Computers & Security, 10 (1991) 639-652

Refereed Article

Algorithm-Sequenced Access Control

Josep Domingo-Ferrer*

Departament d'Informàtica, Universitat Autònoma de Barcelona, 08193 Bellaterra (Barcelona), Catalonia, Spain

A new concept called algorithm-sequenced access control is modelled, whose central idea is that the access rights for a subject change with time, according to an access algorithm. A computing system that supports this type of access control is then described. Security for the system security modules themselves is provided by a combination of cryptographic and hardware tools. The result is a modular access control mechanism, which is particularly suitable for controlling parallel environments.

Keywords: Algorithm-sequenced access control, Computer security, Access algorithms, Software integrity, Cryptography, Signatures, Secure parallel processing.

1. Introduction

Access control is usually modelled in terms of a set S of subjects (processes running on behalf of users or programs), a set O of objects (files or devices) and a set R of rights of subjects to objects. The most widely known approaches to access control, such as the access matrix and others (see refs. [2] and [6] for a review), do not worry about the sequence in which accesses are made. This type of control is possible if the use of capabilities is restricted in an algorithmic fashion. An algorithm-sequenced access control model is stated in this paper, and then a system for supporting it is presented that consists of the following elements:

(1) An *operating system*—a collection of supervisor processes.

(2) A *security administrator*—a human who defines the security policy.

(3) A *protection system*—a reliable ROM software for security management tasks.

(4) A *reference monitor*—a composite hardware module which checks that accesses made by subjects do not violate the security policy.

(5) *Subjects* and *objects.*

In section 2 access algorithms are defined and classified, and examples of their possible applications are given; furthermore, capabilities are shown to be a special type of access algorithm. In section 3, the operation of a reference monitor which controls references as well as their synchronization is specified; because of its modularity, this hardware reference monitor is suited for parallel environments. In section 4, the access algorithm management commands which constitute the protection system are introduced. In section 5, cryptographic structures are described which ensure that the reference monitor is non-bypassable. In section 6, a coding technique based on ref. 4 for providing secure interrupts and program and access algorithm integrity is sketched. Section 7 contains some concluding remarks.

*Present address: Centre de Supercomputació de Catalunya, Diagonal 645, 0828 Barcelona, Spain.

2. Modelling Algorithm-Sequenced Access Control

Let us first introduce some preliminary concepts. *Subjects, i.e.* the active elements which gain access to information, are identified with processes. When a user logs in, he authenticates to the protection system using an authentication protocol (password typing); then the protection system creates a log-in process for the user. Programs run by the log-in process give birth to child processes. On the other hand, *objects* can be any passive repository of information, including files, directories, global sections, mailboxes, queues, etc.; in order to simplify the discussion, objects can be thought of as files. In general, *objects are stored in an enciphered form* in order to prevent access control bypassing.

Notation

The "don't care" or wild card value is represented by an asterisk "*". Identifiers ending in *const* denote constants, identifiers ending in *var* denote variables, whereas identifiers ending in *exp* denote expressions which can be variable or constant. As an exception *sexp* is a two-valued constant expression denoting either a fixed subject *s* or a universal subject.

Definition 1

An access token t is a 4-tuple t = (sexp,rexp,oexp,switch), where sexp stands for a subject, oexp stands for an object, rexp stands for one access right of sexp to oexp. sexp may be either a fixed subject s or the universal subject denoted by any. rexp may be a constant or variable access right among the following: r -read-, w -write-, e -execute- and d -delete-. oexp may represent a constant or variable object. switch is a binary value: if switch = on then the access right rexp to oexp is being enabled for sexp; if switch = off then the access right rexp to oexp is being disabled for sexp. If rexp and oexp are both constant, the token is called a constant access token; if, on the other hand, at least one of them is a variable, the token is a variable access token.

Tokens with a universal subject, or universal tokens, can be applied to any subject and, as it will be shown later, they can be used by their owner to grant a right to another subject.

Definition 2

In our context, an access language instruction i_k can be an instruction of the following types: declaration, input (READ), arithmetic, conditional or unconditional branch (backward or forward), subroutine branch or END. Conditions can be placed on a count, on whether an access has been actually made or not, on object type, etc.

Definition 3

An access algorithm A is an ordered set $(a_1, \ldots, a_n)$, where a_k can be either an access language instruction or an access token.

Access algorithms can be classified in four non-disjoint categories, namely:

- Non-iterative access algorithms with Constant access tokens (*NC* algorithms for short).
- Iterative access algorithms with Constant access tokens (*IC* algorithms for short).
- Non-iterative access algorithms with Variable access tokens (*NV* algorithms for short).
- Iterative access algorithms with Variable access tokens (*IV* algorithms for short).

The categories *NC, IC, NV* and *IV* can be viewed as a lattice structure (Fig. 1), as we will see in the

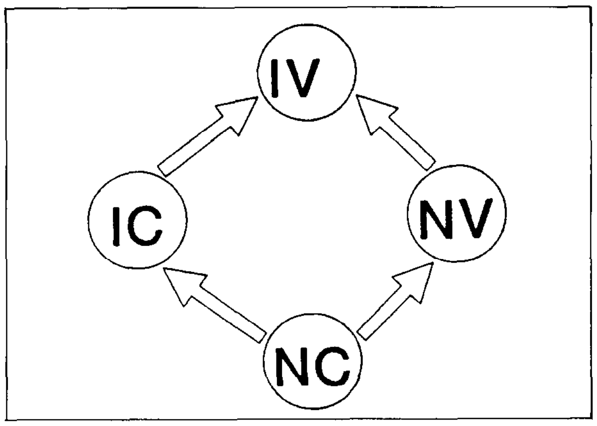

$$\begin{tikzcd} & IV & \\ IC \arrow[ur] & & NV \arrow[ul] \\ & NC \arrow[ul] \arrow[ur] & \end{tikzcd}$$

Fig. 1. Lattice structure of categories for access algorithms.

following subsections. *NC* is the lowest class, whereas *IV* is the highest. Now each class will be described in further detail and the kind of access algorithms used will become clear.

2.1 Non-Iterative Access Algorithms With Constant Access Tokens (*NC* Algorithms)

An *NC* algorithm A has the form $A=(a_1, \ldots, a_n)$, where a_k can be either a constant access token, or any access language instruction but a backward branch (to an instruction "above" the branch instruction). The main idea is that access tokens must be used sequentially and only once. By associating an *NC* algorithm A_i to a subject s_i, it is possible to strictly enforce an access sequence for s_i. Since A_i is non-iterative, once the access sequence is finished, no more accesses at all will be allowed to s_i (the process is "dead" or "frozen" from an input/output point of view); this also means that if a non-existent token is requested (when attempting to perform an illegal access), then the process is frozen, since its access algorithm is used up while vainly searching it for the required token. Of course, s_i may skip some of the $(*, *, *, on)$ enabling tokens specified in A_i. It is possible to place forward branch conditions depending on an access having been made or not, in order for a given access sequence to be conditionally allowed.

Example 1

As an example of application, imagine the following situation. In a psychology department, a computerized testing system is to be used to measure the skill of students in a particular field. Then it is reasonable to create for each testing student a process s_i running the test program *tp* under a special account. This program displays a questionnaire file *qf* and records the student's responses in a response file *rf*; then the same program allows *the students who answered the test* first to determine their absolute score by displaying a key response file *krf* and then to compute their relative score by displaying a statistics file *sf*. Instead of trusting students and the test program, each student is given a one-time password for a special-purpose account which has the following access algorithm: first, execute access to *tp*; then read access to *qf*, then write access to *rf* (for answer recording); after this, if s_i answered the test, read access to *krf* (absolute score) and finally read access to *sf* (relative score). Thus this *NC* access algorithm for s_i could be written in low-level pseudocode as

```
$A_i$:
        $(s_i, e, tp, on)$
        $(s_i, r, qf, on)$
        $(s_i, w, rf, on)$
        $(s_i, w, rf, off)$
        $(s_i, r, qf, off)$
        IF not rf_written GOTO finish
        $(s_i, r, krf, on)$
        $(s_i, r, krf, off)$
        $(s_i, r, sf, on)$
        $(s_i, r, sf, off)$
finish:
        $(s_i, e, tp, off)$
END
```

So authorized accesses as well as their synchronization are enforced.□

2.2 Iterative Access Algorithms with Constant Access Tokens (*IC* Algorithms)

An *IC* algorithm A has the form $A=(a_1, \ldots, a_n)$, where a_k can be either a constant access token or any access language instruction. It is clear that *IC* algorithms are actually a generalization of *NC* algorithms ($NC \subset IC$, see Fig. 1). The key feature of *IC* algorithms is backward branches, which make it possible to repeat a given access pattern either indefinitely or depending on a condition. Also, an *IC* algorithm need not finish when an illegal access is attempted (a timeout exiting condition can be implemented to prevent infinite loops when the required token is not in A).

Example 2

Resuming Example 1, suppose that, for training purposes, the test program allows up to three test repetitions (with some variation between them). Then the access algorithm for the process s_i

running *tp* would be similar to that of Example 1, but inside a loop, in order to perform the three iterations (a count is first initialized to 3, and then decremented for each iteration).□

Theorem 1
Capability systems are a special case of IC access algorithms.

Proof
Let CL_i denote the capability list of subject s_i. Suppose $CL_i = \{(rconst_{i1}, o_1), \ldots, (rconst_{im}, o_m)\}$, where $rconst_{ij}$ denote the access rights of s_i for o_j. Assume that CL_i has already been normalized, so that each $rconst_{ij}$ in it represents a single access right. Then CL_i can be replaced by an *IC* access algorithm:

```
A_i:
again:
        (any, rconst_1, o_1, on)
        ⋮
        (any, rconst_m, o_m, on)
        GOTO again
```

where $rconst_j := rconst_{ij}$ and *any* is the universal subject. It is easy to see that each iteration of A_i allows s_i to perform any of his authorized accesses. Remember that enabling tokens are not mandatory, so that they can be skipped selectively. Moreover the rights are permanent (there are no disabling tokens); still, it is necessary to include a loop in order to avoid freezing the subject upon an illegal access attempt, as would happen in an *NC* algorithm.□

2.3 Non-Iterative Access Algorithms with Variable Access Tokens (*NV* Algorithms)

An *NV* algorithm A has the form $A = (a_1, \ldots, a_n)$, where a_k can be either a variable access token (with variable right $rvar_{ij}$ and/or object $ovar_j$), a constant access token, or any access language instruction but a backward branch. It holds that $NC \subset NV$ (see Fig. 1). As a rule, a variable access token is preceded by as many input instructions as variables it contains (one or two), in order to instantiate them. An access token $a_k = (sexp_i, rvar_{ij}, o_j, on)$, where $rvar_{ij}$ is a variable right, allows $sexp_i$ to perform any operation on o_j (this is equivalent to temporarily putting $sexp_i$ in the owner category of some operating systems [1], as far as o_j is concerned). An access token $a_k = (sexp_i, rconst_{ij}, ovar_j, on)$, where $ovar_j$ is a variable object, allows $sexp_i$ to perform the operation specified by $rconst_{ij}$ on any object (for example, if $rconst_{ij} = r$ — read right—, then $sexp_i$ can read any object in the system). Finally, an access token $a_k = (sexp_i, rvar_{ij}, ovar_j, on)$, where $rvar_{ij}$ is a variable right and $ovar_j$ is a variable object, allows $sexp_i$ to perform any operation on any object in the system (it means temporarily putting $sexp_i$ in the supervisor category). Remark that *NV* algorithms place a restriction on the number of tokens of any kind available to a particular subject, since the algorithm is non-iterative and no token can be used twice.

Example 3
Suppose now that there are several possible questionnaires for the test of Example 1, one of which must be blindly selected by the testing student. Rather than including in the access algorithm an access token for each questionnaire file *qfh*, a token with a variable object *qfvar* and a fixed read access is used; of course, this requires checking that the value assigned to *qfvar* is actually a questionnaire file (use the object type information in the object header; see section 5). So, the access algorithm for the process s_i running the testing program will be

```
A_i:
        (s_i, e, tp, on)
        READ qfvar
        IF not qfvar_is_a_questionnaire GOTO finish
        (s_i, r, qfvar, on)
        (s_i, w, rf, on)
        (s_i, w, rf, off)
        (s_i, r, qfvar, off)
        IF not rf_written GOTO finish
        (s_i, r, krf, on)
        (s_i, r, krf, off)
        (s_i, r, sf, on)
        (s_i, r, sf, off)
finish:
        (s_i, e, tp, off)
        END
```
□

Remark that the principle of least privilege now has time consequences: as soon as an access right is no longer needed, it must be disabled.

2.4 Iterative Access Algorithms with Variable Access Tokens (*IV* Algorithms)

An *IV* algorithm *A* has the form $A=(a_1, \ldots, a_n)$, where a_k can be either a variable access token, a constant access token or any access language instruction. In this case, the tokens in *A* can be used several times.

Example 4

Resuming Example 3, imagine that each student is allowed to choose a questionnaire and to test as many times as he/she wishes. Then the access algorithm for s_i would be similar to that of Example 3, but inside an unlimited loop.□

Let us give the whole access algorithm for true owner and supervisor process categories: this algorithm-sequenced version must include an iteration, in order for it to be able to be resumed after attempting an illegal access. It can be easily seen that if s_i owns an object o_j then

```
$OWNER_{ij}$:
$again$:
        READ $rvar_{ij}$
        $(s_i, rvar_{ij}, o_j, on)$
        GOTO $again$
```

implements the o_j's owner access rights. Also, granting supervisor category to subject s_i can be expressed in the following algorithmic way:

```
$SUPERVISOR_i$:
$again$:
        READ $ovar_j$
        READ $rvar_{ij}$
        $(s_i, rvar_{ij}, ovar_j, on)$
        GOTO $again$
```

3. Reference Monitor Operation

The reference monitor of our system is constituted by three cooperating modules (Fig. 2): a general-purpose processor *P*, which runs in an encoded form the operating system and standard software associated with the current process, a special-purpose security processor *SP* which runs an encoded copy of the access algorithm of the current process, and an encryption module *EM* which handles cryptographic operations (object ciphering and deciphering). A coding that ensures run-time integrity for access algorithms and *P* programs, as well as the structure of *P* and *SP*, are discussed in section 6. The following assumptions are made:

A. *P*, *SP* and *EM* are trusted separate modules (hardware modules for clarity). Separation of *SP* allows a generalization of the scheme to a parallel scenario, in which there are several *EM*s and *P*s, and a single security processor *SP* (a process may use several processors *P* whose accesses are controlled by a single access algorithm running in *SP*). Separation of *EM* and *P* allows the use of several *EM* for each *P* (interleaved encryption–decryption for I/O-bounded environments) or several *P* for each *EM* (shared encryption unit for CPU-bounded environments).

B. The software run by *P* may be untrusted; as for access algorithms run by *SP*, they are trusted

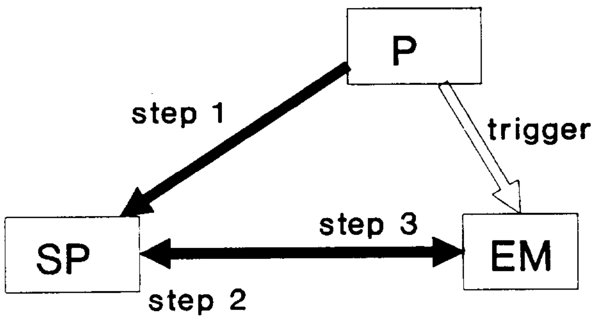


Fig. 2. Reference monitor operation.

because of their restricted management (see section 4).

C. Each processor has a public/secret key pair: $(KP(w), KP(r))$ for P, $(KSP(w), KSP(r))$ for SP, and $(KEM(w), KEM(r))$ for EM; secret keys are unknown outside their processor. Enciphering under the public key is undone by deciphering under the corresponding secret key, and conversely [3].

Now, the reference monitor operates according to the following protocol.

Protocol 1 (Algorithm-Sequenced Access Control Processing)
Assume that P is running software for s_i and that SP has been loaded with s_i's access algorithm A_i. Assume also proper identification, i.e. that A_i really corresponds to the user or program behind s_i (see section 4: **create_subject** *command). Then:*

(1) When P processes an access of s_i to an object o_j, it sends a signed request to SP (using its secret key KP (r)) in order to get an access permit; this signed request specifies $(s_i, rconst_{ij}, o_j)$ where $rconst_{ij}$ is the requested access right for object o_j. The request signature includes an access request number Preq # to avoid substitution or replay. At the end of this step, processor P triggers the encryption module EM.

(2) Having been triggered, EM starts a key exchange in order to obtain a key from SP. It sends to SP a signed message (using KEM(r)) containing a key request number EMreq#.

(3) Now SP acts in the following way:

(a) If P's signature or EM message were not as due (because of some natural or intentional disturbance), then SP does not deliver any key to EM and exits the protocol. Thanks to the identifier s_i included in P's request at step 1, SP can check each time that the access algorithm being run corresponds to the subject currently holding P (consistency checks are useful in case of interrupts).

(b) If a matching token $(sexp_i, rexp_{ij}, oexp_j, on)$, where $sexp_i \in \{any, s_i\}$, $rexp_{ij} = rconst_{ij}$ and $oexp_j = o_j$ ($rexp_{ij}$ and $oexp_j$ are constants or instantiated variables), is active (i.e. has not yet been cancelled by a corresponding $(sexp_i, rexp_{ij}, oexp_j, off)$), then SP goes to 3e.

(c) SP tries to locate a matching token (in the sense of step 3b) by following s_i's access algorithm instruction flow; during this search, skipped (, *, *, on) enabling tokens are not used, whereas (*, *, *, off) disabling tokens are processed by SP, so that the rights contained in them are cancelled if ever enabled. If some input information is needed to evaluate a condition or to satisfy a READ access language instruction, then an exchange of signed messages with P is started by SP to get the required information (SP uses KSP(r) to issue a signed request and P uses KP(r) to answer).*

(d) If one of the following two conditions occurs then SP does not deliver any key and exits the protocol:

- *s_i's access algorithm finishes without yielding a matching token. No more accesses will be allowed to s_i ("frozen" process).*

- *s_i's access algorithm keeps looping longer than a predefined timeout. After the timeout, the access algorithm is resumed at the access language instruction that was about to be processed after the illegal access attempt. There may be a penalty of lost rights due to disabling tokens processed during the search.*

(e) SP computes an object enciphering key KO, called object key, and sends it to EM, so completing the key exchange initiated by the encryption module. SP uses its private key KSP(r) as well as KEM(w) to sign and encrypt the object key, which is recovered by EM using KSP(w) and KEM(r). If another key is sent along with KO, both of them will be used consecutively by EM for all encryptions resulting from the current access of P to o_j with access right $rconst_{ij}$ (this is true when $rconst_{ij} = e$; see constructive proof of Theorem 2 in section 5).

Note that the encryption module is a trusted module that requires a new object key from SP

each time it is triggered. There is only one case in which *EM* can reuse a previously received key: when dealing with a group of *consecutive* equal accesses (for example, reading a character string from a file on a one-by-one basis), only the first access requires running the basic protocol. For a further discussion on object keys, see section 5.

4. Access Algorithm Management: The Protection System

The *protection system* is a trusted program (held in a read-only memory, for example) that complements the operating system for the security tasks; as with the modules in the reference monitor, there is a public/secret key pair (*Kps*(*w*), *Kps*(*r*)) associated with this program. The secret key *Kps*(*r*) is unknown outside the ROM and makes the protection system non-bypassable, so that no one else can seal executable images and access algorithms in a valid fashion (see section 6). Management involves creation of access algorithms, subjects and objects, as well as access algorithm purging and token granting and revoking; it is conducted by the protection system—upon security administrator, operating system or subject request—using the following six commands:

(1) **create_algorithm** *(entity, A). Caller*: security administrator. This command can only be executed by the security administrator and causes the protection system to associate a new access algorithm *A* with *entity*, who may be a user or a program; in the latter case, the program must be encoded according to section 6, and the coding for its first instruction (sort of hash value for the whole program) is stored in an *A* declaration instruction. When using the **create_subject** command, subjects s_i running on behalf of *entity* will be assigned a copy A_i of algorithm *A*. The subject fields of the tokens in *A* have either an *any* value or an *entity* value.

(2) **create_subject**(s_i, A_j, s_j). *Caller*: subject s_i. Operation depends on the entity behind subject s_j to be created:

User A log-in process s_j for user *u* is created by an operating system parent process s_i, which calls *create_subject* to authenticate *u* (the authentication protocol will not be dealt with here). If authentication is successful, a copy A_j of *u*'s access algorithm *A* is returned for s_j; otherwise a null algorithm is returned.

Program with an own access algorithm Operation as in the previous case, but user authentication is replaced by a check of the first program instruction coding against the value stored with the program's access algorithm. s_j identifies now the child process running the program.

Program without access algorithm No new child process is created for the program, which uses the tokens in the s_j parent process's access algorithm. A failure status is returned by the command.

When s_j is assigned processor *P*, A_j is loaded on the security processor *SP*. All subject fields not equal to *any* (*i.e.* equal to *entity*) in the access tokens are initialized to s_j.

(3) **create_object** (s_i, SA_{ij}, o_j). *Caller*: subject s_i. Subject s_i asks the protection system to create an object o_j, and proposes that the access subalgorithm SA_{ij} for object o_j (*i.e.* a set of tokens (*, *, o_j, *) and access language instructions) be included in s_i's access algorithm A_i. First, the protection system examines A_i to determine whether s_i has write access to the directory that is to contain o_j. If it is not so, the object cannot be created; otherwise, the protection system examines SA_{ij} and filters illegal tokens (not related to o_j, for instance). If SA_{ij} is a void subalgorithm, the $OWNER_{ij}$ access subalgorithm is included in A_i (*i.e.* an (s_i, $rvar_{ij}$, o_j, *on*) token is placed into an infinite iteration).

If the object is an executable image, then it is encoded and signed with Kps(r) as specified in section 6. All changes made on s_i's copy A_i are reflected on the corresponding original A. Then object keys are chosen and an object header is appended to the new—void—object (section 5).

(4) **purge** (s_i, A_i). *Caller*: subject s_i. s_i asks the protection system to delete from his access algorithm A_i all tokens ($*$, $*$, o_j, $*$) referred to objects o_j that no longer exist (which have been deleted by an authorized subject); useless access language instructions are deleted as well, by means of a restructuring procedure. These tokens are harmless, since they are not going to be requested any more, but they cause the access algorithm to run slower. So it is advisable to perform **purge** commands periodically; all changes made by s_i on A_i are reflected on the original algorithm A.

(5) **grant** (s_i, t, s_k). *Caller*: subject s_i. s_i asks the protection system to copy a universal enabling token $t=(any, rexp_j, o_j, on)$ from his algorithm A_i to an access subroutine that starts at a location $loc(i,k)$ (a known function of i,k); this subroutine *contains only tokens* (no access language instructions) and the memory region starting at $loc(i,k)$ acts as a one-way channel between s_i and s_k. If A_i also contains a disabling token $t'=(any, rexp_j, o_j, off)$ corresponding to t, then t' is copied after t at $loc(i,k)$. Then the protection system searches s_k's access algorithm A_k for a subroutine branch instruction to location $loc(i,k)$ (only $loc(*,k)$ subroutine branch addresses are possible in A_k). If one is found, the subroutine branch instruction in A_k is bound to the modified access subroutine stored at $loc(i,k)$; otherwise the **grant** operation cannot complete, because s_k has no right to acquire tokens from s_i. Some comments are due:

- If s_i wants to prevent s_k from re-granting t, s_i can use $t_{s_k}=(s_k, rexp_j, o_j, on)$ instead of $t=(univ, rexp_j, o_j, on)$ as the second parameter of the **grant** call. This *particularization* operation requires, however, that the token t possessed by s_i be universal.

- If there is a disabling token t' in A_i corresponding to an enabling token t being granted, t' must also be copied. Otherwise, s_i would be granting a permanent right while possessing it only as temporary.

- It makes no sense granting a disabling token alone. This would allow the grantor to cancel enabling tokens which he actually did not grant.

- t must be either a *universal token* or (s_k, $*$, $*$, $*$), in order to be transferable to s_k; so having universal tokens is equivalent to being able to grant the rights contained in them. The only way for s_i to grant an (s_k, $*$, $*$, $*$) which is not included in his own access algorithm is to possess the corresponding universal token.

- In order for the transferred tokens to be of some use to s_k, they must be stored at $loc(i,k)$ as a valid access algorithm, which will be called by s_k as a subroutine of his access algorithm if the latter contains a suitable branch to subroutine instruction. As will be seen in section 6, forming a valid access subroutine can only be done by the protection system, like modifying A_k to bind it to that subroutine.

(6) **revoke** (s_i, t, s_k). *Caller*: subject s_i. s_i asks the protection system to remove a token t previously granted to s_k from the access subroutine stored at $loc(i,k)$. If there are many copies of t in the subroutine, they are all removed; also, disabling tokens t' corresponding to t are removed. Then the protection system automatically rebinds s_k's access algorithm A_k to the modified access subroutine. Note that the tokens stored at $loc(i,k)$ have been previously granted by s_i, and therefore this subject can revoke all of them if desired.

All commands described in this section involve access algorithm or access subroutine modification, and thus can only be carried out by the protection system, as will be shown in section 6.

5. Token-Based Access Mechanism

The connection between tokens and accesses must be explained in order to show that the reference monitor is non-bypassable. An enabling token must be *really needed* in order to perform an access. Let us think of processor *SP* as being provided with an internal memory. *SP* proceeds as follows.

Algorithm 1 (Token-Processing Algorithm at *SP*)

- *When an enabling token $t = (*, *, *, on)$ from an access algorithm is processed, SP copies this token to a region of its internal memory called the enabling region. Thus, the enabling region contains a list of active access tokens for the current subject, being therefore a sort of dynamic capability that can also be used for conditions on whether accesses have been made.*

- *When a disabling token $t' = (*, *, *, off)$ is processed, the enabling region is searched for a matching enabling token t. If t is found, then SP deletes it from the enabling region.*

Now let us discuss the way in which *SP* obtains the valid object key *KO* to access an object o_j. Remember that $(Kps(w), Kps(r))$ is a public/secret key pair associated with the protection system: $Kps(w)$ is public and $Kps(r)$ is only known to the protection system. When creating an object (**create_object** command), the protection system uses its secret key to create a signed object header.

Definition 4

An object header H_j *for object* o_j *has the form*

$$H_j = (KSP(w)(Kps(r)(KO(r,o_j))),$$

$$KSP(w)(Kps(r)(KO(w,o_j))),$$

$$KSP(w)(Kps(r)(KO(w,o_d))))$$

where $Kps(r)(\ )$ and $KSP(w)(\ )$ denote encryption under $Kps(r)$ and $KSP(w)$, and o_d is the directory containing o_j. An authenticator (consisting of some redundance) can be appended to each object key $KO(*,*)$ before encrypting it, in order to guard against forgeries. The authenticators can be used to store information about the object type.

Theorem 2

If an object header H_j is stored with each object o_j, then it is achievable that only the security processor SP can authorize a read, write, execute or delete access to o_j. Therefore, under the previous assumptions, the basic protocol is correct, non-bypassable and secure against active intrusions.

Construction

Let us assume that subject s_i wants to access object o_j with an access right $rconst_{ij}$. Then processor *P* goes through step 1 of Protocol 1. At step 2, *EM* starts the key exchange with *SP*. Then at step 3, *SP* searches A_i for a matching enabling ($sexp_i$, $rexp_{ij}$, $oexp_j$, *on*) following substeps 3a, 3b, 3c and 3d. Now, if one is found and substep 3e is reached, the suitable key must be delivered by *SP* to *EM*. We will distinguish four cases, depending on $rconst_{ij}$:

A. $rconst_{ij} = r$ (read access). *SP* reads $KSP(w)(Kps(r)(KO(r,o_j)))$ from the header H_j of o_j. Then it retrieves $KO(r,o_j)$ using $KSP(r)$ and $Kps(w)$, and forwards it to *EM* in an authentic and secret way. The information read from o_j will first go to *EM*, where it will be transparently decrypted under $KO(r, o_j)$ before forwarding it to *P*.

B. $rconst_{ij} = w$ (write access). *SP* reads $KSP(w)(Kps(r)(KO(w,o_j)))$ from the header H_j of o_j. Then it retrieves $KO(w,o_j)$ using $KSP(r)$ and $Kps(w)$, and forwards it to *EM* in an authentic and secret way. The information written by *P* to o_j will first go to *EM*, where it will be transparently decrypted under $KO(w,o_j)$ before recording it into o_j.

C. $rconst_{ij} = e$ (execute access). *SP* reads $KSP(w)(Kps(r)(KO(r,o_j)))$ from the header H_j of o_j. Then it retrieves $KO(r,o_j)$ using $KSP(r)$ and $Kps(w)$, and sends $(KP(w), KO(r,o_j))$ to *EM* in an authentic and secret way. All encryptions performed by *EM* will use that pair of keys consecutively, so that *P* will receive the contents of o_j not in plaintext, but enciphered under $KP(w)$. Since the key $KP(r)$ is only known to *P* and by assumption *P* is trusted, the plaintext of o_j will only be retrievable (and executable) inside *P*. Remark that execute access is a restriction of read access.

D. $rconst_{ij} = d$ (delete access). *SP* reads $KSP(w)(Kps(r)(KO(w,o_d)))$ from the header H_j of o_j. Then it retrieves $KO(w,o_d)$ using $KSP(r)$ and $Kps(w)$, and forwards it to *EM* in an authentic and secret way. Thus the process running at *P* will be able to update the directory o_d by deleting from it the entry corresponding to o_j.

So far, correctness has been proved. Now, it is easy to see that the above implementation of the basic protocol is non-bypassable. This follows from the trusted behaviour of *P*, *SP*, *EM* and the protection system, as well as from the secret keys $KP(r)$, $KSP(r)$ and $KEM(r)$ used for signing messages and $Kps(r)$ used for signing object headers. The protection system assigns object keys to an object at its creation time (when the **create_object** command constructs the object header). Therefore, every read or execute access to an object having been written following the above implementation of Protocol 1 is also forced to follow this implementation. Delete access can be considered a write access to the directory that contains the object. Write access could bypass the basic protocol by not using the intended object keys to write into the object: however, this helps only to leak information, but cannot be a way to fabricate new legal readable objects. Furthermore, illegal on-line writing (and information leaking) is avoided because the three elements *P*, *SP* and *EM* have a trusted behaviour: *P* can only "see" objects through *EM*, and *EM* will refuse to work unless it has received a key from *SP*. Whether a legal accessor can read some information on-line and then leak it off-line is a problem outside the scope of our design.

Finally, it is not possible for an active intruder to replay requests sent previously by *P* at step 1 (impersonation attack), since they are numbered. Numbering also prevents substituting a previously signed request for the one sent by *P* to *SP* at step 1 (substitution attack).□

The efficiency of the implementation for Protocol 1 described in the above construction can be improved substantially if a set of declaration instructions specifying all constant objects referred to by some token in the algorithm is included in each access algorithm. When a subject is created at the beginning of a session (**create_subject** command), processing the declaration section allows *SP* to load all object headers on its internal memory (if they do not exceed its capacity).

Remark 1

All keys in our system have so far been presented as public/secret key pairs: $(KP(w), KP(r))$, $(KSP(w), KSP(r))$, $(KEM(w), KEM(r))$ and $(Kps(w), Kps(r))$. Symmetric cryptosystems can also be considered (*e.g.* the Data Encryption Standard [5]). They are faster, and therefore suitable for I/O bound environments, where the reference monitor is used intensively. However, since shared secret keys are needed, symmetric schemes hinder expandability of the reference monitor: addition of supplementary *EM*s and *P*s needs a secure channel for key exchange or a key-exchange algorithm (which means overhead and again using public-secret key pairs). In the symmetric case, $KP(r) = KP(w) = KP$, $KSP(r) = KSP(w) = KSP$, $KEM(r) = KEM(w) = KEM$ and $Kps(r) = Kps(w) = Kps$ are conventional secret keys, whereas $(KO(w,o_j), KO(r,o_j))$ must remain a public/secret key pair, *since the key for writing into objects cannot be the same as the key for reading them.* If $KO(w,o_j) = KO(r,o_j) = KO(o_j)$ then consider the following attack: a spy with write access first bypasses the reference monitor and reads the ciphertext $Y = K(o_j)(X)$ corresponding to the plaintext X of object o_j (off-line scanning of the disk is not prevented by our system, which mainly relies on encryption to protect information); then it uses the reference monitor to write Y into object o_j, so that $X = K(o_j)(Y)$ is written into o_j; finally, a new disk scanning allows the spy to read the plaintext X corresponding to the deciphered o_j. A similar attack can be devised for a spy with read access to write information into an object o_j in a legally readable format: first write plaintext X into o_j by off-line disk recording, then read $Y = K(o_j)(X)$ using the reference monitor and finally write Y into o_j by off-line disk recording.

6. Integrity and Secure Interrupts

When defining access algorithms, some assumptions concerning their integrity have been made. First *SP* must be able to detect any instruction modification, addition or deletion. Second, only the protection system must be able to endorse an access algorithm: nobody else can authorize an access algorithm. On the other hand, nothing has been said so far about interrupts: it must be guaranteed that when interrupted processes (respectively access algorithms) are resumed by *P* (respectively *SP*), they are really the same as before the interrupt. The system described in ref. 4 can be adapted to meet these basic requirements by encoding access algorithms executed by *SP* and programs executed by *P*. User programs and access algorithms are encoded and sealed by the protection system since their creation (**create_object** and **create_algorithm** commands, respectively); the operating system executable images must be encoded from start, in order for log-in processes also to be protected.

6.1 Program and Access Algorithm Encoding

We will only recall here the basic operating principles of the system [4]. Let $i_1, \ldots, i_n$ be a program (respectively an access algorithm), where i_k is a machine language instruction (respectively access language instruction or an access token, Definition 3); i_n is not a branch—it can be an **END** instruction. A normalized instruction format is defined: the length of all i_k is made equal by appending a known filler and then each i_k is appended a redundance pattern, whose length depends on the desired security level. Call the normalized algorithm $I_1, \ldots, I_n$.

Let F be a one-way function (such that F^{-1} is "hard" to compute, see ref. 3), such that in general $F(X \oplus Y) \neq F(X) \oplus F(Y)$. Now, if $I_1, \ldots, I_n$ is a *sequential block* (containing no branches) it will be encoded into a *trace sequence* $T_0, T_1, \ldots, T_n$, where traces are computed in reverse order according to the following equalities:

$$\begin{aligned} T_0 &= F(T_1) \oplus I_1 \\ T_1 &= F(T_2) \oplus I_2 \\ &\vdots \\ T_{n-1} &= F(T_n) \oplus I_n \\ T_n &= F(I_n) \end{aligned} \qquad (1)$$

Now let us introduce branches. If I_k is a *forward unconditional branch* to I_j (*i.e.* $k < j$), it is translated as:

$$\begin{aligned} \ldots\ T_{k-1} &= F(T_k) \oplus I_k \\ T_k &= F(T_j) \oplus I_j \\ T_{k+1} &= \ldots \\ &\vdots \\ T_j &= \ldots \end{aligned} \qquad (2)$$

When I_k is a *forward conditional branch to instruction* I_j, the following traces are computed (also in an index decreasing order):

$$\begin{aligned} \ldots\ T_{k-1} &= F(T_k) \oplus I_k \\ T_k &= F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ T_{k'} &= F(T_j) \oplus I_j \\ T_{k+1} &= \ldots \\ &\vdots \\ T_j &= \ldots \end{aligned} \qquad (3)$$

A *branch to subroutine* (non-recursive subroutine stored separately, like the access subroutines used for granting rights in the **grant** command) at I_k *is similar to a conditional branch from an instruction flow viewpoint*, in the sense that there are two target instructions, namely the first subroutine instruction and the next instruction of the calling program. So, to ensure that a called subroutine already encoded as $\hat{t}_0, \hat{T}_1, \ldots, \hat{T}_m$ will not be replaced, $\hat{T}_0$ is retrieved from $\hat{t}_0 = Kps(r)(\hat{T}_0)$ using $Kps(w)$ (the heading trace t_0 is further explained below) and is included in the calling program as follows:

$$\begin{aligned} &\ldots T_{k-1} = F(T_k) \oplus I_k \\ &T_k = F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ &T_{k'} = \hat{T}_0 \\ &T_{k+1} = \ldots \end{aligned} \tag{4}$$

It can be inferred from the above mechanism that the subroutine requires its own trace computation and authority's endorsement (just as stated when discussing the **grant** command in section 4). As for backward branches from I_k to I_j (*i.e.* $k \geqslant j$), they are slightly more complicated since the trace T_j corresponding to I_j cannot be used to compute T_k or $T_{k'}$ as in forward branches, because T_j follows T_k in the trace computation and depends on T_k (reverse trace computation). Let $\tilde{T}(j)$ be a one-to-one integer function on j, for example the identity function. A *backward unconditional branch* at instruction I_k to instruction I_j is translated as:

$$\begin{aligned} &\ldots T_{j-1} = F(\tilde{T}_j) \oplus F(T_j) \oplus I_j \\ &\tilde{T}_j = F(T_j) \oplus \tilde{T}(j) \\ &T_j = \ldots \\ &\vdots \\ &T_{k-1} = F(T_k) \oplus I_k \\ &T_k = F(\tilde{T}(j)) \oplus I_j \\ &T_{k+1} = \ldots \end{aligned} \tag{5}$$

Finally, the trace structure for a *backward conditional branch* is as follows:

$$\begin{aligned} &\ldots T_{j-1} = F(\tilde{T}_j) \oplus F(T_j) \oplus I_j \\ &\tilde{T}_j = F(T_j) \oplus \tilde{T}(j) \\ &T_j = \ldots \\ &\vdots \\ &T_{k-1} = F(T_k) \oplus I_k \\ &T_k = F(T_{k'}) \oplus F(T_{k+1}) \oplus I_{k+1} \\ &T_{k'} = F(\tilde{T}(j)) \oplus I_j \\ &T_{k+1} = \ldots \end{aligned} \tag{6}$$

Like I_k *and* T_{k-1} *in all this section,* I_j is presented for convenience in the trace T_{j-1}, but it need not be so (for example T_{j-1} can be a branch trace to somewhere else). To avoid recomputation of T_j during an interrupt preceding a backward branch to it, this trace must be signed (see end of section). A *recursive branch to subroutine* can be viewed as a backward branch to the first subroutine instruction, with the proper context saving: thus, the subsequent trace structure follows from the one for backward conditional branches in the same way as non-recursive subroutine branches follow from forward conditional branches. It is not difficult to write a program for generating all trace structures discussed in this section, given a standard program or an access algorithm as input.

After the previous trace computation, it is up to the protection system to endorse the trace sequence with a signature on the heading trace T_0, and on every $\tilde{T}_j$ in a backward branch target. So the protection system uses the key $Kps(r)$ to replace T_0 and the $\tilde{T}_j$s with $t_0 = Kps(r)(T_0)$ and $\tilde{t}_j = Kps(r)(\tilde{T}_j)$, respectively. Finally, the traces are stored in an increasing order $t_0, T_1, \ldots, T_n$ and will be used in lieu of the original $I_1, I_2, \ldots, I_n$.

Remark 2

It is important to stress that trace computation can in principle be conducted by anyone in the system, since it requires no secret keys, except for obtaining t_0 and the $\tilde{t}_j$s. To increase performance and relieve the protection system, standard P programs can be encoded by the same subject that creates them (then the **create_object** command just has to compute t_0 and the $\bar{t}_j$s).

6.2 Properties of the Coding Scheme

The two main properties concerning the coding outlined in section 6.1 will be stated here. The proofs of both results are a straightforward extension of those in ref. 4 and are not included here by reason of space and readability. Assuming that P (respectively SP) contains a preprocessor for evaluating, *i.e.* decoding, the above trace structure and retrieving programs (respectively access algorithms), then we have:

Theorem 3 (Correctness)

The program (respectively access algorithm) $i_1, \ldots, i_n$ can be retrieved and executed from its corresponding trace sequence $t_0, T_1, \ldots, T_n$.

The proof of Theorem 3 is a construction and it shows that, if a preprocessor is pipelined to the main processor in P (respectively SP), then there is no significant increase in the execution time. For example, for a sequential algorithm, i_1 is retrieved by computing $I_1 = F(T_1) \oplus Kps(w)(t_0)$; and the rest of the instructions i_k are retrieved by $I_k = F(T_k) \oplus T_{k-1}$; then while i_k is being retrieved by P's (respectively SP's) preprocessor, i_{k-1} can be executed by the main processor. The second result refers to the integrity property:

Theorem 4 (Run-Time Integrity)

If a program (respectively access algorithm) $i_1, \ldots, i_n$ is stored as $t_0, T_1, \ldots, T_n$ and is evaluated as described in the proof of Theorem 3, any instruction substitution, deletion or insertion will be detected at run-time, thus causing the main processor of P (respectively SP) to stop executing before the substituted, deleted or inserted instruction(s). Moreover, only the last five read traces need be kept in the internal secure memory of P (respectively SP).

Theorem 4 means that, in order to detect any modification when resuming an interrupted program (respectively access algorithm), it suffices for the interrupt servicing routine to save the last five read traces of the interrupted program (respectively access algorithm) in the internal secure memory of P (respectively SP). Program and access algorithm integrity also follow from the theorem. Finally, by construction (one-way functions and signatures on t_0 and the $\bar{t}_j$s), the scheme restricts creation of valid access algorithms to the protection system.

7. Conclusion

This paper presents a thorough design of an access control system, which decides the accessibility relationships between subjects and objects in an algorithm-sequenced way. First the model is stated (section 2), and then (sections 3–6) solutions are given for the main issues of the model. The proposed reference monitor is highly modular, and it allows an easy generalization to a parallel structure that minimizes slowdown. Conventional software support for security is discarded: instead, a combination of cryptographic and hardware support is used. In this way, a new cryptography-based method is used to provide executable code integrity and secure interrupts—two fundamental requirements of a secure system.

References

[1] *Guide to VMS System Security*, Digital Equipment Corporation, Maynard, MA, June 1989.

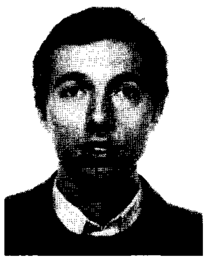
[2] D. E. Denning, *Cryptography and Data Security*, Addison-Wesley, Reading, MA, 1982.

[3] W. Diffie and M. E. Hellman, New directions in cryptography, *IEEE Trans. Inf. Theory*, 22 (1976) 644–654.

[4] J. Domingo-Ferrer and L. Huguet-Rotger, A cryptographic mechanism for non-iterative algorithm enforcing, in *Proc. 4th IFIP International Conference on Data Communication Systems and Their Performance*, G. Pujolle and R. Puigjaner (eds.), Elsevier, Amsterdam, 1991, pp. 425–438.

[5] Data Encryption Standard, FIPS PUB46, National Bureau of Standards, Washington, DC, January 1977.

[6] S. Muftic, *Security Mechanisms for Computer Networks*, Ellis Horwood, Chichester, 1989.



Josep Domingo-Ferrer received the diploma, M.Sc. and Ph.D. degrees in computer science from the Universitat Autònoma de Barcelona in 1988, 1989 and 1991, respectively, all with honours (Extraordinary Graduation Award 1988). He also earned a B.Sc. degree in mathematics from the UNED, Madrid, 1989. His Master's thesis presented a prototype secure communications network (SICNET), where security was almost exclusively achieved using cryptography.

Dr Domingo-Ferrer has published several papers and articles (in *Eurocrypt* and *Computers & Security*, among others) on cryptography, computer security and secure OSI communications, and has spent research visits at the University of Wisconsin-Milwaukee and at the Siemens AG Corporate Research Centre in Munich, Germany. The author's interests and Ph.D. thesis focus on the enhancement of access control through the design and use of software integrity and distributed identification mechanisms. He is a member of IEEE and IACR.