

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

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

E-mail: gilmont@dice.ucl.ac.be, legat@dice.ucl.ac.be, quisquater@dice.ucl.ac.be

Phone: 32-10478062 Fax: 32-10472598

Abstract

In such growing areas as remote applications in large public networks, intelligent agent support, intellectual property and copyright protection, 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, java cards).

This paper proposes an architecture allowing ciphered code execution and ciphered data processing. An internal permanent memory can store cipher keys and critical data for several client agents simultaneously. 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 deciphering 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.

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 or to provide remote processing to the client side. Examples are *plug-in* capabilities of browsers or intelligent agent querying remote database [12,13,14]. 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 or a Trojan horse [5]. More likely, the code may contain errors and corrupt the system data;
- 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 intelligent agents, the remote server should not access the agent's code or data.

These security issues are also true for other applications like electronic commerce [4,8]. 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 user¹ from tempering the software or its data. That means that it is possible to ensure the system integrity, although it may be somewhat difficult to build an extensible operating system. But it is impossible to ensure licensing and object protection, since any resolved user can always access the code and modify it to gain unauthorized privileges.

1.1 Contribution

This paper presents the architecture of a processor able to run ciphered code and manipulate ciphered data. An internal non-volatile memory (NVM) can be used to store cipher keys and critical data for several client agents simultaneously.

The processor memory management unit (MMU) is modified to ensure strong agent protection. This modified unit, that we will call *security management unit* (SMU), only allows the owner task to access its data in the NVM. It is impossible for the user to get any

¹ 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.

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.

The NVM management and access control is performed by a special agent, the *NVM manager*, which is stored externally in a ciphered form, like any other agent. This agent is authenticated by the SMU with the help of a dedicated cipher key buried into the processor.

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.

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 system and cannot tamper with the internal components [4].

Citadel is a physically secure workstation coprocessor that includes a processor, memory and specialized hardware to perform high speed DES processing [9,10]. 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 [7]. 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 [11].

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;
- the DS5002FP has no basic support for OS, and can only execute one program.

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 [11]. 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. Intelligent agents belong to this category, since they contain private data that is not supposed to be seen by the server they are running on.

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 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 $g_{e,k}(P)$ the ciphered program, and $g_{e,k}(P[a..a+L-1])$ the ciphered block at position a , where $g_{e,k}()$ is the cipher function used with the key K . In fact, the encryption is salted with the virtual address a' to strengthen security, so we should note $g_{e,k,a'}(P[a..a+L-1])$. However, for the sake of clarity we will keep the simplified notation $g_{e,k}()$.

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.

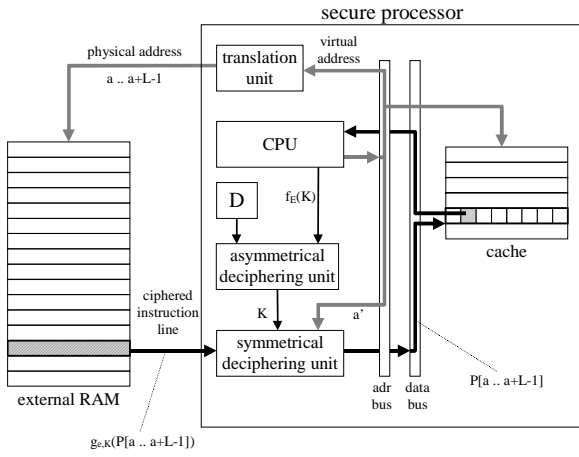


Figure 1 - Instruction block deciphering principle

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 $g_{e,k}(P)$, is loaded into the asymmetrical deciphering unit, which decipheres 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 $g_{e,k}(P[a..a+L-1])$ is deciphered by the symmetrical deciphering unit, using the key K ,

which yields $g_{d,k}(g_{e,k}(P[a..a+L-1])) = P[a..a+L-1]$. 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.

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 is fast and well adapted to this application. The asymmetrical ciphers, like the RSA algorithm for instance, 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 makes it the ideal function 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 [6]).

3 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.

The results given in *Figure 2* 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.

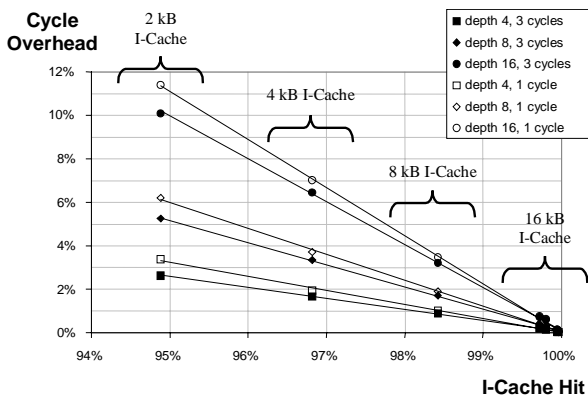


Figure 2 - Deciphering cost

One can see that, 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).

4 Conclusion

The security mechanisms presented in this paper, added to an existing processor core, allow it to execute ciphered programs with very little performance impact. This stands for a stronger protection of intellectual property and software licensing both on the client and the server side. A software vendor can, with an easy key exchange protocol, give a ciphered form of his program. The software only runs on the target processor and cannot be illegally distributed. The hardware protection is not accessible by the user and thus cannot be removed.

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 can take profit of both smart card security level and large resources (CPU, memory, storage devices) available on a mainframe.

5 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] 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.
- [5] X.N.Zhang. Secure Code Distribution. *IEEE Computer*, pp. 76-79, Jun. 1997.
- [6] Hennessy, Patterson. *Computer Architecture : a Quantitative Approach*. Morgan Kaufmann Publishers, 1996.
- [7] IBM. IBM 4758 PCI Cryptographic Coprocessor General Information Manual. IBM Documentation, GC31-8608-00, June 1997.
- [8] B.S.Yee, J.D.Tygar. Secure Coprocessors in Electronic Commerce Applications. *Proceedings of the 1st USENIX Workshop on Electronic Commerce*, July 1995.
- [9] B.S.Yee. Using Secure Coprocessor. Ph.D. Thesis, Carnegie Mellon University, 1994.
- [10] E.Palmer. An Introduction to Citadel – a Secure Coprocessor for Workstations. *IFIP SEC'94 Conference*, Curacao, Dutch Antilles, May 1994.
- [11] 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.
- [12] D.Chess, B.Grosz, 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.
- [13] G.Karjoth, D.B.Lange, and M.Oshima. A security model for aglets. *IEEE Internet Computing*, 1(4) pp.68-77, July/August 1997.
- [14] D.B.Lange, M.Oshima, G.Karjoth, and K.Kosaka. Aglets: Programming mobile agents in java. In T.Masuda, Y.Masunaga, and M.Tsukamoto, editors, *1st Int'l Conf. on Worldwide Computing and Its Applications '97 (WWCA97)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin Germany, March 1997.