Memory protection in embedded 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 — With reference to an embedded system featuring no support for memory management, we present a model of a protection system based on passwords. At the hardware level, our model takes advantage of a memory protection unit (MPU) interposed between the processor and the complex of the main memory and the input-output devices. The MPU supports both concepts of a protection context and a protection domain. A protection context is a set of access rights for the memory pages; a protection domain is a set of one or more protection contexts. Passwords are associated with protection domains. A process that holds a given password can take advantage of this password to activate the corresponding domain. A small set of protection primitives makes it possible to modify the composition of the domains in a strictly controlled fashion.

The proposed protection model is evaluated from a number of important viewpoints, which include password distribution, review and revocation, the memory requirements for storage of the information concerning protection, and the time necessary for password validation.

Keywords: access right; embedded system; password; protection; revocation.

1. INTRODUCTION

We shall refer to a typical embedded system architecture featuring a microprocessor interfacing both volatile and non-volatile primary memory devices, as well as a variety of input/output devices including sensors and actuators. In a system of this type, no provision is usually made for a memory management unit translating the virtual addresses generated by the processor into physical addresses in the primary memory [7], [15]. This situation is not likely to change in the near future; instead, advances in integration technologies will presumably be exploited to reduce system cost and size, rather than to introduce sophisticated forms of storage space addressing and management [6], [18].

In a system featuring no form of virtual to physical address translation, a single address space is shared by all processes; the meaning of an address is unique and is independent of the process that generates this address [11], [20], [22]. In fact, the two aspects, addressing and protection, are unified in the multiple address space model, and are kept separated in a single address space environment [3]. This facilitates data sharing between processes and interprocess interactions. In particular, two or more processes sharing a given data item will simply use the address of this data item, which is unique throughout the system. On the other hand, an erroneous or deliberately harmful process is not prevented from accessing and possibly corrupting the private information items of a different process or even the kernel [12], [30]. In a situation of

this type, provision of mechanisms for the protection of the private information is highly desirable [8], [15], [21], [32].

This requirement for memory protection extends to the components of the same given process. In fact, embedded system programming is a challenging task, as a consequence of the stringent limitations in terms of available memory, the multiplicity of different classes of sensors and actuators, the requirements for concurrency and real-time response, and the limited support for program debugging. Ideally, the *principle of least privilege* [25], [28] should be followed, according to which each software component should be given the ability to access only those memory areas that are indispensable for that software component to carry out its job.

1.1. Access privilege specification

In every protection system model, an important problem is the specification of the *access rights* held by each active entity (*subject*) on the passive entities (*objects*) of the system [26]. A subject is any software entity that can generate memory accesses; thus, a subject can be a scheduled computation (e.g. a process) as well as, in an event-driven environment, an execution activity generated by a hardware interrupt (interrupt handler) [6], [18]. To simplify the presentation, without loss of generality, in the following we shall refer to a process-oriented environment. A *protection domain* is a set of access rights for the protected objects. When a given subject is executed, it is associated with a protection domain that states the objects that this subject can access and the operations it can accomplish on these objects.

In our protection model, the memory address space is logically partitioned into fixed-size pages, and the memory pages are the protected objects. Access right *read* for a given page makes it possible to access the page for read; this is similar to access right *write* for write accesses and to access right *execute* for the execution of the page contents (supposedly, machine code). A protection domain is the specification of a set access rights for the memory pages. In the following, we shall hypothesize that input/output devices are mapped into the single address space, so that the mechanisms for memory protection also apply to the internal registers of the peripheral units.

When a new process is started up, it is associated with an initial domain, and execution begins in this domain. When a process switch takes place (the active process releases the processor and a new process is assigned to the processor) the domain of the new process is activated. An important aspect is that activation of a new domain can take place even in the execution of a single process, as follows from to the evolution of the control flow of that process.

This will be the case, for instance, when execution of a software component is started up; congruently with the principle of least privilege, the composition of the new domain will reflect the memory access requirements of the new component. In this way, if an erroneous software component causes an illegal access attempt to a memory area external to its own protection domain, an exception of violated protection is raised, which reveals the error and limits its consequences.

1.2. Passwords

In a classical protection paradigm, one or more *passwords* are associated with every given protected object; each password corresponds to an access privilege expressed in terms of a subset of the access rights defined by the type of that object [3], [11], [22], [24]. A process aimed at accessing a given object must present a password for this object. The access terminates successfully only if this password matches one of the passwords associated with the object, and the access privilege corresponding to this password includes an access right permitting the access; if this is not the case, an exception of violated protection is raised and the access terminates with failure.

In this work, we shall refer to a variant of the classical password paradigm whereby each password is associated with a protection domain instead of a single protected object. In our page-oriented protection environment, this means that a single password may grant access permissions to several memory pages. A software component that holds a password for a given domain is entitled to access all the memory pages included in this domain, according to the respective access rights. When execution of a software component begins, it presents a password to the protection system. The protection domain corresponding to this password is determined. The software component will be executed with the access rights for the memory pages that are part of this protection domain.

An important aspect of the password approach to object protection is that passwords do not need to be segregated into protected memory regions [22], [24]; instead, they can be freely mixed with ordinary data and can be manipulated by the usual machine instructions for data processing. In fact, if passwords are large and chosen at random, the probability of guessing a valid password is vanishingly low. In particular, passwords can be freely transmitted between software components, and in this case, the recipient of a password will be in a position to activate the corresponding domain and use this domain for its own memory accesses. An aspect related with password distribution is *password derivation*, whereby a password for a given do-

main is transformed into a password for a different, possibly (but not necessarily) weaker domain, e.g. with a few pages excluded, or restricted access rights. Password derivation can be necessary, for instance, for distribution of limited access permissions between software modules.

Simplicity in password distribution, obtained by a simple action of a password copy, implies that the recipient of a password is free to distribute this password to other subjects. Thus, passwords tend to spread over the entire memory system. A related problem is that of the revocation of access privileges. The protection system should incorporate mechanisms that make it possible to reduce the set of access rights corresponding to a given password. It should also be possible to revoke the password, so that it is not viable to use password derivation to circumvent revocation.

In this work, we shall present a model for a hardware/software complex designed to implement a protection environment based on passwords. At the hardware level, a *memory protection unit* (MPU) is interposed between the processor and the memory and input/output devices (Figure 1). The MPU intercepts the memory addresses generated by the processor and uses its own internal registers to ascertain whether the current memory access is authorized; if this is not the case, the MPU inhibits the access and generates an interrupt request to the processor. At the software level, our protection environment implements a system of passwords that supports password distribution and derivation as well as password revocation.

The rest of this paper is organized as follows. Section 2 introduces the memory protection unit in the form of a low-cost addition to the hardware of the microcontroller. Section 3 presents the mechanisms for password generation and derivation, and delineates the association of passwords with protection domains. Section 4 introduces a set of primitives of the protection system, the *protection primitives*, which make it possible to activate a domain, and to modify the composition of the domains in a strictly controlled fashion. Section 5 discusses the proposed protection model from a number of important viewpoints, which include the memory requirements for storage of the information concerning protection, the time necessary for password validation, the review and revocation of access privileges, and the cost of the proposed MPU in terms of additional hardware. Section 6 describes the relation of our work to previous work. Section 7 gives concluding remarks.

2. THE MEMORY PROTECTION HARDWARE

As anticipated in Section 1, in our protection system the memory pages are the elementary units of information on which protection is exercised. We have defined a protection domain as

read, *write* and *execute* access rights for a collection of memory pages; this definition can be extended and refined in terms of subdomains that we call *protection contexts*. Both the concepts of a protection context and a protection domain are supported at the hardware level by memory protection circuitries that we collectively call the memory protection unit (MPU, Figure 2).

In detail, the MPU implements a limited number *c* of protection contexts, as follows. A set of registers, the *context registers* $CR_0, CR_1, \ldots$ is associated with the memory pages, one context register for each page. Context register CR_i associated with page P_i is partitioned into three *context fields*. The read context field consists of c bits; the j-th bit, if asserted, indicates that the j-th protection context includes access right *read* for page P_i (Figure 3). This is similar to the write context field for access right *write*, and to the execute context field for access right *execute*. Thus, the bits in the j-th position of the three context fields of CR_i collectively identify the access rights included in the j-th protection context for page P_i. The composition of protection context j is determined by the contents of all context registers for this protection context.

Thus, for instance, in an MPU configuration featuring four protection contexts ($c = 4$), the size of a context register is 12 bits. If the configuration of context register CR_i is (0011 0010 0100), then for page P_i context 0 includes access right *read*, context 1 includes both access rights *read* and *write*, context 2 includes access right *execute*, and context 3 includes no access right at all.

A protection domain is a set of one or more protection contexts. At any given time, the *active domain* is the protection domain associated with the process being executed at that time (the *active process*). A c-bit internal register of the MPU, the *domain register* (DR), features a bit for each protection context. The composition of the active domain in terms of protection contexts is determined by the contents of the domain register. If the j-th bit of this register is asserted, then the active domain includes the j-th context. The access rights in the active domain are the union of the access rights in all the protection contexts selected by the domain register.

In the foregoing example, if domain register DR contains quantity 0010, then the active domain is formed by a single protection context, CR_1, that is, for page P_i, it includes access rights *read* and *write*. If DR contains quantity 0101, the active domain is formed by protection contexts CR_0 and CR_2, that is, for page P_i, it includes access rights *read* and *execute*.

The configuration of the protection contexts in terms of access rights is stated at the global level and is the same for all processes. Instead, protection domains are configured in terms of

sets of protection contexts at the process level. At any given time, the active process can accomplish those memory accesses that are made possible by the access rights in the active domain. When a process switch takes place (the active process releases the processor and a new process is assigned to the processor), the contents of the domain register are saved into the descriptor of the active process, and the domain register is filled with quantities taken from the descriptor of the new process.

3. PASSWORD CHAINS

A function $F(x)$ is *one-way* if it is easy to compute but hard to invert [2], [13], [16]. This means that given a value x, quantity $F(x)$ can be computed at little effort, but given a value y it is computationally unfeasible to determine a value x such that $y = F(x)$. A *one-way chain* is a sequence of values $x_0, x_1, \dots, x_{v-1}$, where x_0 is the *seed* and each x_i is the result of the evaluation of a one-way function F on the previous value, i.e. $x_i = F(x_{i-1})$ for $0 < i \leq v-1$. Let $F^n(x)$ denote the result of n successive applications of function F on x, e.g. $F^2(x) = F(F(x))$. We have $x_1 = F(x_0), x_2 = F^2(x_0), \dots, x_{v-1} = F^{v-1}(x_0)$.

A function $H(x, p)$ is a *parametric one-way function* if given a value z and a parameter p, it is computationally unfeasible to determine a value x such that $z = H(x, p)$ [27], [34]. An example of practical implementation is $H(x, p) = G(E_x(p))$ where G is a hash function and $E_x(p)$ denotes the encryption of p using symmetric cipher E with key x. Only ciphers that are secure against known plaintext attacks should be used, e.g. DES [34]. Thus, a parametric one-way function corresponds to a collection of one-way functions, a one-way function for each value of the parameter.

In our protection environment, a parametric one-way function H and different parameters can be effectively used to generate the passwords and organize them into one-way chains called *password chains*, as follows. A function H is chosen at system generation time. When a process Q is created, a new parameter is reserved for Q and is used to derive a one-way function F_Q from H; this function is associated with Q, and will never be used for a different process. A new password chain is generated for Q, and the *length* m (i.e. the number of passwords) of this chain is part of the process design. To setup the password chain, a seed is chosen at random; the seed is called the *master password* of process Q, and is denoted by w_0. The subsequent passwords $w_1, w_2, \dots, w_{m-1}$ of process Q are derived from the master password by consecutive applications of one-way function F_Q; we have $w_1 = F_Q(w_0), w_2 =$

$F_Q(w_1), \dots, w_{m-1} = F_Q(w_{m-2})$. Function H is universally known, and the value of the parameter for process Q is only known to Q; this means that Q is the only process that can take advantage of F_Q. As will be shown shortly, this is an important factor in password derivation and distribution.

3.1. Protection domains

Each password is associated with a protection domain. This association is recorded in a system table, called the *password table*, featuring one entry for each existing password. The entry for a given password specifies the domain associated with that password in terms of a configuration of the domain register. Efficiency considerations suggest that the password table should be split into several tables, one table for each password chain; this issue will be investigated in depth in Section 5.1.

A software component that holds a given password can take advantage of this password to activate the protection domain associated with this password. In detail, when the password is presented to the protection system, a search is made in the password table for an entry containing that password. If this search is successful, the corresponding domain configuration is extracted from this entry and is loaded into the domain register.

The composition of the domain corresponding to a password assigned to a given software component will be expressed by a configuration of the domain register that will be decided in accordance with the specific access privilege requirements of this software component, as follows from application of the principle of least privilege.

Let us consider, for instance, a domain organization representing domains by concentric *protection rings* [29]. The most external ring is the least privileged and usually has the highest ring number, the central ring is the most privileged and is usually numbered 0, and each intermediate ring is more privileged than the outer rings. In our protection environment, an m-ring protection model corresponds to a password chain $w_0, w_1, \dots, w_{m-1}$ of length m, in which master password w_0 corresponds to the central ring, the i-th password w_i corresponds to the i-th ring, and the m-th password w_{m-1} corresponds to the most external ring. A software component designed to be executed in the i-th protection ring holds password w_i. The association of protection contexts with domains is incremental: the domain associated with password w_i (ring i) is obtained by adding one or more protection contexts to the domain associated with password w_{i+1} (ring $i + 1$), so that, for instance, the domain associated with password w_0 (ring 0) includes the protection contexts of the domains associated with all the subsequent passwords $w_1, w_2, \dots, w_{m-1}$.

The protection ring model is an example of a hierarchical domain organization; this is not a requisite for password chains. Consider, for instance, a process consisting of two software components: a first component writes executable machine code into a memory page P, and this machine code is actually executed by the other component. In the password chain, the first password will be reserved for the first component and the domain associated with this password will include access right *write* for page P. The second password will be reserved for the second component, and the corresponding domain will include access right *execute* for P. Indeed, in this application, the protection domains are not hierarchical.

3.2. Password distribution and derivation

A process Q that holds a given password can distribute that password to another process R by a simple action of a password copy. In this way, R acquires the access rights in the protection domain associated with that password.

Let $w_0, w_1, \ldots$ be the passwords in the password chain of process Q, and let F_Q be the one-way function associated with Q and used to generate the password chain. If Q holds password w_0, it may distribute a password *derived* from w_0, i.e. a password w_i that follows w_0 in the password chain. To this aim, Q applies one-way function F_Q iteratively i times to w_0 to obtain w_i, as follows from relation $w_i = F_Q(w_{i-1}) = F_Q^2(w_{i-2}) = \ldots = F_Q^i(w_0)$.

Of course, process R that receives password w_i from process Q is not in a position to derive a password that *precedes* w_i in the password chain of Q. This is a consequence of the fact that function F_Q is one-way, and is computationally not invertible. Furthermore, process R cannot apply forward derivation to w_i, as it does not possess F_Q (if R uses its own one-way function F_R, the result will be meaningless).

In summary, a one-way function F_Q is associated with process Q when this process is created. In an implementation that uses a parametric-one-way function H, a result of this type will be obtained by reserving a value p of the parameter for Q. F_Q is used to generate a password chain $w_0, w_1, \ldots, w_{m-1}$, where m is specific to Q. The values of the passwords in this password chain are inserted into the password table; each password will be associated with a domain configuration whose composition in terms of protection contexts is part of the process design. Finally, quantity p, the value w_0 of the master password, and the initial configuration of the domain register are inserted into the descriptor of process Q. When execution of Q is subsequently started up, the initial configuration of the domain register is loaded from its process descriptor.

4. PROTECTION PRIMITIVES

The protection system defines a set of primitives, the *protection primitives*, which allow us to activate a domain, and to modify the composition of the domains in a strictly controlled fashion (Table I). We shall now present these primitives and the actions involved in the execution of each of them. These primitives are designed to be fully implemented at software level by kernel routines (thus, no modification is implied by our form of memory protection in the instruction set of the processor). These routines run in the privileged state, as is necessary to access the internal registers of the MPU and the password table, which are part of the memory regions reserved for the kernel.

4.1. Domain activation

The $activate()$ protection primitive makes it possible to switch from the current domain to the protection domain corresponding to a given password w. This primitive has the form

$activate(w)$

Its execution produces a search in the password table for password w. If this search is successful, the domain configuration corresponding to w is extracted from the password table and is loaded into domain register DR.

As an example of application, let us consider a process Q whose password chain consists of master password w_0 and a second password w_1, let D_0 and D_1 be the protection domains corresponding to these passwords, and let DR_0 and DR_1 denote the domain register configurations for these domains. Suppose that process Q consists of a main program M, designed to be executed in domain D_0, and a software component C designed to be executed in domain D_1. If, for instance, D_1 is a subset of D_0 (that is, it consists of a subset of the contexts that form D_0), then execution of component C will be started up with reduced access privileges, congruently with the activities to be carried out by this component, as follows from application of the principle of least privilege. When process Q is created, quantity DR_0 is stored into the process descriptor; this quantity will be loaded from the process descriptor into the domain register when execution of Q begins, thereby activating domain D_0, as is required for the execution of main program M. Subsequent execution of component C will be preceded by a call to $activate(w_1)$, which loads quantity DR_1 from the password table into the domain register, thereby causing a switch from domain D_0 to domain D_1. At the end of execution of C, a call to $activate(w_0)$ will be used to return to domain D_0.

4.2. Domain composition

Two primitives, called $grant()$ and $revoke()$, make it possible to alter the composition of the domains in terms of protection contexts. These primitives make it possible to access the contents of the password table to modify these contents in a strictly controlled fashion. Let us consider a component of the active process that holds the master password of the password chain of this process (the *active password chain*). This software component can take advantage of these primitives to modify the protection domains associated with the subsequent passwords in the active password chain, by adding or removing the protection contexts that are included in the domain associated with the master password.

In more detail, let Q be the process of the software component executing $grant()$, and let $w_0, w_1, \ldots$ be the passwords in the password chain of Q. Furthermore, let D_0 and D_i be the domains associated with master password w_0 and password w_i, and let DR_0 and DR_i denote the domain register configurations for these domains. The $grant()$ primitive has the form

$grant(w_0, i, msk)$

Argument w_0 is the master password, argument i identifies password w_i, and argument msk is a bit configuration of the same size as the domain register. For each bit in msk that is asserted, execution of this primitive adds the corresponding protection context to domain D_i, provided that this context is part of domain D_0. Execution of this primitive reads domain register configurations DR_0 and DR_i from the entries of the password table corresponding to w_0 and w_i. The new value of DR_i is computed by setting the bits that are asserted in both DR_0 and msk. Finally, the result is written back into the password table entry for w_i.

The $revoke()$ primitive is as follows:

$revoke(w_0, i, msk)$

Arguments w_0, i and msk are the same as in the $grant()$ primitive. For each bit in msk that is asserted, execution of this primitive removes the corresponding protection context from domain D_i, provided that this context is part of domain D_0. Execution of this primitive reads quantities DR_0 and DR_i from the password table. The new value of DR_i is computed by clearing the bits that are asserted in both DR_0 and msk; the result is written back into the password table.

4.3. Password derivation

A further protection primitive, the $derivePassword()$ primitive, is used for password derivation. Let Q be the process of the software component executing $derivePassword()$, let $w_0, w_1, \ldots$ be the passwords in the password chain of Q, and let F_Q be the one-way function

associated with Q and used to generate the password chain. The $derivePassword()$ primitive has the form

$$w_{i+j} \leftarrow derivePassword(w_i, j)$$

The arguments are a password w_i and an index j, called the *derivation index*. Let w_{i+j} denote the j-th password following w_i in the active password chain. This primitive returns password w_{i+j}. This password is obtained from w_i by applying function F_Q iteratively j times, as follows from relation $w_{i+j} = F_Q(w_{i+j-1}) = F_Q^2(w_{i+j-2}) = \ldots = F_Q^j(w_i)$. This primitive cannot be used for a password derivation starting from a password that is not part of the active password chain, as it uses one-way function F_Q (the result would be meaningless).

5. DISCUSSION

In low power microcontroller, the overall application memory layout is simple and quite static [33]. On the other hand, static memory allocation is not a prerequisite of our protection system, which is able to comply with forms of dynamic memory allocation as well. The programmer will define the memory requirements of each process, implicitly by the program structure, or explicitly by inserting ad-hoc directives in the source program. The compiler will generate the initial configurations of the context registers, as well as the configuration of the domain register that corresponds to each password in the password chain of that process.

5.1. Considerations concerning performance

As anticipated in Section 3.1, efficiency considerations suggest splitting the password table into several tables, one table for each password chain. Let $w_0, w_1, \ldots, w_{m-1}$ be the password chain of process Q, where quantity m is specific to Q. We shall denote the corresponding password table by PT_Q. This table features m entries; the i-th entry contains password w_i and the specification of the domain associated with this password, expressed in terms of a configuration of domain register DR.

In this implementation of the password table, a password assumes the form of pair (w, Q), where Q indicates a process. This means that password w is part of the password chain of process Q and consequently, the corresponding domain configuration will be contained in password table PT_Q. Of course, with respect to an implementation featuring a single password table for all processes, significant advantages follow in terms of the processing time required for password validation. The expected number of password comparisons to find the given password is $(m + 1)/2$. On the other hand, the memory cost for password storage is increased by

the necessity to include quantity Q in the representation of a password. Up to 256 individual software activities can be reputed a good trade-off between usability and memory cost. In this hypothesis, the memory requirement for storage of quantity Q is one byte, and is low.

The time necessary for password validation can be reduced to a single password comparison if each password is stored in the form of a triple (w_i, Q, i), where Q indicates a process, and i denotes the i-th entry of password table PT_Q, i.e. the entry reserved for password w_i (Figure 4). Of course, if the contents of this entry do not match quantity w_i, password validation fails. Time efficiency is improved at the expense of the increased memory size of a password, as is required to store quantity i; if the length of a password chain is up to 16 passwords, four bits will be sufficient, so the additional memory cost is negligible.

A further space-time tradeoff can be conceived whereby less space is necessary for storage of the password table but the cost in terms of processing time for password validation increases. In this new approach, for process Q, we maintain the value of a single password, that is, master password w_0. The password table assumes the form of a domain table featuring one entry for each password in the password chain of Q; the i-th entry contains the specification of the domain associated with the i-th password w_i, expressed in terms of the corresponding configuration of the domain register. When password (w_i, Q, i) is presented for validation, one-way function F_Q is iteratively applied i times to w_0, as follows from relation $w_i = F_Q(w_{i-1}) = F_Q^2(w_{i-2}) = \ldots = F_Q^i(w_0)$. If the result matches password w_i, then password validation is successful. In this case, the specification of the domain corresponding to w_i is extracted from the i-th entry of the domain table. Of course, in this approach the space necessary for password storage is kept to a minimum, but processor time is necessary to evaluate several passwords at each password validation. If the length of a passwords chain is m passwords, the expected number of password evaluations is $(m - 1) / 2$.

5.2. Access privilege revocation

As shown in Section 3.2, a software component can distribute a password to another software component by a simple action of a password copy; so doing, the corresponding access privilege is transmitted to the recipient. As a consequence of this simplicity in access privilege distribution, access privileges tend to spread throughout the system. A related problem is that of access privilege revocation; a process should be given the ability to revoke the validity of the passwords in its own password chain. In our system, a result of this type can be simply obtained by changing the one-way function used to construct the password chain. So doing, we replace all the passwords in the password chain except the master password. An action of this

type invalidates all the passwords that match the old passwords.

As seen in Section 3, a parametric one-way function corresponds to a collection of one-way functions, a one-way function for each value of the parameter. In our system, when a process is created, a value of the parameter is chosen at random and is reserved for that process. Then, the one-way function obtained in this way is used to generate the password chain for the new process. In an implementation of this type, replacement of the one-way function is simply obtained by changing the value of the parameter, at low cost in terms of processing time. The new passwords will be evaluated using the new one-way function; these passwords will be inserted into the password table.

Despite its simplicity, this password revocation mechanism results to possess several interesting properties. Revocation is [10]:

- *Transitive*. If a given password is revoked, the effects of the revocation extend to all the copies of this password, to all the passwords derived from the revoked password and to all their copies, independently of the software components holding them. In fact, all these passwords are part of the same password chain, a copy of a given password is indistinguishable from the original, and passwords have no memory of the consecutive copy actions.
- *Temporal.* The effects of a revocation can be reversed by taking advantage of the same mechanism used for the revocation. After replacement of the one-way function, the validity of the old passwords can be restored by simply returning to the previous one-way function. In an implementation using a parametric one-way function, a result of this type is simply obtained by returning to the previous value of the parameter.
- *Deferred.* A process that activated the domain associated with a given password (e.g. by issuing the $activate()$ protection primitive to load the corresponding domain configuration into the domain register) is allowed to continue to use this domain even after this password has been revoked, until the activation of a new domain. This is a consequence of the fact that revocation involves the one-way function, and does not alter the contents of the domain register. Deferred revocation can be important to avoid that a software component remains in an inconsistent state [10], as will be the case if an immediate revocation comes into effect when an operation is being executed on a given object and the operation is prevented to terminate successfully. On the other hand, if a form of immediate revocation is necessary, e.g. in the case of a security violation, the operating system may well force modifications of the contents of the context registers and the domain register. Of course, situations of this type may well be in contrast with the normal relationships between the

software components.

- *Independent*. Two passwords associated with identical domains, which are part of different password chains, can be revoked independently of each other. In fact, the change of the one-way function of a given password chain has no effect on the validity of the passwords in a different password chain, independently of the domains associated with these passwords.

The revocation mechanism based on replacement of the one-way function revokes all the password in the password chain except the master password. Thus, the effects of this mechanism are global for the passwords of the given process, and involve all the passwords matching the revoked passwords, independently of their present location in the memory system and of the processes that hold these passwords. In fact, our protection system is also able to support a form of local revocation of access rights, to reduce the extent of selected domains. A result of this type can be obtained by repeated executions of the $revoke()$ protection primitive. In detail, as seen in Section 4.2, $revoke()$ allows a software component that holds the master password of the active password chain to modify the composition of the domain associated with another password in this password chain, by eliminating protection contexts from this domain.

5.3. Hardware costs

In a system featuring a memory management unit (MMU) interposed between the processor and the primary memory system, the MMU is usually aimed at: (i) translating the virtual addresses generated by the processor into physical addresses in the primary memory; (ii) implementing a virtual memory system (that is, a virtual memory space much larger than the physical space); and (iii) supporting forms of memory protection and security [9]. Typical MMU implementations take advantage of a page table stored in the primary memory. The page table features an entry for each page of the virtual space. The entry for a given virtual page specifies the number of the corresponding physical page in the primary memory. The physical page number is paired with the offset to obtain the physical address of the referenced information item. An important problem is to avoid that, for each memory access, a second memory access is necessary to inspect the page table. To this aim, the MMU contains an associative cache of the page table in the form of a translation lookaside buffer (TLB) [14]. When the address translation circuitry receives a logical address, all the TLB entries are searched simultaneously for the corresponding physical address. If no match is found in the TLB, the TLB is loaded with quantities taken from the page table in the primary memory. If no free entry is available in the TLB, the least recently used algorithm is typically used to find an entry to be replaced.

The page table and the TLB are normally used also to enforce forms of memory protection. Each page is associated with a set of access rights specifying the operations that can be carried out on this page, and the typical access rights are read, write and execute. When the processor attempts to access an information item in a given page, the access rights for this page are compared with the type of access. The access is permitted only if the access rights are actually verified.

When a process switch occurs (the current process releases the processor and a new process is assigned to the processor) the contents of the TLB are completely invalidated. This is necessary since virtual-to-physical address mapping is different for different processes. Alternatively, each TLB entry is tagged with a process number, and in this case, only the entries relevant to the running process are considered in the address translation.

As anticipated in Section 1, the MMU is a costly component. In particular, the TLB is characterized by high hardware complexity and high power consumption [4]. In contrast, our design relies on a memory protection unit whose design takes advantage of the fact that the composition of the protection contexts in terms of access rights is fixed and independent of the process. It follows that when a process switch takes place, the contents of the context registers do not vary. No caching mechanism is necessary to load these contents from the primary memory. The processor accesses the context registers directly, to initialize them under control of the operating system. We have obtained this important result by taking advantage of the concept of a protection domain defined in terms of a collection of protection contexts. The domain register allows us to specify different protection domains for different processes. As seen in Section 2, on the occurrence of a process switch, the contents of the domain register are replaced with the configuration for the new process; this fast action activates the protection domain of the new process.

We may conclude that, mainly owing to the absence of any form of caching, the hardware cost of the MPU is low, and is much lower than that of an MMU with the address translation part removed.

6. RELATION TO PREVIOUS WORK

6.1. The access matrix

In a classical model, the protection system takes the form of a matrix, called the *access matrix* (AM), featuring a row for each domain and a column for each object [26]. Element $AM_{i,j}$ in row i and column j of the access matrix specifies the access rights included in domain

d_i for object o_j. A subject is a pair (process, domain), that is, it consists of the execution of a given process in a given domain. When a process switches to a different domain, a new subject is activated. Every access attempt performed by a subject to a given object must be validated, by verifying that the domain of this subject includes the access rights that are necessary for successful execution of the access (*principle of complete mediation* [25]). In this model, domains are also protected objects. The access rights defined for a given domain make it possible to activate that domain and modify its composition, for instance.

Different methodologies have been devised for storage of the access matrix [26]. In the *access control list* approach, an access control list is associated with each object; the list for a given object takes the form of a sequence of pairs (domain identifier, access rights), i.e. the specification of the set of access rights that are included in the given domain. In the *capability list* approach, the composition of a protection domain is specified in terms of a list of capabilities. A capability is a pair (object identifier, access rights), i.e. the specification of a set of access rights for the given object.

6.2. Capabilities and passwords

A critical problem of every capability system is that of capability segregation in memory [5], [22], [35]: we must prevent a process holding a given capability from taking advantage of the processor instructions for ordinary data manipulation to modify the composition of this capability, e.g. tampering with the access right field to obtain an undue amplification of access rights, or modifying the object identifier field causing the capability to reference a different object. In the absence of segregation, it is even possible that a process forges a new capability from scratch.

Several solutions have been devised to the capability segregation problem. In a segmented memory environment, special segments, which we shall call the *capability segments*, can be reserved for capability storage (in contrast, the *data segments* will be reserved for storage of ordinary information items). This approach leads to undue complications in the representation of objects in memory, e.g. at least one capability segment will be necessary for each object, to store the capabilities for the data segments containing the internal representation of this object. The processor instruction set needs to be enlarged to include ad-hoc instructions for capability processing (the *capability instructions*). Alternatively, in a system supporting a form of tagged memory, a tag is associated with each memory cell to specify whether this cell contains a capability or an ordinary information item. In this case, too, the instruction set of the processor will be expanded to include the capability instructions. If one of these instructions is executed

on a cell that is not tagged to contain a capability, an exception of violated protection is raised inside the processor and the memory access is inhibited. Contrary to the requisite of hardware standardization, this approach requires specialized memory banks aimed at storing the cell tags.

Passwords capabilities [1], [22] are an important improvement on the capability concept. In the password capability approach, a set of passwords is associated with each object, and each password corresponds to a set of access rights. A password capability is a pair (object identifier, password); it grants its holder the access rights for the specified object that are associated with the password. If passwords are large and randomly distributed, the probability to guess a valid password to forge a password capability is vanishingly low. It follows that password capabilities can be stored in undifferentiated memory cells, and can be mixed in memory with ordinary information items. As such, they are an effective solution to the segregation problem.

Our protection system embodies a variant of the password capability approach, in which the protected objects are the memory pages and the protection domains, and a capability takes the form of a password. Each password is associated with a domain, and the access rights for the memory pages corresponding to the given password are those included in this domain.

6.3. Derivation

In capability systems, the instruction set usually includes one or more instructions supporting forms of capability derivation in the direction of weaker access rights, i.e. to reduce the extent a given capability by eliminating part of the access rights it contains. The access right field of a capability can be effectively codified by reserving a bit for each access right; this bit is set if the access right is present. In this case, capability derivation is simply obtained by clearing the bits corresponding to the access rights to be eliminated.

In contrast, in a password capability environment, derivation is a complex action that implies a password replacement; the new password will correspond to the reduced set of access rights. To this aim, a software component is necessary, that we shall call the *password manager*. A process aimed at reducing a given password capability will transfer this password capability to the password manager. In turn, the password manager will return a new password capability expressed in terms of the weaker password.

As seen in Section 4.3, in our system processes can produce the effect of a password derivation autonomously, by issuing the $derivePassword()$ primitive that takes advantage of the composition of the password chains. No form of password manager is necessary. Significant advantages follow in terms of effectiveness and simplicity in object implementation.

6.4. Access privilege revocation

Different solutions have been proposed to the problem of access privilege revocation in capability-based protection environments. For instance, in [31], forms of selective revocation are obtained by taking advantage of access indirection. To this aim, a reference monitor is interposed between the holder of an access privilege for a given protected resource and the resource. The resource owner controls revocation by interacting with the reference monitor. In [17], a form of automatic, global revocation is proposed that takes advantage of short capability lifetimes to enforce capability expiration. Non-revoked capability renewal is supported. In [10], a capability propagation graph is constructed by recording the propagation of a capability from subject to subject, as the system runs. To exercise revocation, the graph is inspected to find the identity of all subjects that hold capability copies. All these solutions are prone to affect performance in capability management negatively. They tend to subvert the main advantage of the capability protection paradigm, i.e. simplicity of access right transmission between processes, which was among the original motivations for the introduction of the concept of a capability [19].

In the password capability model, access right revocation can be effectively obtained by replacing the value of a password with a new value. This action invalidates all the password capabilities expressed in terms of the replaced password. Furthermore, a form of partial revocation can be obtained by reducing the extent of a given password in terms of the access rights associated with that password.

As seen in Section 5.2, in our system two different mechanisms support access privilege revocation. If we change the one-way function used to construct the password chain of a given process, we invalidate all the passwords in that password chain except the master password. In an implementation taking advantage of a parametric one-way function, the one-way function can be replaced by simply changing the value of the parameter. This action will be followed by a new evaluation of the passwords in the password chain; the new passwords will be inserted into the password table. Furthermore, we can modify the composition of the domain associated with a given password by eliminating protection contexts from this domain; a result of this type will be obtained by issuing the $revoke()$ protection primitive. As seen in Section 4.2, the cost of an action of this type in terms of execution times is extremely low, as it only implies an access to the password table to modify the bit configuration expressing the domain to be reduced.

6.5. Ad-hoc hardware for memory protection

Ad-hoc hardware solutions to enforce forms of mandatory isolation of individual software activities in embedded systems have been the object of a few studies in the past. In [15], a memory map checker is proposed, designed to be logically interposed between the processor and the memory modules. The address space of the microcontroller is partitioned into fixed-size blocks, which are grouped to form segments allocated to protection domains, statically at compile time or dynamically through a memory heap. The memory map checker captures the memory write signals from the processor, and actually accesses the main memory only if the write address is valid. No mechanism prevents a software module from corrupting its own state, as each module resides completely within the boundaries of a single protection domain.

In [8], attacks to manipulate the flow of control are considered with special reference to embedded systems. A hardware solution is presented that supports a division of the stack into a data stack and a control flow stack. The control flow stack is aimed at storing the return addresses; this stack is hosted in a memory module at a location different from that of the data stack. Hardware mechanisms prevent accidental or deliberately harmful modifications of the contents of the control flow stack. In particular, access to this stack is restricted to the call and return instructions; as a result, return addresses on the stack are protected from being overwritten with arbitrary data.

In [33], it is assumed that in a low power microcontroller the overall application memory layout is simple and quite static, suggesting a trade-off between hardware cost and usability to fix the maximum number of individual software activities. A protection scheme is proposed that takes advantage of a memory protection unit implementing a segmented view of the address space. The MPU acts on the enable signal of the memory and input-output devices to block a data transfer physically, if an access violation is detected. The rationale for segmentation is to permit a flexible memory layout, e.g., in a memory-mapped view of input-output devices, to protect the small memory areas corresponding to the internal registers of these devices. However, complexity of the MPU hardware is increased, in particular, by the necessity to maintain an associative segment lookup table for storage of the segment descriptors.

7. CONCLUDING REMARKS

With reference to an embedded system featuring no support for memory management, we have presented a model of a protection system based on passwords. Our model takes advantage of a memory protection unit interposed between the processor and the complex of the main memory and the input-output devices. The MPU supports both concepts of a protection context

and a protection domain. A protection context is a set of access rights for the memory pages; a protection domain is a set of one or more protection contexts. In contrast to the classical approach that associates passwords with protected objects, we associate passwords with protection domains. A process that holds a given password can take advantage of this password to activate the corresponding domain. A simple set of protection primitives makes it possible to modify the composition of the domains in a strictly controlled fashion. Passwords are organized in one-way chains called password chains, and associated with processes.

We have obtained the following results:

- The proposed model is compatible with a typical constraint of embedded systems, that the hardware cost of the MPU must be low, e.g. much lower than that of a traditional memory management unit with the memory translation hardware removed. In our proposal, the low complexity of the MPU hardware is mainly a consequence of the fact that the composition of the protection contexts in terms of access rights is fixed for all processes. When a process switch takes places, only the domain register needs to be updated; the contents of all the other MPU registers are left unaltered, and no caching mechanism is necessary to restore the protection state of the new process.
- The organization of the passwords into password chains makes password derivation possible, whereby a software component that possesses a password in the active password chain can derive and distribute the subsequent passwords. No intervention of a form of password manager is necessary for password derivation.
- An effective form of password revocation is supported. By changing the one-way function used to generate the active password chain, we change all the passwords in this password chain except the master password. Consequently, the old passwords are invalidated. This password revocation mechanism results to be transitive, temporal, deferred and independent.
- Trade-offs are possible between the memory requirements for storage of passwords and the password table, and the time necessary for password validation. If the password table is split into several tables, one table for each password chain, and each password includes the name of the process and the position of the corresponding password in the password chain, a single password comparison is sufficient for password validation. In a password table, the space necessary for password storage can be reduced to a single password, i.e. the master password, but in this case, we have to re-evaluate several passwords at each action of password validation.

ACKNOWLEDGMENT

This work has been partially supported by the TENACE PRIN Project (Grant no. 20103P34XC_008) funded by the Italian Ministry of Education, University and Research.

REFERENCES

[1] M. Anderson, R. D. Pose, C. S. Wallace, "A password-capability system," *The Computer Journal*, vol. 29, no. 1 (1986), pp. 1–8.

[2] S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk, *Cryptographic Hash Functions: A Survey*. Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Wollongong, Australia, 1995.

[3] J. S. Chase, H. M. Levy, M. J. Feeley, E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 4 (1994), pp. 271–307.

[4] J.-H. Choi et al., "A low power TLB structure for embedded systems," *Computer Architecture Letters*, vol. 1, no. 1 (2002).

[5] M. de Vivo, G. O. de Vivo, L. Gonzalez, "A brief essay on capabilities," *SIGPLAN Notices*, vol. 30, no. 7 (July 1995), pp. 29–36.

[6] A. Dunkels, B. Grönvall, T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004, pp. 455–462.

[7] B. Egger, S. Kim, C. Jang, J. Lee, S. L. Min, H. Shin, "Scratchpad memory management techniques for code in embedded systems without an MMU," *IEEE Transactions on Computers*, vol. 59, no. 8 (August 2010), pp. 1047–1062.

[8] A. Francillon, D. Perito, C. Castelluccia, "Defending embedded systems against control flow attacks," *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, Chicago, Illinois, USA, November 2009, pp. 19–26.

[9] B. Furht, V. Milutinovic, "A survey of microprocessor architectures for memory management," *Computer*, vol. 20, no. 3 (March 1987), pp. 48–67.

[10] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6 (November 1979), pp. 575–586.

[11] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke, "The Mungi single-address-space operating system," *Software — Practice and Experience*, vol. 28, no. 9 (July 1998), pp. 901–928.

[12] N. Ho, A. V. Dinh-Duc, "MemMON: run-time off-chip detection for memory access violation in embedded systems," *Proceedings of the 2010 Symposium on Information and Communication Technology*, Hanoi, Vietnam, August 2010, pp. 114–121.

[13] Y.-C. Hu, M. Jakobsson, A. Perrig, "Efficient constructions for one-way hash chains," *Proceedings of the Third International Conference on Applied Cryptography and Network Security*, New York, NY, USA, June 2005; in: *Lecture Notes in Computer Sciences*, vol. 3531, Berlin, Heidelberg: Springer-Verlag, 2005.

[14] B. Jacob, T. Mudge, "Virtual memory: issues of implementation," *Computer*, vol. 31, no. 6 (June 1998), pp. 33–43.

[15] R. Kumar, A. Singhania, A. Castner, E. Kohler, M. Srivastava, "A system for coarse grained memory protection in tiny embedded processors," *Proceedings of the 44th Annual Conference on Design Automation*, San Diego, California, USA, June 2007, pp. 218–223.

[16] L. Lamport, "Password authentication with insecure communication," *Communications of the ACM*, vol. 24, no. 11 (November 1981), pp. 770–772.

[17] A. W. Leung, E. L. Miller, S. Jones, "Scalable security for petascale parallel file systems," *Proceedings of the ACM/IEEE Conference on Supercomputing*, Reno, Nevada, November 2007.

[18] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, D. Culler, "TinyOS: an operating system for wireless sensor networks," in: *Ambient Intelligence*. New York: Springer-Verlag, 2005, pp. 115–148.

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

[20] L. Lopriore, "Protection structures in multithreaded systems," *The Computer Journal*, vol. 56, no. 4 (April 2013), pp. 478–496.

[21] L. Lopriore, "Hardware support for memory protection in sensor nodes," *Microprocessors and Microsystems*, vol. 38, no. 3 (May 2014), pp. 226–232.

[22] L. Lopriore, "Password capabilities revisited," *The Computer Journal*, vol. 58, no. 4 (April 2015), pp. 782–791

[23] D. S. Miller, D. B. White, A. C. Skousen, R. Tcherepov, "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 2006.

[24] R. Pose, "Password-capabilities: their evolution from the Password-Capability System into Walnut and beyond," *Proceedings of the Sixth Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January 2001, pp. 105–113.

[25] J. H. Saltzer, M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9 (September 1975), pp. 1278–1308.

[26] P. Samarati, S. de Capitani di Vimercati, "Access control: policies, models, and mechanisms," in: R. Focardi, R. Gorrieri (Eds.), *Foundations of Security Analysis and Design*. Berlin, Heidelberg: Springer, 2001, pp. 137–196.

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

[28] F. B. Schneider, "Least privilege and more," *IEEE Security & Privacy*, vol. 1, no. 5 (September–October 2003), pp. 55–59.

[29] M. D. Schroeder, J. H. Saltzer, "A hardware architecture for implementing protection rings," *Communications of the ACM*, vol. 15, no. 3 (March 1972), pp. 157–170.

[30] M. Simpson, B. Middha, R. Barua, "Segment protection for embedded systems using run-time checks," *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, San Francisco, California, USA, September 2005, pp. 66–77.

[31] J. S. Shapiro, J. M. Smith, D. J. Farber, "EROS: a fast capability system," *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, USA, December 1999; in: *Operating Systems Review*, vol. 34, no. 5 (December 1999), pp. 170–185.

[32] O. Stecklina, P. Langendörfer, H. Menzel, "Towards a secure address space separation for low power sensor nodes," *Proceedings of the 1st International Conference on Pervasive and Embedded Computing and Communication Systems*, Algarve, Portugal, March 2011.

[33] O. Stecklina, P. Langendörfer, H. Menzel, "Design of a tailor-made memory protection unit for low power microcontrollers," *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems*, Porto, Portugal, June 2013.

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

[35] M. V. Wilkes, "Hardware support for memory protection: capability implementations," *ACM SIGARCH Computer Architecture News*, vol. 10, no. 2 (March 1982), pp. 107–116.

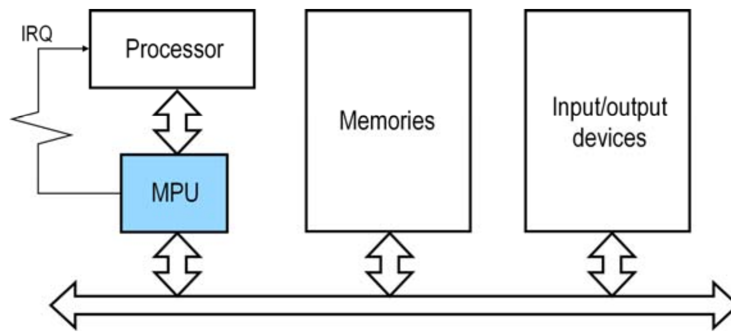

Figure 1. Hardware configuration featuring a memory protection unit interposed between the processor and the complex of the memory and input/output devices.

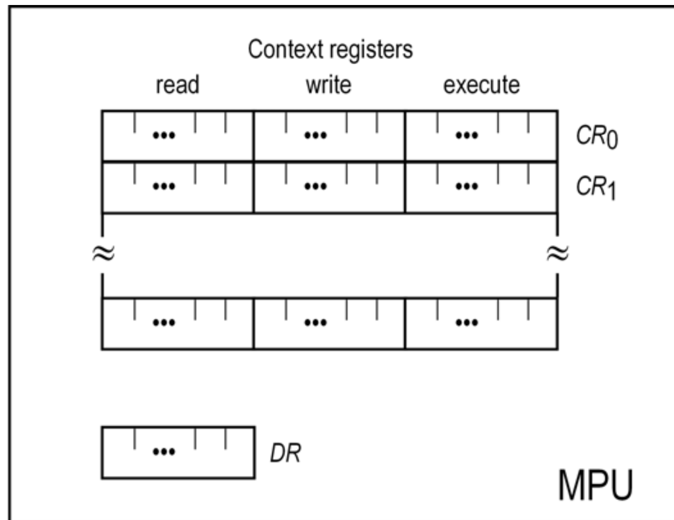

Figure 2. Memory protection unit featuring a set of context registers $CR_0, CR_i, \ldots,$ one register for each memory page, and a domain register DR that determines the composition of the active domain in terms of the contents of the context registers.

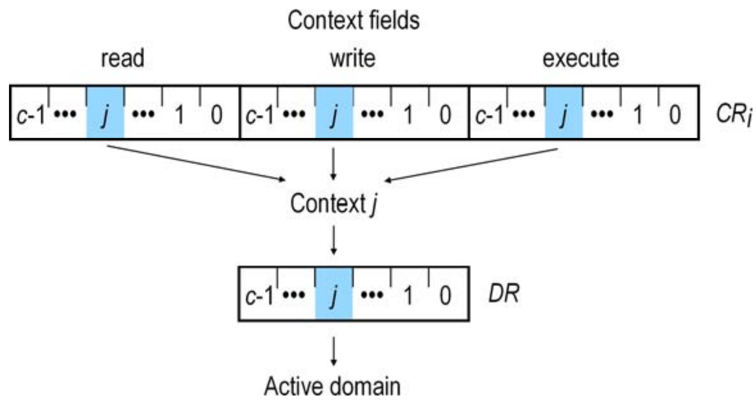

Figure 3. Configuration of context register CR_i associated with page P_i. The bits in the j-th position of the three context fields together identify the access rights for P_i included in the j-th protection context. Domain register DR selects the contexts that form the active domain.

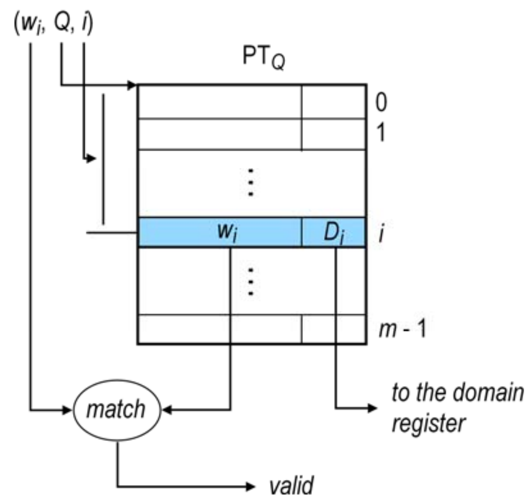

Figure 4. Password validation. Each password is stored in the form of a triple (w_i, Q, i), where Q indicates a process, and i identifies the i-th password in the password chain of Q, which is stored in the i-th entry of password table PT_Q.

Table I. Protection primitives.

$activate(w)$
Activates the domain associated with password w.

$grant(w_0, i, msk)$
In the active password chain, adds each protection context specified by msk to the domain associated with the i-th password, provided that this context is part of the domain associated with master password w_0.

$revoke(w_0, i, msk)$
In the active password chain, removes each protection context specified by msk from the domain associated with the i-th password, provided that this context is part of the domain associated with master password w_0.

$w_{i+j} \leftarrow derivePassword(w_i, j)$
Returns the j-th password following password w_i in the active password chain.