An Architecture of Security Management Unit for Safe Hosting of Multiple Agents

Tanguy Gilmont(1), Jean-Didier Legat, Jean-Jacques Quisquater
Microelectronics Laboratory, Université Catholique de Louvain
Place du Levant 3, B-1348 Louvain-la-Neuve, Belgium

ABSTRACT

In such growing areas as remote applications in large public networks, electronic commerce, digital signature, intellectual property and copyright protection, and even operating system extensibility, the hardware security level offered by existing processors is insufficient. They lack protection mechanisms that prevent the user from tampering critical data owned by those applications. Some devices make exception, but have not enough processing power nor enough memory to stand up to such applications (e.g. smart cards).
This paper proposes an architecture of secure processor, in which the classical memory management unit is extended into a new security management unit. It allows ciphered code execution and ciphered data processing. An internal permanent memory can store cipher keys and critical data for several client agents simultaneously. The ordinary supervisor privilege scheme is replaced by a privilege inheritance mechanism that is more suited to operating system extensibility. The result is a secure processor that has hardware support for extensible multitask operating systems, and can be used for both general applications and critical applications needing strong protection.
The security management unit and the internal permanent memory can be added to an existing CPU core without loss of performance, and do not require it to be modified.

Keywords: ciphered software, software protection, secure operating system, itinerant agent, crypto processor

1. INTRODUCTION

New applications in large public networks (like the Internet) need more security. In multimedia applications, programs that we will call ***agents*** are dynamically downloaded into a system at run-time to allow the processing of new objects (example: *plug-in* capabilities of browsers). This extensibility feature raises questions about the integrity of the system, the licensing of the agent, the protection and privacy of the objects manipulated by the agent:

- the integrity of the system depends on the privilege given to the agent code. It is not always possible to trust programs obtained from sources such as a public network. The downloaded code may contain a virus (this does not only involve the trust of the original source, but also the reliability of the network : the code can be modified during the transport), or a Trojan horse[16]. More likely, the code may contain errors and corrupt the system data. To avoid integrity hazard, the privilege of the agent code should be restricted to the necessary domain;
- the licensing of the agent concerns the protection of the code. The license limits the number of executions or the amount of execution time, the list of nodes where the program can be executed, and the list of services given by the program. The user should not be able to remove those limitations. Sometimes, reverse-engineering of the code should also be prevented;
- the protection and privacy of the objects is a much more vague concept, as it depends on the nature of the object and the agent. In the case of *applets*, the author may want to keep the script secret, for example. The agent should then only provide the result of its execution, and the system should not be able to access the script source.

These security issues are also true for other applications like electronic commerce[12,25] and database servers supporting itinerant agents or *aglets*[29,30,31] from external clients. For example, in the case of electronic commerce, the object to protect is the electronic currency stored in the system. From the agent's point of view, the user should not be able to credit his account illegally, without using the agent protocol. From the user's point of view, his account should not be lost or tampered with by other applications running on the system.
While a part of the protection can be done by software, the operating system (OS) needs sufficient hardware mechanisms to guarantee the system security. While existing processors found on workstations and database servers have adequate hardware to offer basic protection in multitasking environment, they lack mechanisms which prevent the user(2) from tampering the software or its data. That means that it is possible to ensure the system integrity, although it may be somewhat difficult to

[1] e-mail: gilmont@dice.ucl.ac.be; phone: (32)(10)478062; fax: (32)(10)472598

[2] by "user", we mean any skilled person who has physical access to the computer, and who can modify the application software, the OS and even parts of the hardware to achieved his malevolent goal.

build an extensible operating system. But it is impossible to ensure licensing and object protection, since any resolved user can always go in supervisor mode and be granted access to any code and data by the processor.

1.1. Contribution

This paper presents the architecture of a new hardware ***security management unit*** (SMU), which is an evolution of the classical memory management unit (MMU) offering much stronger security mechanisms. Like the MMU, the SMU is added to a CPU core, the result is a secure processor that has hardware support for extensible operating systems, and can be used for both general tasks and critical applications demanding strong protection (from other programs and from the user). Programs that run securely on smart cards can be executed in the secure processor with the same level of security, and profit by more memory and more processing power.

The SMU allows the processor to run ciphered code and manipulate ciphered data. Its internal non-volatile memory (NVM) can be used to store cipher keys and critical data (like confidential information and electronic currency) for several client agents simultaneously. The ordinary supervisor privilege scheme is replaced by a privilege inheritance policy that is more suited to operating system extensibility. The protection mechanisms are transparent to the client agent programs, i.e. they require no additional code.

The SMU only allows the owner task to access its data in the NVM. It is impossible for the user to get any information illegally, nor to modify the content of the NVM, even by altering the operating system. Since the user has no direct access to the cipher keys, he cannot modify the client agents or reverse-engineer the code of the applications.

Although the CPU should have some specific registers and instructions to control the SMU, it is possible to add the SMU to an existing CPU core and to control it with a dedicated interface without significant loss of performance. For example, some CPU cores (like the ARM[20]) have a coprocessor interface which can be used to control the SMU.

Compared to a MMU, the access control and the privilege inheritance scheme have no performance impact. The ciphered code and data are processed by a symmetrical deciphering unit before being stored in the processor level-1 internal cache. The delay introduced by this device is attenuated by two factors:

- the normal programs have low cache miss rates : the code and data are often fetched from the cache, where they are stored in plain form. The SMU will slightly increase the miss penalty time;
- symmetrical cipher functions (e.g. the Data Encryption Algorithm, DEA) are fast and can be pipelined.

In both aspects, the deciphering unit delay is not unlike the external memory delay problem that fast processors solve with caching. In our case, we cannot take profit of an external[3] level-2 cache, because the plain code and data should not be stored outside the chip for security reasons.

Finally, the die area and complexity of the SMU chip is comparable with an ordinary MMU. However, the surface of the NVM must be taken into account, and the processor chip is larger in this respect.

This paper is structured as follows: subsection 1.2 presents an overview of the related work in software and hardware areas. Section 2 explains the general principle of ciphered code execution, then discuss the security and performance considerations. Section 3 details how the software is secured by the SMU, it is divided in four parts : the presentation of the software structure and the NVM manger (subsections 3.1), how the SMU manages the privileges (subsection 3.2), and the installation procedure of secured software (subsections 3.3 and 3.4). The last subsection (3.5) deals with some potential attacks against the SMU and shows how they are countered. Section 4 gives some preliminary results. Section 5 concludes this paper and is followed by the references.

1.2. Related Work

Several projects are studying extensible systems : SPIN (domain and type enforcement extension, access control mechanisms for extensible systems)[1,2], VINO (extensible systems assembled from reusable components, application-driven policy), Exokernel (application-level management of physical resources)[3], Hydra (extension with user-defined access rights) and Mach (moving functionality out of the kernel for ease of modification).

Juice, Java, and Kimera are technologies for distributing executable components across networks. They enforce the security by interpreting the executable, or by analyzing the source before compilation, but they don’t provide secret code and data protection.

The OS are characterized by their security policy. Related work can be found in[5,6,7,8,9,10,11].

Work in the hardware security mechanism is rudimentary. The only domain thoroughly examined is smart cards. These devices are standard processor architectures with specific coprocessors, confined into a physically secured base. A non volatile memory allows the permanent storage of data like electronic currency, confidential information, and other data. They only communicate via a reduced number of pins using a standard protocol, so the user is not allowed to by-pass the operating

[3] some chips, like the Intel Pentium Pro, do have an internal level-2 cache, but the manufacturing costs of the MCM technology are prohibitive in this case.

system and cannot tamper with the internal components[12]. MULTOS is an example of OS running on smart cards and supporting multiple agents. These devices are limited to small tasks as they have not enough resources for managing a multitask environment with database servers and multimedia applications.

Citadel is a physically secure workstation coprocessor that includes a processor, memory and specialized hardware to perform high speed DES processing[26,27]. The IBM 4758 PCI Cryptographic Coprocessor is a PCI-bus card with an Intel 80486 processor controlling a DES chip and a RSA chip, and memory[24]. Both are coprocessor that can be "plugged" into a workstation to allow fast cryptographic processing and key storage. They are complete, almost stand-alone systems, not single chips, and are external devices : they are not running all the applications, thus the secured applications running on those coprocessor and the general programs executed by the main processor cannot be mixed. Also, the security level of those devices are lower than a single chip as they are more vulnerable to physical attacks[28].

The Dallas Semiconductor DS5002FP Secure Microprocessor Chip, which improves some security holes of its predecessor the DS5000FP, is at the moment the closer devices related to our SMU architecture. This chip can load and execute software that is stored in encrypted form, at encrypted addresses. Our architecture offers several advantages in comparison:

- the DS5002FP executes code compatible with the 8-bits 8051 processor, and it can only address 8 banks of 128 Kbytes : this processor has not enough processing power for the kind of application we are examining. It is aimed at small, stand-alone applications like TV decoder boxes or smart card prototypes. Our SMU architecture is suitable to multiscalar RISC, VLIW or DSP processor cores;
- the DS5002FP has no basic support for OS, and can only execute one program. Our architecture has full support for extensible OS, and can execute several secured (encrypted) programs simultaneously in a multitask environment.

2. CIPHERED CODE EXECUTION

Copyright protection and software licensing are usually done in software, and are grounded on the assumption that most users have no sufficient knowledge of the system to break it[28]. While this is true on average, one skilled and malevolent user can use common debugging tools and trace the execution down to the part of code responsible of the protection and remove it. If all users were isolated from each other, the loss would not be worth further attention for widespread software. Unfortunately this is not the case and many users can connect to Bulletin Board Services (BBS) or to archive servers on global networks like the Internet. If only one user is able to remove (*to crack*) a specific software protection, he can easily build a small program that does the removal automatically, and put it on a public server. Any other user can then get this program, crack the software and distribute it at will.

Another victim of insufficient protection is Intellectual Property (IP). Some classes of software (CAD tools, for instance) require complex algorithms or heuristics that are not publicly known, their success is mainly based on the efficiency of those algorithm implementations. Therefore, it is essential for them to prevent any possibility of reverse-engineering, which cannot be guaranteed as long as the plain code is available.

Finally, if a processor equipped with NVM is to be used for electronic commerce and similar applications, it must have some mechanism that only allows the real owner to access its data stored in the NVM. Any method of authentication used by the owner software can be uncovered by careful examination of the code, so it should be kept secret.

The security management unit solves those problems by allowing the execution of ciphered code. The ciphered software is stored in the external RAM, the instructions are decrypted on the fly without significant loss of performance by the SMU when they are fetched by the processor. The instructions are only in plain form inside the processor. Since the user has no access to the plain code, he cannot modify or reverse-engineer it. We call ***secured task*** or a ***secured agent*** any ciphered task protected from user tampering.

2.1. General Principle

The principle, as illustrated in *Figure 1*, is the following one: the program P is initially ciphered with a symmetrical key K, by blocks of length L. We note $g_{c,K}(P)$ the ciphered program, and $g_{c,k}(P[a..a+L-1])$ the ciphered block at position a, where $g_{c,K}()$ is the cipher function used with the key K; $g_{d,K}()$ is the corresponding deciphering function. In fact, the encryption is salted with the virtual address a' to strengthen security, so we should note $g_{c,K,a'}(P[a..a+L-1])$. However, for the sake of clarity we will keep the simplified notation $g_{c,K}()$.

When the processor fetches an instruction from the ciphered program P, the block containing the instruction is loaded from the external memory, deciphered by the symmetrical deciphering unit and stored in a cache line, inside the processor. The needed instruction is then sent to the prefetch queue of the CPU for decoding and execution.

The user must not directly access the key K, and for licensing purposes, the key should only be valid for one processor. Therefore, the key K is ciphered with the public asymmetrical key of the processor E, which correspond to its private key D.

The private key is stored in the processor and cannot be accessed in any way by the user. It can only be used by the asymmetrical deciphering unit, to obtain the key K. We note $f_D()$ the asymmetrical cipher function used with the key D. The complete procedure to execute a ciphered program is:

1. the ciphered key $f_E(K)$, given with the ciphered program $g_{c,K}(P)$, is loaded into the asymmetrical deciphering unit, which deciphers it and gives the result $f_D(f_E(K)) = K$ to the symmetrical deciphering unit;
2. the processor puts the virtual address *a'* of the instruction on the internal bus. The MMU translates the virtual address *a'* into the physical address *a* and outputs it to the external RAM with a read request for the whole block containing the instruction (addresses *[a..a+L-1]*);
3. the read block $g_{c,K}(P[a..a+L-1])$ is deciphered by the symmetrical deciphering unit, using the key *K*, which yields $g_{d,K}(g_{c,K}(P[a..a+L-1])) = P[a..a+L-1]$. The plain block is stored in the internal cache;
4. the instruction is read from the cache and put into the prefetch queue of the CPU;
5. on following fetches in the same block, the data are directly read from the cache.

This principle, explained for code execution, can also be used for data processing. If a ciphered data is modified, the corresponding cache line must be written in the external memory when the cache is flushed (if the case of a write-back cache policy). The symmetrical deciphering unit has to be bi-directional to cipher the block before storing it to the external memory.

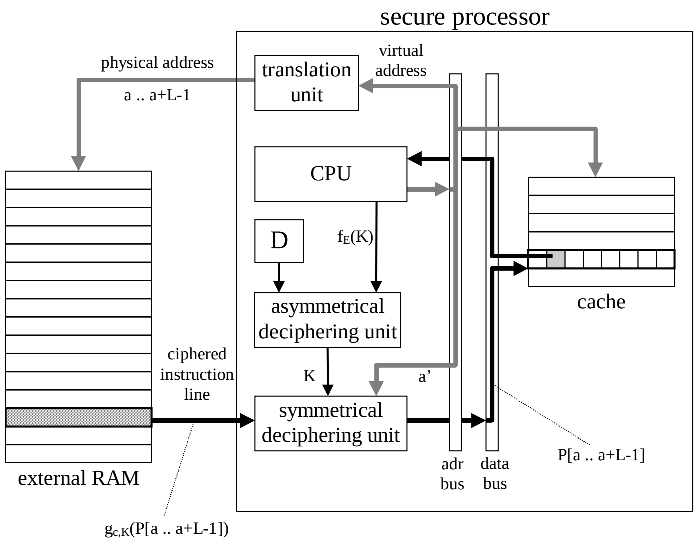

Figure 1 - Instruction block deciphering

2.2. Security and Performance Considerations

If the plain program could be stored in external memory, we could use a pre-process stage in which the processor would decipher the program and write it in the external memory instead of the internal cache. This would allow direct execution of the instructions once the program is decrypted and the run-time performance hit of the cipher function would be less important. But the external memory cannot be considered as safe, since it can be shared among several processors or devices. If the plain program was stored externally, the user could easily fool the processor and copy it.

The processor has to decipher the instructions on the fly, so the deciphering cost must be taken into account. A symmetrical cipher (like the Data Encryption Algorithm, triple-DEA, or IDEA) is fast and well adapted to this application. The asymmetrical ciphers (like the RSA algorithm) are much slower, but the processor only needs this kind of function to get the plain symmetrical key *K*. This can be done each time the program is run, before fetching the first instruction, or only the first time the program is run if the processor can store the key in the internal NVM. In any case, the cost is negligible since only one operation is done before program execution.

Because of the branches in the program code, there must be a way to decipher the program *P* starting from any address. The block nature of the DEA, the triple-DEA or the IDEA algorithms makes them the ideal functions from this point of view, when an instruction cache is present into the processor. The cache line size may be a small multiple (1 or 2) of the size of the blocks processed by the cipher function (typically 64 bits). The processor behaves like any ordinary processor that loads a whole cache line, then fetches the instruction from the cache. The length of the cache line is a trade-off between security and performance: if the line is too short, the encryption mechanism will be easily cracked, and if the line is too long, branches will yield more penalty (deciphering cost) and the cache will be less efficient (less cache misses but more bus traffic for cache misses[22]). For better performance, a pipelined version of the block deciphering unit is preferred. Pipeline hazards due to branches are reduced by using a branch prediction algorithm in the prefetch queue of the deciphering unit. The prediction information can be forwarded to the CPU fetch stage together with the deciphered instruction, to avoid doing the same prediction twice and to keep the prediction scheme consistant.

3. SECURED HOSTING OF CLIENT AGENTS

The ciphered execution principle is the basic mechanism of the SMU for secured client agents. It guarantees that the user will not have access to the plain code of those agents *outside* the chip. In this section, we examine the security mechanisms needed to protect the agent *inside* the chip. If the agent was the only software executed on the processor, it would not be necessary. But we want to allow the execution of *several* agents in a *multitask* environment, which are the typical conditions for network servers (*Figure 2*).

3.1. NVM Manager

In our secure processor, several clients can store their cipher key and other data in the internal NVM, located inside the chip. The NVM access arbitration, too complex to be done in hardware, is accomplished by a small kernel dedicated to the NVM management : the ***NVM manager***. This kernel offers services to the client agents, like deciphering and storing a new key, initializing a new external memory space for ciphered code or data, storing and loading data in the NVM. The NVM manager uses a reserved program identifier (PID), that grants access to the NVM. The SMU verifies that no other program uses this PID, and that each NVM access is done by the NVM manager.

Ideally, the NVM manager program should be stored in ROM, inside the processor chip. This way, it could not be modified by the user, and the PID test would be easy for the SMU : it would only grant NVM access to code executed in that ROM. However, the client agents stored externally must have the same protection level than the NVM manager : they are run as secured tasks. Our approach is to use the same security mechanisms for both, and to allow an external RAM-located NVM manager. The main advantages are the saving of silicon area (for the ROM) and the flexibility. Several NVM managers, responding to different needs, can be used with the same processor, and the code upgrading is easier.

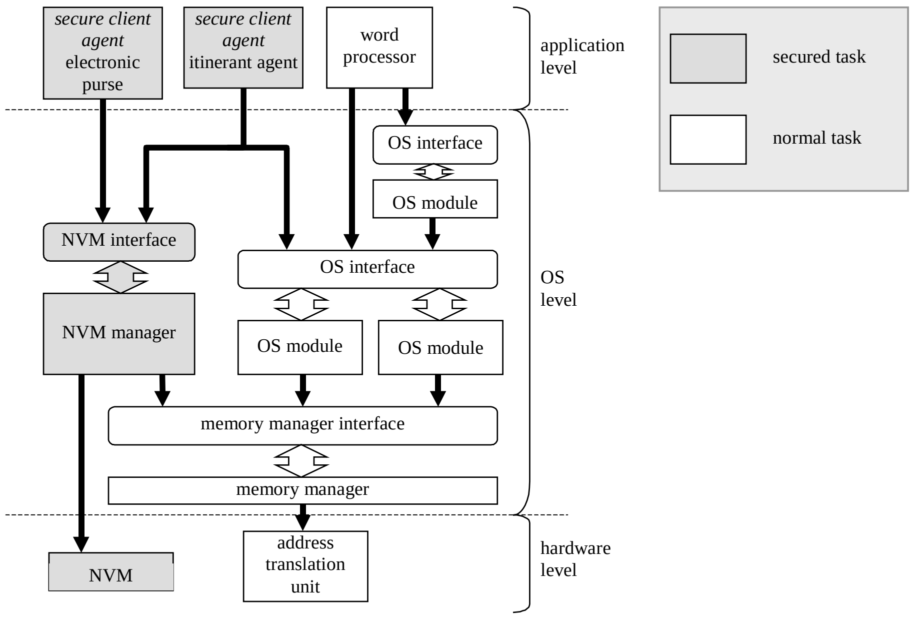

Figure 2 - Software and hardware structure of the secure processor.

3.2. Secured Descriptors

The programs executed on the secure processor access the memory by providing virtual addresses, like any processor allowing basic protection. Each program has its own translation tables containing descriptors used to do the mapping between virtual and physical address space. In a multitask environment, it is common to use segmentation and pagination[18,19] :

- the ***segment*** defines the limits of an addressable (virtual) space and its access attributes. It also defines the type of memory space (text, data or stack, typically), its owner, the linear address base, and other kind of information needed for memory management. One program can use several segments : one for the code (text), one for the stack, and one for the data is the minimum used in a standard configuration. The linear address base of the segment is added to the virtual address given by the instruction, to yield the ***linear address*** ;

- the linear address space of a segment is divided into ***virtual pages*** of fixed length (a power of two, usually 4096 bytes in most architectures, but it can range between 512 bytes and 4 Mbytes, sometimes beyond). Each logical page is translated into a physical page of the same length by the mean of a page table. The benefit is twofold :
    1. the logical pages can be moved anywhere in the available physical space, two successive logical pages being not necessarily successive in physical space : allocating segments and changing their size is easier ;
    2. the unused pages can be stored temporarily on slower mass storage media (hard disk), and the virtual space can be larger than the available physical space.

Each memory reference is handled by the segmentation translation unit, so it is a good candidate to access control. The SMU architecture uses the segmentation for different purposes, each one corresponding to a defined segment descriptor type :

- the ***memory segment descriptor*** defines memory segments, as described before, it is used for memory accesses. There are two segment types: data and code. Usually, load and store instructions address the data segments and branches address the code segments;
- the ***task descriptor*** refers to an entry into the task table, which is used to store the task environment pointers. If an branch instruction has a destination address which refers to a task descriptor, it causes a task switch instead of a normal branch. The pointer of the task table gives the address of the task environment, which contains all the background information needed to execute the task code: values of the registers, current instruction to execute, processor flags, stack pointer and task identity. When the processor switch between tasks, it stores the environment of the old task, then loads the environment of the new task and begins its execution. The remaining part of the virtual address (virtual page and offset) is ignored, only the segment identifier is used in this case;
- the ***call descriptor*** defines an interface which allows other tasks to use services from a task. Branches to call descriptors allow controlled function calls and privilege inheritance between different tasks. In a classical processor, a user program can use services of the OS by executing a trap or a software interrupt. This mechanism changes the privilege of the running task and forces the call to a address fixed by the OS. The OS routine is run in supervisor mode and can access any resource it needs. In our secure processor, a user program executes a ***system call***, which is a branch to a call descriptor. The privileges of the called task are added to those of the caller during the execution of the routine, so it can access its own resource, but also the data of the caller task. Usually, only one call descriptor is used by interface, and the function identifier is given in argument in a register. The remaining part of the virtual address is also ignored with this kind of descriptor.

Other descriptor types are used for exception and interrupt handling, but are not discussed here.

The segment descriptors are used to define the task privileges, as for an ordinary processor MMU. The “user” programs are not allowed to access the segment descriptor tables, nor can they access the page descriptor tables, task descriptors and other information reserved for the memory management. This privilege is only granted to one task, here called the ***memory manager***, belonging to the inner part of the OS kernel (*Figure 2*).

The descriptors of the NVM manager and secured tasks are certified, to forbid user tampering (such descriptors will be called ***secured descriptors***). Indeed, a skilled user could modify the OS kernel and manipulate the descriptors of any secured task. On this basis he could, for example, set the register’s contents and issue a call to a chosen address of the secured task, to gain some access or information he should not have.

Since the descriptor to certify is not very long (typically 64 bits) and we want to protect it from an collision-type attack, we will reject public verification. Another good reason to choose private verification (i.e. with a secret key) is the performance constraint : it will be faster with a symmetrical encryption scheme.

The certificate is built by the NVM manager, by hashing and ciphering the descriptor with the same secret key used to cipher the code (or the date) segment itself. This is also an easy way to identity the key ownership. The index of the key in the NVM key table is included (in clear) in the segment descriptor. When a valid memory access is requested to a secured segment, the symmetrical key is automatically loaded from the table into the deciphering unit. It is not necessary for the OS to call the NVM manager and load the key each time a secured task is to be executed. No task can load the key from the deciphering unit or from the table during this operation, so the secret key is confined into the processor.

The creation of the secured descriptors is done at the initialization phase of the secured software with the help of the NVM manager, since this is the only task that can be trusted, and it is also the only one allowed by the SMU to access the key. The newly created descriptors are then handed on to the memory manager, that puts them into the segment descriptor table. A tampered memory manager cannot modify the descriptors, thanks to the certificate. It can delete them, or give them to another task, but those operations cannot provide the user any advantage :

- if a secured call descriptor is given to the wrong task, it only allows this task to call the secured agent. Since the interface enforce calls to defined entry points, no real attack is possible. The called function must check the validity of its parameters;

- if a secured memory segment descriptor is given to the wrong task, the SMU will detect any attempt to use it, since the task should be ciphered by the same key as the one identified by the descriptor;
- if a secured task descriptor is assigned to the wrong task, the deciphering will yield illegal op-code, for the wrong cipher key is used, and an exception will be triggered by the processor.

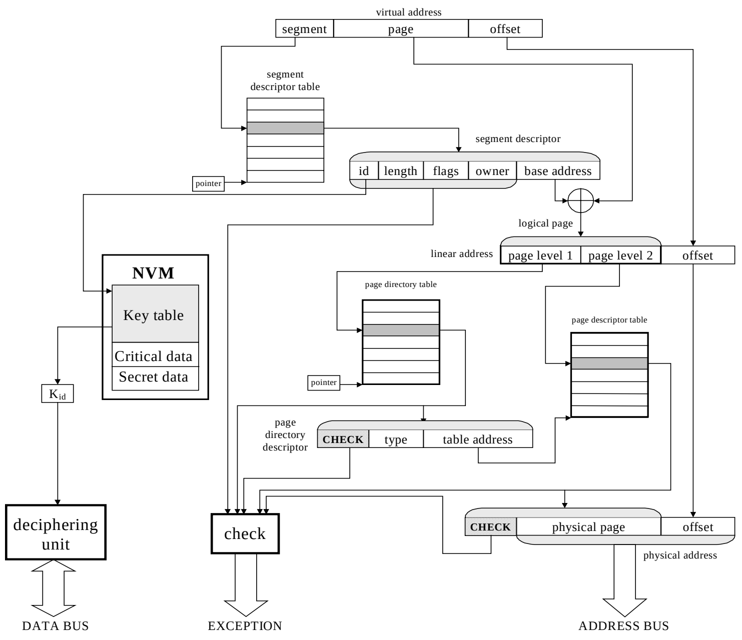

Figure 3 - Secured descriptor principle. The CHECK fields are computed by hashing and ciphering the descriptors with the secret key. If one descriptor is modified, the hash code will be incorrect, and any attempt to use the descriptor will generate an exception. This diagram only shows the functional parts for one type of access, and does not include the speed-up devices used to maintain the performance (TLB, parallel cache lookup and address translation, ...). Other kind of accesses yields variations in the addressing scheme (for instance, page descriptors for system calls are more complex and the offset part is ignored).

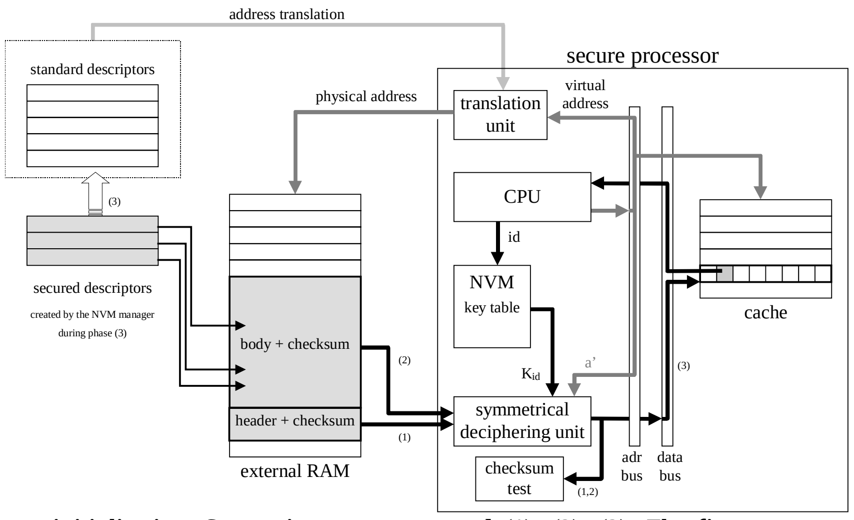

Figure 4 - NVM manager initialization. Successive steps are noted (1), (2), (3). The first two steps are certificate checks. During step 3, the code included in the body is executed and the secured call descriptors are added to the descriptor table.

3.3. NVM Manager Installation

The NVM manager is installed by the operating system. It is first loaded into a memory data segment, as two blocks: the header and the body. Both are ciphered using a secret key. The header contains the body size, the initialization routine address offset, the body certificate and the header certificate. The body contains the manager code and static data.

The operating system is unable to initialize the manager itself, because it cannot decipher the blocks, and cannot create the secured descriptors. The descriptors must be created by the manager itself, so the processor needs a bootstrap instruction to do the initialization. The instruction format is :

```
        INIT adr,id,pid
```

where adr is the block address, id is the key identity number and pid the identity number of the program to initialize - here, the NVM manager. The instruction execution involves the following steps (*Figure 4*) :

1. the header certificate is verified ;
2. the body certificate is verified ;
3. a system call to the initialization address is executed.

During the third step, the program identity is set to *pid* and the program initialization routine, which has now access to the NVM and to the processor keys, creates the needed secured descriptors. These descriptors will then be available to other programs that will have access to the NVM manager services.

3.4. Secured Client Agent Installation

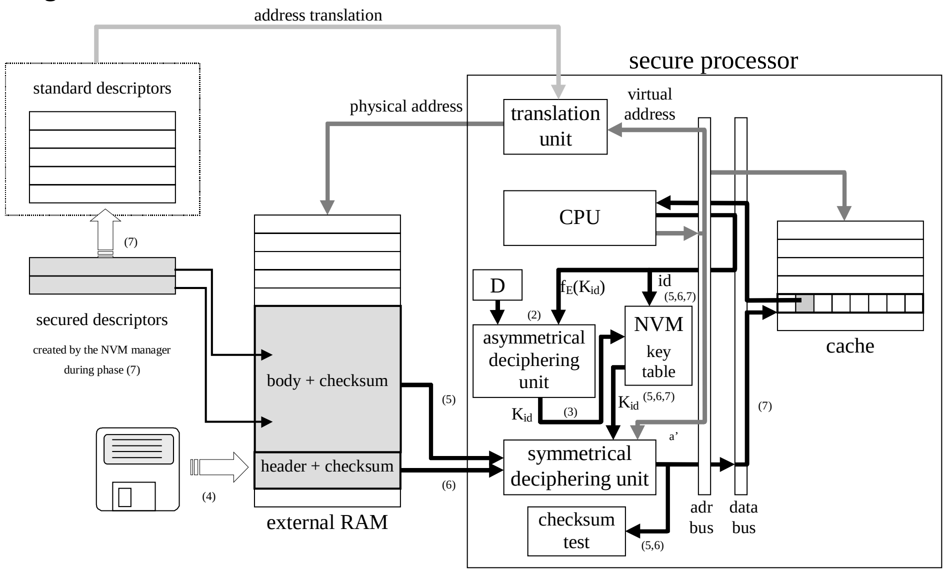

Figure 5 - Client agent initialization. The NVM manager is already installed but its components (code and secured descriptors) are not represented in this figure.

The difference when installing a secured agent is that the NVM manager is already operational. The services of the manager are available to make the agent installation possible.
Each secured agent uses its own secret key, which is used

- to cipher the agent code image, stored on external media and loaded by the OS ;
- to run the ciphered code from memory.

That key has to be stored into the NVM before the agent initialization. Since the agent vendor cannot put the secret key K_{id} safely into the NVM himself, he provides a ciphered secret key $f_E(K_{id})$. The asymmetrical key *E* used is certified by the processor constructor, and is the processor's public key. The corresponding secret key *D* is buried in the processor, and is not accessible by the user. The NVM manager is the only software program to access this key, and it only uses it to decipher the agent's secret key.
The installation of a secured agent is done with the following procedure (*Figure 5*):

1. the very first time, the agent key is not yet in the NVM. The user gets the agent image and the ciphered secret key $f_E(K_{id})$;
2. the OS calls the NVM manager and gives the ciphered secret key. The manager uses the processor secret key *D* associated with the public key *E* the agent vendor has used, to decipher the agent key : $K_{id} = f_D(f_E(K_{id}))$;
3. the manager puts the agent key K_{id} into one free slot of the NVM key table, and the corresponding key identification number *id* is returned to the OS. If no more slot is available, an error code is returned;
4. the OS loads the agent image into memory and executes an INIT instruction, with the agent key *id* and the agent *pid*, which is chosen by the OS ;
5. the image header is verified ;
6. the image body is verified ;

7. the agent initialization routine is called. The agent program can use the manager services to install secured descriptors, so that other normal programs will be able to access the agent's services.

Once the agent key K_{id} is in the NVM key table, only steps 4-7 are necessary to install the secured agent again.

A secured agent must be authenticated before getting access to data stored into the NVM, to make sure it is the real owner. This authentication is done at installation time, when the agent's secret key is used to verify the code image certificate.

When a task requests NVM access with a system call to the NVM manager, the key identifier of the caller's descriptor is checked against the *id* of the NVM table. The NVM is not used for large data storing, and its access by a secured agent is not frequent, so the extra software verification has no performance impact. No additional hardware mechanism is needed to perform those verifications.

3.5. Security Considerations

The normal use of the NVM manager is to provide some services to other programs by the way of system call descriptors. Examples of services the manager provides are :

- allocation of memory into the NVM memory ;
- load and store of critical data and secret keys ;
- release of allocated memory ;
- secured descriptors generation ;
- asymmetrical key calculus.

We are supposing that the NVM manager and the client agents are error-free and contain no security hole, and we are only considering hardware weaknesses. All potential attacks summarize in trying to make the processor execute chosen instructions under the manager identity, to gain an illegitimate access to the NVM contents. Those attacks can be divided into three classes :

1. The **first class of attacks** deals with modification of the NVM manager code. The attacker can modify the manager image stored on external media or loaded in data memory by the OS, when it is not yet installed. It is ciphered with a secret key the user cannot have access to, so that the only possible attack is blind modification of the code. To succeed, the modification must meet the following conditions :
   - it must generate new ciphered blocks that, once deciphered, yield code with correct instructions. If an incorrect instruction is met, an exception is generated and the attack fails ;
   - it must change the code into new instructions that do not generate exceptions by doing illegal operations, like addressing a bad segment, jumping to a bad address, ...
   - it must be useful to the attacker. The more severe condition is that the modification must give the attacker some privilege he should normally not have, like any modification to the secured memory, or the generation of a secured descriptor with a key he does not have ;
   - it must not be detected by the body certificate.

   The chance of succeeding are given by the instruction code format, the cipher method, and the certification method. In practice, the two first are fixed by hardware consideration, and the designer can only influence the protection level by choosing the certification method used.

   The other possible attack before installation is to build an image with a known secret key, and to issue the INIT instruction with the manager reserved *pid*. This attack will fail because the hardware mechanism will only allow the use of the manager secret key with operations related to the manager *pid*.

   Once the manager is installed, its code can still be modified. It is not possible to have a total protection of the manager segment in memory, since it is external to the processor chip. Even if mandatory processor protection could forbid modification to the segment, external access to the memory chip makes it possible by fooling the processor. With such physical attack, the body certificate is useless.
2. The **second class of attacks** is to try to make a system call to a bad address, that is, an address which is not a legal entry point[21]. The trivial case of an illegal call descriptor modification is countered by the descriptor certificate verification.

   The call descriptor uses a segment descriptor reference, so this last one must also be secured. Changing the physical address base of the segment descriptor is equivalent to an offset shift of the descriptor call, and so provides the attacker with the ability to choose the physical call address into the manager code. Another way of shifting the code is to physically move the whole manager segment memory, and to choose the call address, the call descriptor address being constant.

   The solution is to take the offset into account when ciphering the code itself. The deciphering of a moved line will yield a random line, with a high probability for incorrect instruction codes that will trigger an exception. The offset is relative to the beginning of the segment, so there is no need to decipher and encipher the code again when it is installed in memory. The segment address is defined by a secured descriptor, and cannot be modified.

   The attacker can force the physical address location of the manager, so he can make the following attack :

- he installs the manager to a chosen address *adr_1*, then copies the generated secured descriptors (the segment descriptors and call descriptors) ;
- he restarts the processor, and installs the manager to another address adr_2, which is carefully chosen by the attacker. He adds the previously copied descriptors to the table, and uses them to make an illegal system call, since those descriptors are relative to adr_1 instead of adr_2. The difference adr_1-adr_2 is the shift amount necessary for the attack to succeed.

This attack would be easily countered by putting the manager segment data into secured registers, but this protection is not necessary since the attack is basically equivalent to the memory moving attack. The protection mechanisms are the same in both cases.

3. The **third class of attacks** is to try to get some information, or to influence the manager program flow by interrupting it. The attacker can easily make the manager generate an exception around a chosen address, for instance with the help of additional hardware that generates a fake memory fault, or by modifying the manager code content as explained before. When an exception occurs in a standard OS, the processor switches to the appropriate operating system module. This module attempts to determine the problem, so it usually has access to the instruction that generated the exception, the register's content and the stack. If the problem can be solved, the program that generated the exception is continued. Here, using this scheme, the attacker could build his own OS, and read the manager's register content, and even modify them. Obviously, such an operation cannot be allowed for secured programs.

   So, when an exception or an interrupt occurs within a secured program, the environment is saved (to a secured data segment), the otherwise readable data are cleared before entering the OS module, and the environment is restored when returning to the secured program. This means that errors occurring within secured programs cannot be recovered by the OS, but only by the NVM manager.

The physical attacks against the chip are not examined in this article, but the reader should be aware that such attacks are possible (especially for devices like smart cards and home-based TV decoder boxes). In order to make those attacks near impossible, the circuit should contain alarms and its layout should be designed accordingly, but this subject is outside the scope of the present paper.

4. RESULTS

Performance evaluation was achieved by behavioral simulation of a secure processor architecture including an 32-bits ARM-7 core, 8-kB data cache, variable-sized (2, 4, 8 and 16 kB) instruction cache, the SMU and the DEA deciphering unit. The deciphering unit is pipelined and its throughput is one 64-bits block per cycle. It has been simulated with different depth sizes (4, 8 and 16 cycle delay) to show the performance impact of the cipher algorithm complexity.

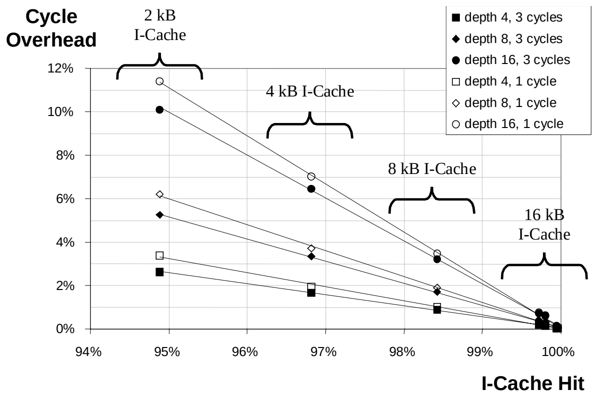

Figure 6 – Deciphering cost

The results given in *Figure 6* show the overhead of the deciphering pipeline in two situations: with an ideal external memory (1 delay cycle and 1 burst cycle, in dotted lines), and with a typical memory (3 delay cycles and 1 burst cycle, in plain lines). The results were averaged on several real programs ported to the ARM architecture (mainly GNU tools). To avoid cache artifacts, and to obtain worse-case results, only the programs with code size significantly bigger than the cache size were chosen.

Individual results (not shown for the sake of clarity) have different overheads, but are located on the same line if represented on a overhead/cache-hit graph, the depth of the deciphering pipeline being constant.

For typical values of instruction cache and pipeline depth (e.g. 8 kB I-Cache, 4 cycles, or 16 kB I-Cache, 8 cycles), the deciphering cost is kept under 1% compared to a classical architecture running the plain program (i.e. not ciphered and not secured).

5. CONCLUSION

The security management unit presented in this paper offers several advantages over the existing devices.

The first advantage is the possibility to run ciphered code and to process ciphered data, with the corresponding keys stored into an internal permanent memory located inside the chip. This allows a stronger protection of intellectual property and software licensing. A software vendor can, with an easy key exchange protocol, give a ciphered form of his program. The software only runs on the right processor, cannot be illegally distributed and the hardware protection is not accessible (and not removable) by the user.

The second advantage is that the processor is able to host several secured client agents, which can manipulate and store critical data safely from any user tampering. It means that the system equipped with this secure processor takes profit of both smart card security and large resources availability (CPU, memory, storage devices). Examples of secured client agents are electronic currency protocols, financial transactions, ciphered database processing, itinerant agents which makes any computer system based on this architecture an ideal complement of the smart card.

Finally, the privilege inheritance mechanism provides a better hardware support for extensible OS design, and makes possible the coexistence of secured tasks and untrusted tasks in the processor environment.

The security management unit and the internal permanent memory can be added to an existing CPU core without significant loss of performance, by the mean of a small interface. Thus, it is not necessary to design a special CPU core for this device, and the security management unit can be adapted to several CPU technologies (RISC, CISC, VLIW, DSP).

6. REFERENCES

1. R.Grimm, B.Bershad, “Access Control in Extensible Systems,” Technical Report, Dept. of Computer Science and Engineering, University of Washington, Seattle, UW-CSE-97-11-01, May 1997.
2. R.Grimm, B.Bershad. Security for Extensible Systems, “The 6th Workshop on Hot Topics in Operating Systems (HotOS-VI),” Cape Cod, Massachusetts, pp. 62-66, May 1997.
3. D.R.Engler, M.F.Kaashoek, J.O’Toole Jr., “Exokernel : an Operating System Architecture for Application-Level Resource Management,” *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, December 1995.
4. S.J. Tanenbaum, R.van Renesse, H.van Staveren, “Amoeba : a Distributed Operating System for the 1990s,” IEEE Computer, pp. 44-53, May 1990.
5. D.Bell, L.Lapadula, ”Secure Computer Systems : Mathematical Foundations,” Technical Report, Mitre Corporation, Vol.1, ESD-TR-73-278, 1973.
6. L.Lapadula, D.Bell. Secure, “Computer Systems : A Mathematical Model,” Technical Report, Mitre Corporation, Vol.2, ESD-TR-73-278, 1973.
7. D.Bell, L.Lapadula, “Secure Computer Systems : Unified Exposition and Multics Interpretation,” Technical Report, Mitre Corporation, ESD-TR-75-306, 1975.
8. K.Biba, “Integrity Considerations for Secure Computer Systems,” Technical Report, Mitre Corporation, MTR-3153, 1975.
9. R.S.Sandhu, “Lattice-Based Access Control Models,” IEEE Computer, Vol.26, No.11, pp. 9-19, Nov. 1993.
10. R.S.Sandhu, P.Samarati, “Access Control : Principles and Practice,” IEEE Communication Magazine, pp. 40-48, Sep. 1994.
11. R.S.Sandhu, E.J.Coyne, H.L.Feinstein, C.E.Youman, “Role-Based Access Control Models,” IEEE Computer, Vol.29, No.2, pp. 38-47, Feb. 1996.
12. L.C.Guillou, M.Ugon, J.J.Quisquater, “A Standardized Security Device Dedicated to Public Cryptology,” *Contemporary Cryptology : The Science of Information Integrity*, edited by Gustavus J.Simmons, IEEE Press, pp. 561-613, 1992.
13. B.W.Lampson, “A Note on the Confinement Problem,” Communications of the ACM, Vol.10, No.16, pp. 613-615, Oct. 1973.
14. S.Lipner, “A Comment on the Confinement Problem,” Operating System Review, Vol.9, No.5, pp. 192-196, Nov. 1975.
15. E.Amoroso (AT&T Bell Laboratories), *Fundamentals of Computer Security Technology*, Prentice Hall, 1994.
16. X.N.Zhang, “Secure Code Distribution,” IEEE Computer, pp. 76-79, Jun. 1997.
17. A.Pfitzmann, B.Pfitzmann, M.Schunter, M.Waidner, “Trusting Mobile User Devices and Security Modules,” IEEE Computer, pp. 61-68, Feb. 1997.
18. A.Silberschatz, P.B.Galvin, *Operating System Concepts*, Addison-Wesley, 1994.
19. U.Vahalia, *UNIX Internals : the New Frontiers*, Prentice Hall, 1996.
20. S.Furber, *ARM System Architecture*, Addison-Wesley, 1996.
21. R.B.K.Dewar, M.Smosna, *Microprocessors : a Programmer’s View*, Mc Graw Hill, 1990.

22. Hennessy, Patterson, *Computer Architecture : a Quantitative Approach*, Morgan Kaufmann Publishers, 1996.
23. Department of Defense Computer Security Center, “Department of Defense Trusted Computer System Evaluation Criteria,” Departement of Defense Standard DoD, Dec. 1985.
24. IBM, “IBM 4758 PCI Cryptographic Coprocessor General Information Manual,” IBM Documentation, GC31-8608-00, June 1997.
25. B.S.Yee, J.D.Tygar, “Secure Coprocessors in Electronic Commerce Applications,” *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, July 1995.
26. B.S.Yee, “Using Secure Coprocessor,” Ph.D. Thesis, Carnegie Mellon University, 1994.
27. E.Palmer, “An Introduction to Citadel – a Secure Coprocessor for Workstations,” *IFIP SEC’94 Conference*, Curacao, Dutch Antilles, May 1994.
28. R.Anderson, M.Kuhn, “Tamper Resistance – a Cautionary Note,” *Proceedings of the 2nd USENIX Workshop on Electronic Commerce*, Oakland, California, pp. 1-11, Nov. 1996.
29. D.Chess, B.Grosof, C.Harrison, D.Levine, C.Parris, and G.Tsudik, “Itinerant agents for mobile computing,” IEEE Personal Communication Systems, 2(5), pp. 34-49, Oct. 1995.
30. G.Karjoth, D.B.Lange, and M.Oshima, “A security model for aglets,” IEEE Internet Computing, 1(4) pp.68-77, July/August 1997.
31. D.B.Lange, M.Oshima, G.Karjoth, and K.Kosaka, “Aglets: Programming mobile agents in java,” *1st Int'l Conf. on Worldwide Computing and Its Applications '97 (WWCA97),* Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, March 1997.