Enhancing Security in the Memory Management Unit

Tanguy Gilmont, Jean-Didier Legat, Jean-Jacques Quisquater
Microelectronics Laboratory, Université Catholique de Louvain
Place du Levant 3, B-1348 Louvain-la-Neuve, Belgium
Phone: +32(10)472540, Fax: +32(10)472598
E-mail: {gilmont,legat,quisquater}@dice.ucl.ac.be

Abstract

We propose an hardware solution to several security problems that are difficult to solve on classical processor architectures, like licensing, electronic commerce, or software privacy. The memory management unit which provides multitasking and virtual memory support is extended and given a third purpose: to supply strong hardware security support for the software layer. The principle of this enhanced device, that we call a Security Management Unit (or SMU), is based on ciphered program execution and access control. It is composed of a pipelined block ciphering/deciphering unit, an internal permanent memory and logic control, whose interaction is explained in this paper.

1. Introduction

Most difficult security problems encountered in several fields of computer science involve applications that must be protected from the user. The solutions proposed nowadays are far from acceptable, as they all summarize in some complex piece of software protection that any determined and skilled user can analyze and remove, given enough time. As long as the assembly code is available for analyze and modification, one cannot rely on software-based protection.

Examples of such applications are software licensing, know-how protection in commercial software and software privacy for remotely executed programs [29,30,31]. Among other fields for which additional security support is needed are operating system extensibility [1,2,3,4], authentication, subscription to remote services and electronic money storage, electronic commerce [25].

Two other important issues about software protection are compatibility and transparency. A solution, however perfect, is not acceptable if the user has to replace his whole software library: the proposed mechanisms should bring as less modifications as possible in the software layer. However, the operating system kernel will obviously have to be adapted. The same constraint holds for the hardware layer: the proposed device will have to fit into an existing scheme. Finally, the behavior of the security device should be transparent for the applications

1.1. Contribution

The solution we propose in this paper is based on additional hardware security support, inside the processor chip. To add strong protection while keeping compatibility and transparency, we enhance the capabilities of the typical memory management unit. The improved device, that we call the *Security Management Unit* (abbreviated *SMU*), is composed of several parts:

- a pipelined block cipher unit, that allows the execution of ciphered code and the processing of ciphered data;
- an internal permanent memory, to store the secret keys needed by the cipher unit, and confidential or critical data owned by the applications. The access to this memory is controlled so that the user cannot steal the keys, and the applications cannot tamper with each other's data;
- the traditional MMU parts like TLB, segment and page registers, control logic that make up the virtual address translation unit and the multitasking support. The logic is extended to manage the protection of the secured programs and to control the cipher unit and the internal memory access;
- an internal L1 cache interface, to provide suitable performance level when the processor is executing ciphered code. This interface works together with the cipher unit that behaves like a prefetch unit.

1.2. Paper outline

This paper is structured as follows: section 2 explains the cipher unit principle, section 3 details the architecture of the memory management and is divided into 3 parts. The segmentation and pagination are explained in subsection 3.1, the security enforcement is explained in subsection 3.2 and the privilege inheritance policy is

overviewed in subsection 3.3. Simulation results are given in section 4. The paper concludes in section 5, and references are given in section 6.

2. Cipher unit

The principle 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. A symmetrical cipher algorithm has been chosen mainly because of the performance it can achieve with pipelining. The cipher function takes the virtual address as a "salt", to prevent block substitution attacks.

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 cipher 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 (figure 1).

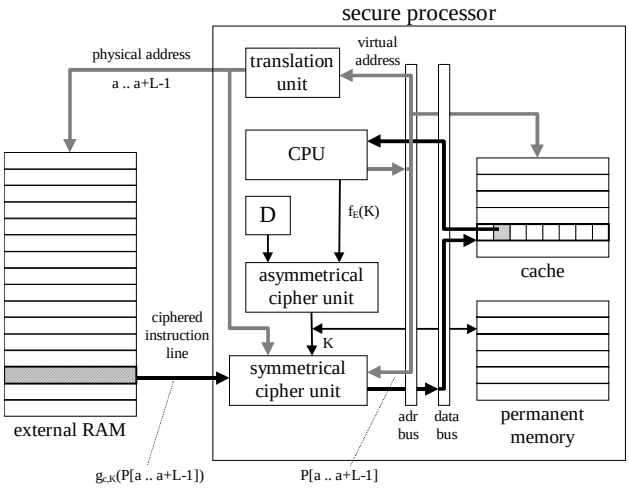

Figure 1 - Instruction block deciphering

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 *E* of the processor, 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 cipher unit, to obtain the key *K*. We note $f_D()$ the asymmetrical cipher function used with the key *D*. Another key pair (D_m, E_m) certifies the origin of the processor. The complete procedure to execute a ciphered program is:

1) the ciphered key $f_E(K)$, given together with the ciphered program $g_{c,K}(P)$, is loaded into the asymmetrical cipher unit, which deciphers it and gives the result $f_D(f_E(K)) = K$ to the symmetrical cipher unit. The key *K* may also be stored into the internal memory for future references. The fact that an asymmetrical cipher algorithm is slower is not important in this case, since the processing has only to be done once, before the first execution of the program;
2) the processor puts the virtual address of the instruction on the internal bus. The translation unit computes the physical address *a* and, if there is no corresponding cache hit, 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 cipher 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 (in the case of a write-back cache policy). The symmetrical cipher unit has to be bi-directional to cipher the block before storing it to the external memory.

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 or triple-DEA algorithms makes them 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, 2 or 4) 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 block size is a trade-off between security and performance: if too short, the encryption mechanism will be easily cracked, and if the block 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]).

The ability of executing ciphered programs, the keys being out of user's reach, has several advantages:

- with the help of a simple key exchange protocol, the user can buy ciphered software and execute it only on his computer. The licensing can be managed in software, since the user cannot read the plain assembly code, he cannot modify the program to remove the protection;
- the program *P* can be ciphered before the user buy the software, and many copies ciphered with the same secret key *K* can be sent to different vendors before being sold. The editor initially chooses one secret key *K*, computes the ciphered program $g_{c,K}(P)$ and writes

it, probably with a plain trial version, onto a large amount of media (like CD-ROM for instance) that are sent to selling points. The end-user willing to buy the full version send the ciphered public key $f_{Dm}(E)$ of his processor to the editor. The editor uses the public key from the manufacturer to get the certified public key E and gives the buyer $f_E(K)$. This way, the editor does not have to cipher each software separately (*Figure 2*);

- no virus can infect the program, and no Trojan Horse can be installed;
- the know-how contained in high-end software cannot be reverse-engineered, since the assembly code (or even any interpreted script) can be made unavailable by ciphering it;
- the user can make any backup copy he needs, which is not true with most of the software protections nowadays.

More elaborate protocols can be implemented on this basis, which protect the privacy of the buyer so that he does not have to give a public key identifying him or his computer.

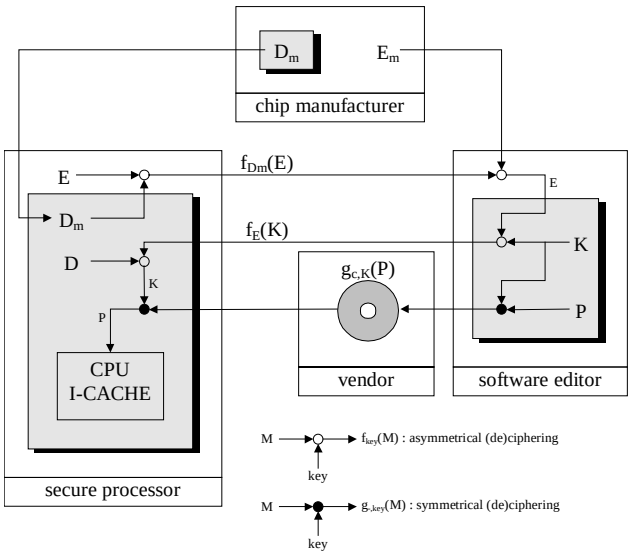

Figure 2 – key management example. Shaded areas are secret.

3. Memory management

In our hypothesis, the malevolent user is skilled and has physical access to the computer, he also is super-user and can configure or modify the operating system to get more privilege than he should have. For example, he can use debugging tools to trace down the program execution. We must also suppose that he has enough electronic equipment to snoop the external buses and signals, or to dump any external memory contents. We will not consider deeper physical attacks [28], however, or attacks based on time and/or electromagnetic observation (like the timing attack and SPA/DPA attacks).

The cipher unit is not sufficient to secure the programs, there are other problems to solve like how to manage the keys and the internal permanent memory and how to control the interaction between secured programs and other software in a multitasking environment. Since the operating system cannot be entirely trusted, the critical security management tasks are performed by a trusted kernel driver we will call the *NVM manager* (*Non-Volatile Memory manager*). Besides, the segmentation and pagination system is extended to provide a strong protection for the programs to secure. This topic is described in the following subsection (3.1 Segmentation and pagination).

The NVM manager offers services to the secured programs, like deciphering and storing a new key, initializing a new external memory space for ciphered code or data, storing and loading data in the internal permanent memory (i.e. 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 from that ROM. However, the ciphered programs stored externally must have the same protection level than the NVM manager : they all are run as secured tasks. So we use the same security mechanisms for both, thus allowing 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 and developed by third parties, can be used with the same processor, and the code upgrading is easier.

3.1. Segmentation and pagination

Our architecture uses a 32-bit virtual address, divided into three parts:

1) the segment register number (the three most significant bits 31-29);
2) the page number (bits 28-12);
3) the page offset (bits 11-0).

Segmentation helps to reduce the information redundancy in the page descriptors. The segment number is given indirectly by the segment register number. The eight segment registers are loaded by the operating system during the initialization and at each task switching. If a task needs more than eight segment references, it can load new segments number into the segment registers, or use a specific prefix instruction. The base address in the segment descriptor is added to the virtual page address to

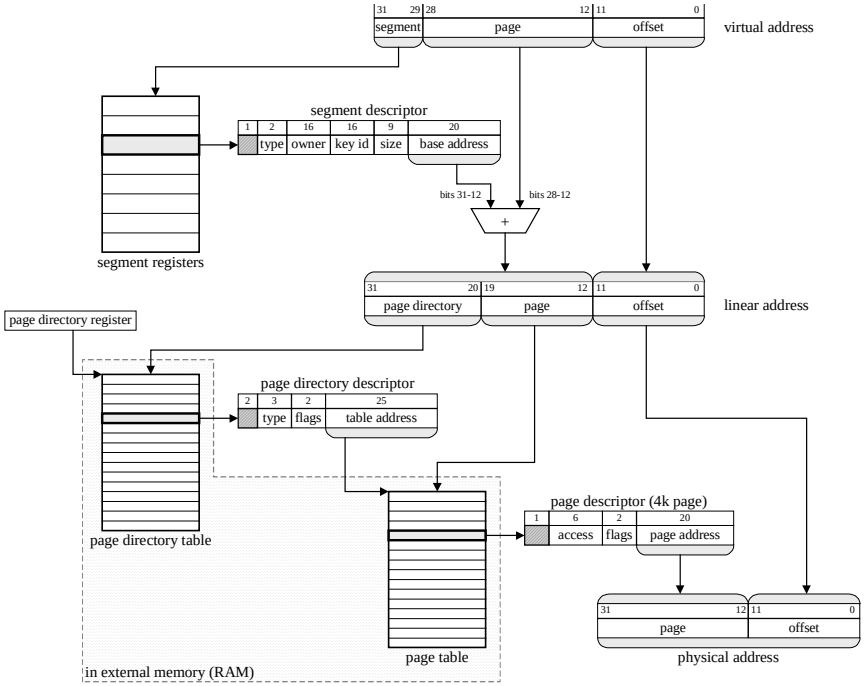

Figure 3 – Virtual address translation

yield the linear address (figure 3). The linear page address is then translated into a physical page address by a 2-level page table scheme. The 2nd level table may contain descriptors for different page sizes (4k, 64k and 1M) to match the different kinds of memory needed by the applications. For 64k and 1M pages, the remaining bits of the linear page address are used as an address offset, so the total offset field may range from 12 to 20 bits, depending on the page size.

To maintain an acceptable performance in spite of the two memory accesses needed to translate the virtual address, we use a typical fully-associative 64-entry translation look-aside buffer. The virtual (pre-MMU) cache and the MMU work in parallel and, provided the page descriptor is in the TLB and the corresponding entry is in the cache, both give their result at the same time, and the cache tags can be compared with the physical address of the page descriptor. To avoid synonyms, the page bits never overlap with the cache line bits: in our architecture, the page size is at least 4k (bits 12 and up) and the cache line is determined by bits 11-3 (bits 2-0 select the byte in the block).

3.2. Security management

The programs executed on the secure processor access the memory by providing virtual addresses, like any processor allowing basic protection [18]. Since each memory reference is processed by the translation unit, it is the ideal place to handle access control and key management. The translation unit processes the virtual addresses by using the following items:

1) *the segment descriptor* is the first item of the translation procedure. It includes the task identifier of the segment owner and its key identifier.

   The classical processors have mainly two running modes: the user and supervisor modes, whereas our architecture use a more flexible, privilege inheritance-based policy that we shall describe later. This policy is the reason of the owner identifier in the segment descriptor: it is used to compute the access rights of a given page. As we shall see, the owner identifier is also used to change the identity of the current task in some operations.

   The key identifier is only used with ciphered code and data pages. When a reference to such a page occurs, the key is automatically fetched from the NVM key table and given to the cipher unit. Thus, any task that reads, writes or jumps to a ciphered memory location does not need to worry about the key management which is transparently handled by the hardware;
2) the second item of the translation is the *page directory descriptor*, obtained from the page directory table with bits 31-20 of the linear address. This descriptor includes the following information:
   – the second-level table address;
   – the type of data it contains: empty table, regular 4k/64k/1M page descriptors, or *trap descriptors*;
   – two flags: the first states the descriptors are certified and the second indicates whether the table is in memory or not;

3) the last item is the *page descriptor*. The regular page descriptors are the same than the one used in standard MMU, they contain:
   - the physical page address, which is merged to the offset to yield the physical address to be put onto the address bus;
   - the access rights (read/write/execute) for the segment owner and for other tasks;
   - one flag to denote the presence of the page in memory, for memory swapping;
   - one flag to signal whether the page is ciphered or not. If this flag is set, instruction and data fetching are done with the help of the cipher unit. The key, whose identifier is given by the segment register, is automatically loaded into the cipher unit key register.

   The trap descriptors will be described in the next subsection (3.3 Privilege inheritance policy). They provide support for task switching, system call with identity change, interrupt and exception handling.

   The page descriptors (regular and trap) can be certified. In this case, they contain a certificate and are wider:128 bits instead of 32 bits. The certificate can only be created by the NVM manager task, it is computed by hashing and ciphering the three descriptors (segment, page directory and page) using the owner key. The certified descriptors are always checked when they are loaded from memory, to prevent descriptor forging attacks.

   Entries of the TLB and cache that correspond to certified and/or ciphered pages are marked with a dedicated flag. When a "ciphered" cache hit occurs, the SMU verifies the ownership of the block before revealing its contents, to prevent cache attacks. In a similar way, certified TLB descriptors are flushed when the task identity changes.

The certified descriptors, or *secured descriptors*, are the second part in the enhanced security support given by the SMU, next to the cipher unit. Together, those two features allow the hosting of secured tasks that even the OS cannot attack. The only influence it can have is to stop the task execution and remove it from memory.

The obvious weak point is the interaction between a secured task and the other tasks. Basically, there are two classes of interaction:

- *between two secured tasks*: for example, an electronic purse driver and a transaction agent, which are coming from two different parties and are running on the same computer to provide some electronic commerce capability. The transaction agent may, at some point, call the electronic purse driver to proceed to a withdrawal. The electronic purse driver needs to check the clearance of the caller (here, the transaction agent), and to verify if it is an authorized client. Typically, the two tasks will engage in a zero-knowledge protocol to certify themselves to each other. There is no security hazard from the OS, since the whole communication can be ciphered and secured. The only event to foresee is a communication cut-off if one task is halted;
- *between a secure task and an unknown task*. The secured task may depend on other tasks, the most common case being OS support. The secure task should be aware that a call to another task may result in a failure (for example, an exception), or a erroneous result, intentional or not. While hardware security support is provided to allow safe hosting of applications, the secured task should of course be designed to recover from other task's errors. Another part of the solution is to certify critical parts of the operating system.

At system initialization, a bootstrap mechanism is provided for the NVM manager installation, since no other task is able to build the certified descriptors of this ciphered driver. The code and data segments are embodied in an image file, which also includes a header and a certificate. The key used to decipher the driver is loaded into the processor chip when it is issued by the manufacturer or by the NVM manager provider. This key is only used for the driver, as it grants access to the internal permanent memory of the SMU.

The bootstrap procedure is started once the image file is loaded into memory. The image certificate is verified, then a system call at the initialization address of the header is executed. The task identity is changed to the owner of the image (i.e. the NVM manager) which can certify its own descriptors and make them available to other tasks [33].

3.3. Privilege inheritance policy

The modern OS concept consists of several module layers, each being confined to one defined purpose and accessing limited resources. Data used within a module should be hidden from other modules and from the user programs if the OS design is clean. In real implementations, some data are shared between several modules for efficiency [18,19,21].

The problem with classical processors is the availability of only one supervisor mode for the whole OS software. The OS modules have no identity and the organization in layers and modules are an abstract concept, since the hardware only supports user and supervisor modes. Once in supervisor mode, an OS module is granted total control of the processor, and has access to all its resources, so no protection mechanism can prevent the whole system corruption if the module does not behave correctly.

This awkward situation makes difficult the implementation of features like OS extensibility. Additional modules, from other sources than the OS vendor, are prone to bring holes into the OS security and stability. The only secure way to allow extensions is to

run them as user programs, and to check their interaction with the OS routines. This control must be done in software for the most part, and is not efficient.

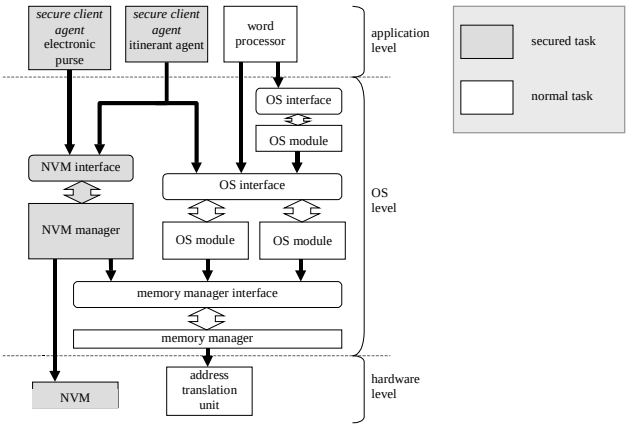

Figure 4 - Software and hardware structure of the secure processor.

In our architecture, the supervisor mode is replaced with *privilege inheritance*, which is based on the concept of minimal privilege grant and task identification [9,10,11]. When a task needs the service of another module, it executes a special call instruction (the *system call*) which is the equivalent of the software interrupt or trap instruction used on classical processors. The system call is a controlled branch to the interface of the module with task identity change. This means that the privileges of the called module are linked with the privileges of the caller task, when the module routine is executed. However, additional constraints exist for secured tasks: the called module will not be able to access ciphered pages of the caller, nor will it be recognized by the NVM manager as the caller task.

The inheritance and task identity are managed by the SMU, with the help of trap descriptors. There are several kinds of trap descriptors:

- the *system call*, as described before, it is very similar to the trap mechanism of standard processors. Special care is taken when restoring identity, to prevent identity substitution attacks of secured tasks;
- the *task switch*: a reference to this descriptor cause the switching of the task context;
- the *slow* and *fast interrupt* are used to treat hardware interrupts. The slow interrupt saves the task context, whereas the fast interrupt is executed under the identity of the interrupted task if authorized by the current policy;
- the *exception* is used to treat software and hardware failures. The invoked routine is given information about the context of the fault and inherits the privileges of the interrupted task if authorized by the current policy. To stop secret information leakage, the context of a secured task is cleared before calling the exception routine, and no inheritance is permitted in this case.

In many cases, it is useful to have access inheritance, especially when the supervisor mode is no longer present. For instance, when a task calls a service of another task, the called task may need access to the caller's data. When such a call occurs, the descriptor table lists of the two tasks are linked and given to the called one during its execution. Classical processors need no similar mechanism, since the called OS service is executed in a more privileged mode, and is automatically granted the caller's access rights. On the other hand, the caller also has access to the segment of the other unrelated user tasks, and this unnecessary privilege makes the system less secure.

4. Results

An architecture based on the 32-bit ARM 7 TDMI core has been developed, which is composed of the CPU core, an Harvard cache of two 8-kByte units (256 lines of 4-way sets, 8-byte blocks) using the near-LRU Deville's replacement algorithm [34], the security management unit logic with the virtual translation unit (segmentation and pagination, 64-entry fully-associative TLB), a 2-kByte permanent memory and an 12-stage triple-DEA cipher unit (4-stage for the simple DEA).

To evaluate the overhead introduced by the code deciphering process, this architecture has been simulated at the system level, for several cache sizes (2, 4, 8 and 16 kB for each cache: instruction and data) and several pipeline depths of the cipher unit (4, 8, 12 and 16 stages). The external memory was characterized by 3-1-1-1 read/write cycles, the internal instruction and data caches had no delay cycles.

The benefit of a fetch prediction unit, which would feed the cipher pipeline, was also studied in those simulations. The motivation of this prediction unit is to reduce the cost of pipeline flushes, due to branch hazards in the code flow. To obtain the lower bound of the improved overhead, we used a perfect prediction scheme (based on the instruction trace).

The worst case figures have been obtained with the SPECint95 *go* benchmark, the results of which are given in figure 5. The graph legend specifies the depth of the pipeline (d_n stands for a depth of n stages) and whether fetch prediction was used or not (p denotes the use of perfect prediction). The deciphering cost can be kept under 2,5 %, with 8 kB I-cache for the deepest pipeline and with 2 kB I-cache for the 4-stage pipeline (12-stage if prediction is used).

When we compare the results obtained without fetch prediction (the plain curves) and those obtained with perfect fetch prediction (the dotted curves), we can

observe that the overhead of the latter are reduced, especially for small cache sizes.

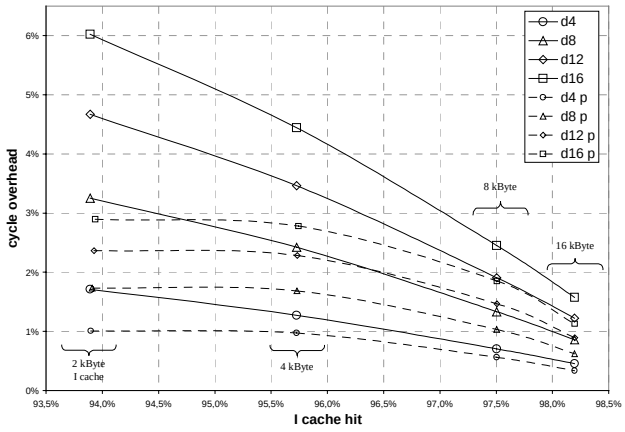

Figure 5 – Deciphering cost

5. Conclusion and future work

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 know-how 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 protection is not accessible (and not removable) by the user.

The second advantage is that the processor is able to host several secured programs, 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), which makes it 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 unstable 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 (under 2,5 % for common cache sizes). Thus, it is not necessary to design a special CPU core for this device, the software library can be preserved and the security management unit can be adapted to several CPU technologies (RISC, CISC, VLIW, DSP).

Future work will investigate existing OS kernel modification to support the SMU, and other processor core adaptation. The fetch prediction unit will be examined in more details, to compare its die area cost versus an instruction cache at the same performance level. Finally, when an AES candidate will be chosen for ASIC design, it will probably replace the older DEA in the architecture, providing a safer key length and the opportunity for 128-bit bus.

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.
32. T.Gilmont, J-D.Legat, J-J.Quisquater, "An Architecture of Security Management Unit for Safe Hosting of Multiple Agents," *International Workshop on Intelligent Communications and Multimedia Terminals (COST254)*, Ljubljana, pp 79-82, Nov. 1998.
33. T. Gilmont, J-D.Legat, J-J.Quisquater, "An Architecture of Security Management Unit,"*Proceedings of SPIE: Security and Watermarking of Multimedia Contents*, San Jose, Vol.3657, pp. 472-483, Jan. 1999.
34. Y.Deville and J.Gobert, "A Class of Replacement Policies for Medium and High-Associativity Structures," Computer Architecture News, Vol.20, No.1, pp. 55-61, 1992.