field for the EXECUTE access right, of the same size as the R and W fields and a similar meaning. The concept of a privilege level is supported as follows. Let us refer to a system featuring *n* privilege levels numbered from 0 (the most privileged) to *n* - 1 (the least privileged). We shall reserve a protection context for each privilege level. Protection context C_i includes the access privileges associated with the *i*-th privilege level that are not part of the lower levels. If the running thread is assigned privilege level *i*, then in the current domain register CDR we set bit *i* corresponding to protection context C_i, as well as bits *i* + 1, *i* + 2,..., *n* - 1 corresponding to the protection contexts reserved for the lower levels. So doing, we cause C_i to inherit the access privileges of the lower levels. Promotion to a new privilege level corresponds to a change in the contents of CDR. When a thread switch takes place, the contents of CDR are replaced with a configuration corresponding to the privilege level of the new thread. Finally, at any given time, the memory regions that can be accessed by the running thread at that time are stated by the contents of the handle registers; a page can be actually accessed only if it is part of a segment referenced by one of these registers.

4.10 Considerations concerning performance

In our system, a virtual memory address has the form *<hReg, d>*, where *hReg* is the name of a handle register and *d* is a displacement in the segment referenced by this handle register; that is, with respect to an architecture featuring flat addresses, the address is extended to contain the

name of a handle register. In a possible implementation of the instruction set, the most significant bits of the displacement will be reserved to codify quantity *hReg*. In this implementation, the instruction size is not increased with respect to flat addresses; positive effects ensue on the complexity of the decode logic, for instance. Of course, in this approach fewer bits are available for the displacement; in a 64-bit architecture, the resulting size of the address space is still really large, and this is true even if a single address space is shared by all processes.

On the other hand, a close analogy exists between an instruction set in which each address includes the name of a handle register and the instruction set of a classical segmented memory architecture, in which each address includes the name of a segment register. Furthermore, in the address translation circuitry, the path from the handle registers to the primary memory is not more complex than the typical path from the segment registers to the primary memory in an architecture featuring a form of segmentation with paging. With respect to an architecture of this type, we may conclude that the addition of the handle registers and the *<hReg, d>* address format have no negative impact on system performance.

Cryptographic handles

Handles are double-encrypted using a symmetric key cipher. Double encryption enhances security only marginally [44]; in our design, we take advantage of a double encryption to support different forms of access right revocation, at both levels of all the handles held by a given process and all the handles referencing a given segment. This issue has been investigated in depth in Subsection 4.6.

In the transformation of plaintext handle $H = \{S, T\}$ into the ciphertext form $H = \{S^*, T, T^*\}$ the cipher should guarantee a careful mixing of the bits in the S^* and T field to form the validation field T^* (see Figure 2). This is essential to prevent tampering with the portion of the validation field that corresponds to port T and increasing the strength of the port. In this respect the XOR cipher is an example of a bad choice, which is subject to forms of known-plaintext attacks, for instance. A solution based on the DES cipher has a much higher computational cost, which can be supported by *ad-hoc* hardware [45].

A ciphertext handle is transformed into plaintext in the execution of the *hLoad* protection primitive, which loads a handle register with the result of the transformation. The handle register can be subsequently used for a potentially unlimited sequence of accesses to the segment referenced by that handle, unless the register is reused to contain a different handle. In the presence of an adequate number of handle registers, the necessity of handle register reuse is a relatively rare event. Thus, the cost in terms of execution times of an action of handle decrypt corresponds to more object accesses, and is comparatively low.

The inverse transformation of a plaintext handle into a ciphertext occurs in the execution of the *hStore*, *hReduce* and *hTranscode* protection primitives. *hStore* is used after a process key change to save the handles contained in the handle registers and encrypt them by using the new key. *hReduce* is aimed at limiting the access privileges contained in a given handle in view of

the transfer of this handle to a different thread of the same process. *hTranscode* is necessary to codify a handle by using the key of a different process in view of the transfer of the handle to a thread of this process. Of course, all these actions are relatively infrequent.

When execution of a given process is suspended as part of the actions involved in a process switch, the current state of the process is saved into the process descriptor, and this includes the contents of the handle registers. When execution of the process is subsequently resumed, the process state is restored with quantities taken from the process descriptor. No action of handle transformation from/to ciphertext is involved in the process switch.

We may conclude that the actions of handle encryption and decryption are comparatively rare, and the computational cost of the cipher is not a critical factor.

Cache implementation

As seen in Subsection 3.1, in our architecture address translation takes advantage of two caches, a segment table cache STK and a page table cache PTK. STK is aimed at storing recently-used segment table entries; this is similar to PTK for the page table entries. We hypothesized that both caches are managed at the hardware level. This is only a simplifying assumption that is neither required nor implied by the overall design of our MMU. In an alternative, program-controlled configuration, the caches are managed at the software level. In this approach, the instruction set of the processor includes special instructions that are directed to the caches; these instructions are aimed at instructing the caches to load the address translation information that will be used in the near future. The compiler will insert these special instructions at appropriate points of the object code. Let us refer to the segment table, for instance. The program logic contains explicit information concerning utilization of the contents of this table. When a handle register is loaded with a handle referencing a given segment, the segment table entry for this segment is likely to be used in the near future, and can be pre-loaded into the cache. An approach of this type may lead to important reductions in power consumption, with positive effects on the cooling and packaging requirements, for instance [46].

Storage requirements

Let us consider an implementation of the handle-based memory protection paradigm outlined in the previous sections within the framework of a 2^{64}-byte single address space. Let us hypothesize that a 64-bit address is partitioned into a 32-bit segment name and a 32-bit displacement. If the page size is 4 Kbytes, the displacement consists of a 20-bit page number and a 12-bit offset. As far as the port size is concerned, we aim at a form of fine-grained memory protection [47] that follows a *small protection domain* approach [48]. Small protection domains can be an important support in the debugging phase of program development. At run time, they facilitate fault detection, recovery and retry. A port size of 8 bits makes it possible to define 7 different protection contexts (one bit being reserved for the OWN context). In this hypothesis, an effective implementation of the small domain approach is possible, e.g., a protection structure featuring up to 127 protection domains or 7 privilege levels.[7]

In an environment of this type, the size of a plaintext handle $H = \{S, T\}$ is 5 bytes (4 bytes for segment name S and 1 byte for port T), and the size of a ciphertext handle $H^* = \{S^*, T, T^*\}$ is 10 bytes (4 bytes for S^*, 1 byte for T and 5 bytes for the validation field T^*). The total storage requirement of the handles referencing segment S depends on the *sharing factor sf*, i.e., the number of processes that share access to this segment, and is given by $10 \cdot sf$ (one ciphertext handle for each process). Let s denote the size (in pages) of segment S. In the page table of S, the storage requirement of the protection information is connected with the protection fields R and W, and is equal to $2 \cdot s$ bytes. If the size of the segment encryption key is 8 bytes, the total storage requirement ρ for segment protection is given by $\rho = 10 \cdot sf + 2 \cdot s + 8$. Let us define the memory overhead τ as the ratio between the size of the protection information for a segment and the segment size; for 4-Kbyte pages we have $\tau = \rho / (4096 \cdot s)$. Let us now refer to one-page segments ($s = 1$, a worst-case analysis). For a private segment ($sf = 1$) the memory requirement ρ is 20 bytes, and the memory overhead τ is 0.5 per cent. For a segment shared by two processes ($sf = 2$), ρ is 30 bytes and τ increases to 0.7 per cent. At a higher degree of sharing, $sf = 10$, τ is 2.7 per cent. Of course, τ decreases for larger segments, e.g., for a highly-shared, 10-page segment ($sf = 10$ and $s = 10$) we have $\tau = 0.3$ per cent.

We may conclude that the total storage requirements of the information for memory protection are a fairly negligible fraction of the overall requirements for segment storage.

5. CONCLUDING REMARKS

We have considered a single-address-space system that implements a form of segmentation with paging within the framework of the multithreaded model of application programming. We have presented a paradigm for memory protection that is based on the application of techniques of symmetric-key cryptography. A salient feature of our paradigm is that the protection system interface, as defined by the protection primitives, hides cryptography from application programs, in the implementation level. In fact, cryptography is functional to handle protection rather than to handle utilization for memory reference; application programs do not need to be aware of the duality of plaintext and ciphertext handles, for instance. We have obtained the following results:

- A handle held by a given process in memory is encrypted by using the key of this process. An attempt made by a different process to decrypt the handle into a handle register for subsequent memory access will use a different key, thereby resulting in a meaningless plaintext handle. In this way, process keys prevent handle stealing.

- Segment sparsity in the virtual space and a low-density of virtual space allocation are not requisites. Even in the presence of a high number of segments, when the probability of guessing a valid segment identifier is not evanescent, it is realistically impossible to construct a correct validation field for a ciphertext handle referencing a given segment, as this

---

7. In contrast, for example, the HP's PA-RISC is limited to 4 privilege levels [42].

action requires knowledge of the segment key. In this way, segment keys prevent handle forging.

- Access privilege transfer between processes requires an action of handle transcode that implies the transformation of the handle to plaintext using the key of the granting process and the subsequent transformation of the same handle back to ciphertext using the key of the recipient process. The cost of the transcode in terms of processing time is mitigated by the fact that interactions between different processes are relatively infrequent. Conversely, all the threads of the same given process use the encryption key of this process, and access right transmission between these threads can be accomplished by a simple action of handle copy at low processing time costs.

- The mechanism of protection contexts and the current protection domain make it possible to grant each thread a fraction of the access rights of its father process. The only processing time cost connected with a domain switch between the threads of the same process is that of a change of the contents of the current domain register. We have seen how protection contexts make it possible to implement a number of well-known protection paradigms, including hierarchical domains, protection rings, capability lists and access control lists.

- The two encryption keys, the process key and the segment key, allow us to implement techniques of access right revocation at little effort. By modifying the encryption key of the given segment we revoke all the handles referencing this segment, independently of the processes that hold these handles. Orthogonally, by modifying the key of a given process we revoke all the handles held by this process, independently of the segments they reference. Furthermore, by modifying the contents of the protection array of a given segment we can limit access permissions at the level of the single pages that form the segment, and by modifying the contents of the current domain register we can exercise access privilege revocation at the level of the single thread of a multithreaded application.

The idea of protecting pointers by means of cryptographic techniques is not new [49], [50]. In the protection system presented in this paper, we take advantage of protection contexts defined at the page level, and the separate encryption of the name of a segment and the specification of the access privileges for this segment, to obtain a level of flexibility in access privilege management well suited to memory protection in multithreaded applications and single address space environments.

REFERENCES

[1]   Cekleov, M. and Dubois, M. (1997) Virtual-address caches. Part 1: problems and solutions in uniprocessors. *IEEE Micro*, **17**, 5, 64–71.

[2]   Qiu, X. and Dubois, M. (2008) The Synonym Lookaside Buffer: a solution to the synonym problem in virtual caches. *IEEE Transactions on Computers*, **57**, 12, 1585–1599.

[3]   Zhou, X. and Petrov, P. (2006) Low-Power Cache Organization Through Selective Tag Translation for Embedded Processors With Virtual Memory Support. *Proceedings of the 16th ACM Great Lakes Symposium on VLSI*, Philadelphia, PA, USA, April, pp. 398–403. ACM, New York, NY, USA.

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

[5] Chase, J. S., Levy, H. M., Feeley, M. J. and Lazowska, E. D. (1994) Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, **12**, 4, 271–307.

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

[7] Miller, D. S., White, D. B., Skousen, A. C. and Tcherepov, R. (2006) Lower Level Architecture of the Sombrero Single Address Space Distributed Operating System. *Proceedings of the 8th IASTED International Conference on Parallel and Distributed Computing and Systems*, Dallas, Texas, USA, November. IASTED, Calgary, Alberta, Canada.

[8] Kleiman, S., Shah, D. and Smaalders, B. (1996) *Programming With Threads*. Prentice Hall, Upper Saddle River, NJ, USA.

[9] Lewis, B. and Berg, D. (1996) *Threads Primer: A Guide to Multithreaded Programming*. Prentice Hall, Upper Saddle River, NJ, USA.

[10] Chiueh, T., Venkitachalam, G. and Pradhan, P. (1999) Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions. *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Charleston, South Carolina, USA, December, pp. 140–153. ACM, New York, NY, USA.

[11] Lohmann, D., Streicher, J., Hofer, W., Spinczyk, O. and Schröder-Preikschat, W. (2007) Configurable Memory Protection by Aspects. *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, Stevenson, Washington, USA, October. ACM, New York, NY, USA.

[12] Ferguson, N. and Schneier, B. (2003) *Practical Cryptography*. John Wiley & Sons, New York, NY, USA.

[13] Trappe, W. and Washington, L. C. (2005) *Introduction to Cryptography with Coding Theory*, Second Edition. Prentice Hall, Upper Saddle River, NJ, USA.

[14] Lopriore, L. (2002) Access control mechanisms in a distributed, persistent memory system. *IEEE Transactions on Parallel and Distributed Systems*, **13**, 10, 1066–1083.

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

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

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

[18] Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R. (2000) Introducing the IA-64 architecture. *IEEE Micro*, **20**, 5, 12–23.

[19] Swaminathan, S., Patel, S. B., Dieffenderfer, J. and Silberman, J. (2005) Reducing Power Consumption During TLB Lookups in a PowerPC™ Embedded Processor. *Proceedings of the Sixth International Symposium on Quality of Electronic Design*, San Jose, CA, USA, March, pp. 54–58. IEEE Computer Society, Washington, DC, USA.

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

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

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

[23] Neumann, P. G. and Feiertag, R. J. (2003) PSOS Revisited. *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December, pp. 208–216. IEEE Computer Society Press, Los Alamitos, CA, USA.

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

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

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

[27] Shapiro, J. S., Smith, J. M. and Farber, D. J. (1999) EROS: A Fast Capability System. *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, USA, December, Operating Systems Review **34**, 5, pp. 170–185. ACM, New York, NY, USA.

[28] Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D. (2001) An Empirical Study of Operating Systems Errors. *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, Banff, Alberta, Canada, October, pp. 73–88. ACM, New York, NY, USA.

[29] Swift, M. M., Bershad, B. N. and Levy, H. M. (2005) Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, **23**, 1, 77–110.

[30] Tanenbaum, A. S., Herder, J. N. and Bos, H. (2006) Can we make operating systems reliable and secure?. *Computer*, **39**, 5, 44–51.

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

[32] Vochteloo, J., Russell, S. and Heiser, G. (1993) Capability-Based Protection in the Mungi Operating System. *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, Asheville, NC, USA, December, pp. 108–115. IEEE Computer Society, Washington, DC, USA.

[33] Tuck, N., Calder, B. and Varghese, G. (2004) Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow. *Proceedings of the 37th International Symposium on Microarchitecture*, Portland, Oregon, USA, December, pp. 209–220. IEEE Computer Society, Washington, DC, USA.

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

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

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

[37] Hagimont, D., Mossière, J., Rousset de Pina, X. and Saunier, F. (1996) Hidden Software Capabilities. *Proceedings of the 16th International Conference on Distributed Computing Systems*, Hong Kong, China, May, pp. 282–289.

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

[39] Schellekens, D., Tuyls, P. and Preneel, B. (2008) Embedded Trusted Computing with Authenticated Non-Volatile Memory. *Trusted Computing – Challenges and Applications*, Lecture Notes in Computer Science 4968, pp. 60–74. Springer, Berlin.

[40] Ekberg, J.-E. and Asokan, N. (2010) External Authenticated Non-Volatile Memory with Lifecycle Management for State Protection in Trusted Computing. *Trusted Systems*, Lecture Notes in Computer Science 6163, pp. 16–38. Springer, Berlin.

[41] Alves, T. and Rudeli, J. (2007) ARM Security Solutions and Intel Authenticated Flash – How to Integrate Intel Authenticated Flash with ARM TrustZone for Maximum System Protection. *Design & Reuse*, October. http://www.design-reuse.com/articles/16975

[42] Wilkes, J. and Sears, B. (1992) A comparison of protection lookaside buffers and the PA-RISC protection architecture. HP Laboratories Technical Report HPL–92–55, Palo Alto, CA, USA.

[43] Lee, R. B. (1989) Precision architecture. *Computer*, **22**, 1, 78–91.

[44] Gaži, P and Maurer, U. (2009) Cascade Encryption Revisited. *Proceeding of the 15th International Conference on the Theory and Application of Cryptology and Information Security*, Tokyo, Japan, December, Lecture Notes in Computer Science 5912, pp. 37–51. Springer, Berlin.

[45] Eisenbarth, T. and Kumar, S. (2007) A survey of lightweight-cryptography implementations. *IEEE Design & Test of Computers*, **24**, 6, 522–533.

[46] Kadayif, I., Nath, P., Kandemir, M. and Sivasubramaniam, A. (2007) Reducing data TLB power via compiler-directed address generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **26**, 2, 312–324.

[47] Shen, J., Venkataramani, G. and Prvulovic, M. (2006) Tradeoffs in Fine-Grained Heap Memory Protection. *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, San Jose, California, USA, October, pp. 52–57. ACM, New York, NY, USA.

[48] Gehringer, E. F. (1982) *Capability Architectures and Small Objects*. UMI Research Press, Ann Arbor, MI, USA.

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

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