J Comput Virol (2008) 4:101–113
DOI 10.1007/s11416-007-0062-0

SSTIC 2007 BEST ACADEMIC PAPERS

Improving virus protection with an efficient secure architecture with memory encryption, integrity and information leakage protection

Guillaume Duc · Ronan Keryell

Received: 5 January 2007 / Revised: 15 July 2007 / Accepted: 18 August 2007 / Published online: 5 September 2007

Abstract Malicious software and other attacks are a major concern in the computing ecosystem and there is a need to go beyond the answers based on untrusted software. Trusted and secure computing can add a new hardware dimension to software protection. Several secure computing hardware architectures using memory encryption and memory integrity checkers have been proposed during the past few years to provide applications with a tamper resistant environment. Some solutions, such as HIDE, have also been proposed to solve the problem of information leakage on the address bus. We propose the CRYPTOPAGE architecture which implements memory encryption, memory integrity protection checking and information leakage protection together with a low performance penalty (3% slowdown on average) by combining the Counter Mode of operation, local authentication values and MERKLE trees. It has also several other security features such as attestation, secure storage for applications and program identification. We present some applications of the CRYPTOPAGE architecture in the computer virology field as a proof of concept of improving security in presence of viruses compared to software only solutions.

1 Introduction

Many computer applications need certain levels of security, confidentiality and confidence that are beyond the scope of current software and hardware architectures. Of course, there are many cryptographic algorithms, network protocols, secure operating systems and some applications that use these methods, but all of them rely on a strong hypothesis: the underlying software and hardware themselves need to be secure. However this critical hypothesis is never verified, except for small applications that can fit onto a smartcard, for example.

Software applications, such as anti-virus or operating systems, can be used to verify the integrity of the execution context or even repair it, but their execution themselves must be protected. Furthermore, software architectures have grown to a size far beyond what can be proved as inherently correct and contains unfortunately many errors, bugs... So, it is interesting to add specific hardware support to improve the global security of the system.

During the last few years, several hardware architectures (such as XOM [27–29], AEGIS [36,37] and CRYPTOPAGE [8,12,14,21,26]) have been proposed to provide computer applications with a secure computing environment. These architectures use memory encryption and memory integrity checking to guarantee that an attacker cannot hinder the operation of a secure process, or can only obtain as little information as possible about the code or the data manipulated by this process. These secure architectures try to prevent, or at least detect, physical attacks against the components of a computer (for example, the Microsoft X-BOX video game console was attacked in [19] by sniffing the bus of the processor with a logic analyzer) or logical attacks (for example, the administrator of the machine or a virus tries to steal or modify the code or the data of an application).

Such architectures can, for instance, be very useful in distributed computing environments. Currently, companies or research centers may be reluctant to use the computing power provided by third-party computers they do not control on a grid, because they fear that the owners of these computers might steal or modify the algorithms or the results of the

G. Duc (✉) · R. Keryell
ENST Bretagne, CS 83818, 29238 Brest Cedex 3, France
e-mail: Guillaume.Duc@enst-bretagne.fr

R. Keryell
e-mail: Ronan.Keryell@enst-bretagne.fr

distributed application. If each node of the grid uses a secure computing architecture that guarantees the integrity and the confidentiality of the distributed application results, these security issues disappear.

However, as the address bus is not modified in these secure architectures, the memory access patterns are accessible to the attacker. Zhuang et al. in [39] show that these memory access patterns can be sufficient to identify certain algorithms and so to obtain information about the code of a secure application, in spite of the encryption.

To stop this information leakage, they present HIDE (Hardware-support for leakage-Immune Dynamic Execution), an infrastructure for efficiently protecting information leakage on the address bus [39]. However, the impact of this infrastructure on encryption and memory checking is not studied.

In this paper, we present the CRYPTOPAGE extension of the HIDE infrastructure to provide, in addition to the protection of the address bus, memory encryption and memory checking. We also study the impact of this architecture on the computer virology field.

The rest of this paper is organized as follows. Section 2 describes the CRYPTOPAGE architecture. Section 3 presents some applications that can benefit from this architecture. Section 4 studies the relationships between this architecture and the computer virology field. Section 5 presents information about the performance of this system while Sect. 6 presents related work in this field.

2 Architecture

In this section, we present the security objectives and the key components of the CRYPTOPAGE architecture.

2.1 Objectives of the architecture

The objective of our architecture is to allow the execution of secure processes. We consider two properties:

- *confidentiality*: an attacker must obtain as little information as possible about the code or the data manipulated by the process;
- *integrity*: the proper execution of the process must not be altered by an attack. If an attack is detected, the processor must stop the program.

The processor has two new execution environment in addition to the normal non-secure execution environment. The first one protects the confidentiality and the integrity of the secure processes that are running in this environment. The second one only protects the integrity of the secure processes.

The processor must be able to execute secure processes in parallel with other normal (non-secure) processes with respect to an operating system adapted to the architecture but not necessarily secure or trusted. One key hypothesis is that the operating system itself does not need to be trusted. It may be compromised, or at least, be too large to be bug-free.

We assume that everything outside the chip of the processor (for instance the memory bus, the hard drive, the operating system, etc.) is under the full control of an attacker. For instance, he can inject bogus values in the memory, modify the operating system to wiretap processor registers or to modify hardware contexts, execute secure and non-secure processes of his choice, etc.

However, the attacker cannot directly or indirectly probe inside the processor. In particular, we consider that timing attacks [22], differential power analysis (DPA [23]) attacks, etc. are avoided by other means beyond the scope of this article. Moreover, we do not consider denial of service attacks because they cannot be avoided (the attacker can choose not to supply power to the processor).

The attacker can also modify the execution of system calls but we consider that these attacks are similar to denial of service attacks and that this problem should be taken into account at the application level.

We want to protect the integrity and confidentiality of a secure application against hardware attacks but we do not protect it against itself. If the secure application contains security holes, they can be exploited to modify its behavior.

2.2 Key mechanisms

In this section we describe the key mechanisms of the CRYPTOPAGE architecture implemented to achieve the security objectives described above. The low-level details of these mechanisms can be found in [7–14,21,26].

Figure 1 summarizes the CRYPTOPAGE architecture. The blocks in grey are those which are present in a normal processor and the other blocks are specific to the CRYPTOPAGE architecture. In spite of the fact that there are many new mechanisms, they are relatively cheap to implement in hardware compared to the number of transistors in existing processors.

Each CRYPTOPAGE processor is unique and has an embedded private key used to decrypt the symmetric keys of secure applications and to sign results. A certificate, signed by the manufacturer of the processor, attests to the authenticity of the corresponding public key. The private key can only be used by the security mechanisms of the processor and is not accessible by any attacker.

2.2.1 Confidentiality

In order to guarantee the confidentiality property, the CRYPTOPAGE architecture makes intensive use of encryption at different levels. The basic idea is that everything that is outside of the processor, and so under the control of an attacker,

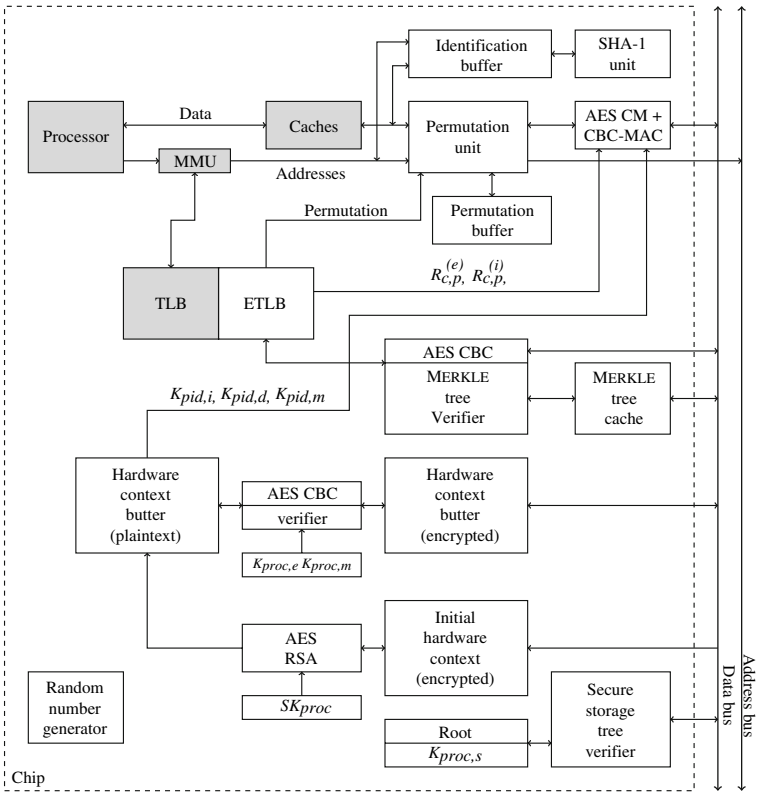


Fig. 1 The CRYPTOPAGE architecture

must be encrypted. This includes the code and the data of the secure processes when they are stored in the executable file or in memory, and the hardware contexts of the secure processes during interrupts.

As the processor is supposed to be impossible to be wire-tapped and to be tampered, the information that is inside the processor (data or code in caches for instance) can be stored unencrypted for better performances.

Memory encryption First, the code and the data of a secure process that is running in the execution environment that protects the integrity and the confidentiality, are encrypted while they are outside of the processor. Each secure process has two symmetric keys, $K_{pid,i}$ and $K_{pid,d}$ used to encrypt, respectively, its code and its data.

The encryption unit takes place on the processor between the highest level cache and the memory bus. As the unit of transfer between the processor and the memory is a line, the encryption engine works on lines. When a line is read from memory, it is first decrypted and then stored in the cache. When a line is unloaded from the cache, it is encrypted before leaving the processor.

The encryption algorithm used is the AES [32] in the counter mode of operation [33]. In this mode, a counter is encrypted using the AES algorithm to generate a pad and this pad is combined with the block, with a simple *bitwise exclusive or*, to generate the encrypted block. To decrypt a block, the pad is generated in the same way and combined with the encrypted block to generate the decrypted block.

The main advantage of this mode is that, if the counter used to encrypt a block is known, the pad can be computed in advance without the data block itself. This is, in particular, useful during read operation from memory because, if the counter is known, the pad can be computed while the encrypted block is read from memory. If the encryption engine is faster than the memory access, when then encrypted block arrives in the processor, the end of the decryption only requires one clock cycle (in the counter mode, the final operation is a simple *bitwise exclusive or*). So the latency added by the decryption process is negligible.

In the CRYPTOPAGE architecture, the counter used to encrypt a line is composed of the virtual address of the line and a random number chosen for the memory page of the line (see section on information leakage below). They are always available before the memory access so our architecture benefits from the advantage of the counter mode.

Interrupts During an interrupt, the hardware context of a secure process (the hardware context contains the state of the process, i.e., the values of all the registers of the processor) is saved to one of several special buffers inside the processor called hardware context buffers. Only the processor has access to these buffers so the operating system cannot wire-tap the state of the process. If the number of secure processes scheduled by the operating system is greater than the number

of hardware context buffers in the processor, the operating system can ask the processor to encrypt the hardware context of a secure process and to store it in memory. It can also ask the processor to load an encrypted hardware context from memory and to decrypt it to a hardware context buffer. So, even if an hardware context is in memory, the operating system, or some other processes cannot access it thanks to the encryption mechanism.

During an interrupt, after the storage of the hardware context of the secure process in an hardware context buffer, the content of the registers of the processor is erased so the operating system cannot access it. However, a secure process can ask the processor not to erase the content of some registers, for instance to allow the storage of the arguments of system calls.

Information leakage on the address bus The CRYPTOPAGE architecture is based on the HIDE infrastructure [39]. This infrastructure reduces the information leakage that exists on the address bus because, even if all the code and data of a secure process is encrypted while they are transferred and stored in memory, the address bus is not modify so an attacker can retrieve the memory access patterns. Zhuang et al. in [39] have shown that these memory access patterns can be sufficient to identify certain algorithms and so to obtain information about the code of a secure application, in spite of the encryption and so, to violate the confidentiality property.

The HIDE infrastructure, as described in [39], memorizes the sequence of addresses accessed by the processor and permutes the memory space before an address recurs. More precisely, the protected memory space is divided into several chunks. This protection is implemented by modifying the cache mechanism of the processor. Whenver a line that belongs to a protected chunk is read from memory (during a cache miss), it is stored in the cache, as usual, but it is also locked. While locked, this line cannot be replaced or written back to memory. Whenver a line needs to be flushed because there is no space left in the cache for a new line and all lines are locked, a permutation of the chunk occurs.

During this permutation, all the lines belonging to the chunk that is being permuted are read (from memory or from the cache), then the addresses of these lines are permuted and the lines are unlocked and re-encrypted. So between each permutation, a line is written to memory and read from memory only once. In addition, the re-encryption of the lines prevents an attacker from guessing the new address of a given line inside a chunk after a permutation. With this mechanism, an attacker cannot learn that one line has been read more than another nor can he find interesting memory access patterns at a grain finer than the chunk size. The current permutation table for a chunk is encrypted and stored in memory.

To reduce the cost of the permutations, Zhuang et al. [39] proposes to do the latter in the background. When the processor detects that the cache (or one set of the cache, depending on its structure) is almost full, it begins a background permutation. With this mechanism, the performance penalty is negligible (1.3% according to [39]).

In the CRYPTOPAGE architecture, the secure processes are protected with HIDE. The chunks used are the memory pages. In addition, during the permutation of a page, two random numbers are chosen by the processor. They are used by the mechanisms that protect the integrity (they are included in the computation of the MAC) and confidentiality (they are included in the computation of the counter used to perform the encryption) of the data of the page.

2.2.2 Integrity

In this section, we describe how the integrity of a secure process is protected.

Memory integrity In the CRYPTOPAGE architecture, the integrity of the code and the data of a secure process is protected by computing a *Message Authentication Code* (MAC) with a symmetric key $K_{pid,m}$, specific to a secure process. Since an attacker does not know this key, it cannot inject a piece of data in memory because it cannot compute a valid MAC for it. In addition, the address of the data protected is also included in the computation of the MAC so an attacker cannot permute two pieces of data in memory.

However, to protect the integrity of a line against replay attacks (i.e., an attacker that puts, at a given address in memory, a value and its MAC that were previously stored at this address), the MAC is not sufficient in itself. To solve this problem, the computation of the MAC also includes a random number that is specific to a memory page and chosen during each permutation of the page.

The random number used to compute the MAC of the lines of a memory page is stored with the other information of this page (for instance, the permutation table needed by HIDE). To speed up the access to these information, they are stored in an extension of the *Translation Lookaside Buffers* (TLB), called ETLB.

However, if an attacker can replay these pieces of information (and so the random number that protects the integrity), it can replay data in memory. So the CRYPTOPAGE architecture has a mechanism to protect the information with respect to a page against replay attacks. This mechanism is based on hash trees (MERKLE tree).

A binary tree is built. Each node of the tree contains a cryptographic hash of the content of its children. The leaves of the tree contain a cryptographic hash of the information about a page. The root of the tree is stored inside the processor so it cannot be replayed. When the information about a memory page is load from memory to the ETLB, the corresponding branch of the tree (from the leaf that corresponds to

the page to the root) is checked. When they are modified, the corresponding branch is updated. As the root of the tree is unalterable, with this mechanism, an attacker cannot replay the information about a page and so it cannot replay data in memory.

Interrupts The integrity of any hardware context is protected with a MAC computed with a symmetric key embedded in the processor, so an attacker cannot modify them, even if the hardware contexts has to be stored outside of the processor.

An optional mechanism, based on a log storing information about the storage of hardware context outside of the processor, prevents an attacker from replaying an old hardware context (i.e., trying to reload a secure process in a past state).

2.2.3 Other mechanisms

Other mechanisms that can be useful to secure applications are implemented in the CRYPTOPAGE architecture.

Signals A special mechanism is implemented to allow the processing of software signals (used in UNIX operating systems). A signal requires the diversion of the normal execution flow of a process but this situation can be seen as an attack attempt by the CRYPTOPAGE architecture.

The secure processes can specify to the processor the address of a function that has to be called whenever a signal is triggered. The operating system, with a special instruction, can ask the processor to execute this signal handler for a secure process. So, the operating system cannot modify the execution flow as it wants. Another special instruction allows the operating system to restore the original hardware context (the one saved before the triggering of the signal) and so, to restore the original execution flow, but while keeping the modifications made in memory by the signal handler.

This mechanism also forces the signal starts and the signal ends to be properly nested.

Secure storage The CRYPTOPAGE architecture provides each secure process with a small secure non-volatile storage space protected against replay attacks. In this place, a secure process can store encryption keys or cryptographic hashes that can be used to protect a larger storage space.

This secure storage mechanism is also based on a MERKLE tree. The root of this storage tree is securely stored inside the processor. Some of the operations of verification and update of this tree are delegated to the operating system in order to increase flexibility but without compromising the security of the mechanism.

Program identification In some cases, for instance electronic voting or open-source digital rights management applications, it may be useful to have access to the code source (for example to check that there is no backdoor in the program) of a secure process that runs in the execution environment that protects the integrity and confidentiality of its code and data (because it manipulates sensitive data).

The CRYPTOPAGE architecture has a mechanism that allows the user to know if an encrypted binary file contains the same program that a non-encrypted binary file. So, if the user is provided with exactly the same build environment that the one used to generate the secure application, it can recompile it from the source and use this mechanism to know whether the result of this recompilation is the same as the secure process in spite of the encryption.

The processor computes a cryptographic hash over the initial code and data of the secure process and returns it to the user so it can compare it with the one computed over the program it has rebuilt.

Attestation In some cases, a secure application may have to prove to an external entity that it is running on a CRYPTOPAGE architecture and so that its confidentiality and integrity are protected.

In the execution mode that protects both the integrity and confidentiality, this feature is not necessary because the secure application can hold secrets (for instance asymmetric keys and certificates) directly embedded in its code (and hence encrypted). These secrets can be used to established a secure transmission channel (with the SSL protocol for example) with an external entity. As the code of the secure application is encrypted with the public key of the CRYPTOPAGE processor on which the application will be executed, and as the corresponding private key is embedded inside the CRYPTOPAGE processor, and so only accessible by it, if the secure communication is properly established, the secrets must have been deciphered and so, the application must be running on a correct CRYPTOPAGE architecture.

However, this feature is useful in the execution mode that only protects the integrity of a secure application because the solution described before is no longer applicable. Indeed, the confidentiality of the code and data manipulated by a process in this execution environment is no longer protected. So the CRYPTOPAGE architecture has a special instruction that allow a secure process to ask the processor to sign a result. The result given by the secure process and an identifier of the secure process are signed with the public key of the processor, and the certificate produced by the manufacturer of the processor is attached. Thus, the external entity can check the signature, verify that it was generated by a CRYPTOPAGE processor and identify the process that produced the result.

2.2.4 Operating system

To support the CRYPTOPAGE architecture, the operating system needs to be adapted. However, it does not require to be trusted because it cannot break the properties that the architecture guarantees to secure processes.

A special flag in the executable file of an application indicates whether the application requires one of the two secure execution environments (integrity only or integrity and confidentiality) or not. If the operating system detects this flag, it loads the code and data in memory and loads the initial hardware context of the application (which contains its symmetric keys, encrypted with the public key of the processor) in the initial hardware context buffer. Then, it asks the processor to decrypt it. When the decryption is achieved, the secure process can be started from the initial hardware context that has just been decrypted.

During interrupts and context switches, the operating system has to manage the hardware context buffers. It may have to ask the processor to encrypt the hardware context of a secure process to store it in memory and to load and decrypt the hardware context of another secure process. These operations can be time consuming, so the scheduler may give the priority to the secure processes whose hardware contexts are already present in an hardware context in the processor because, in this case, the context switch is faster.

The operating systems also plays a part in several security mechanisms of the CRYPTOPAGE architecture such as the verification of the MERKLE tree that protects the information about the memory pages or the verification of the secure storage tree. These delegations reduce the hardware modifications needed to implement these mechanisms and allow more flexibility in their management.

2.2.5 Impacts on applications

In order to generate secure applications, the compilation tools have to be adapted. After the link phase, the symmetric keys (used for the encryption of the code and the data and the protection of their integrity) of the application are generated and encrypted with the public key of the processor on which the application will be executed. Next, the code and the data of the application are encrypted using these keys.

The system calls also need some modifications. Before a system call, a secure process must ask the processor not to erase the registers where the arguments of this system call are stored. If necessary, it must decrypt these arguments if they are stored in memory so the operating system can access them. These operations can be implemented in the system call wrappers of the C standard library for instance.

3 Applications

Before presenting how CRYPTOPAGE can secure existing applications, we present some applications impossible to do without a secure execution mode such as the one provided by CRYPTOPAGE.

3.1 Secure distributed computing

Grid computing is a very topical subject where the CRYPTOPAGE architecture could be very useful to provide secure grid computing.

A grid groups together the computing power of many computers which can be distributed on many different places. These computers can be dedicated to this task (for instance it is the case in the French grid project called Grid'5000 [17]) or not. In the latter case, the computers are owned by universities, companies or even the general public and only the unused processor time is used by the grid applications (it is the case, for instance, of the Folding@Home [15] project). These grids are more and more used by research centers or companies.

However, the entities that submit applications on the grid have not, in general, a physical control on all the nodes of the grid. The owner of one of these nodes can—by making an physical or a software attack—recover the code or the data of the application that is running on its node, or disturb it in order to force it to generate false results. This lack of confidentiality and integrity either on the code or the date can raise a problem of intellectual property or performance (if the computations have to be performed several times on different nodes to check their integrity). These issues can slow down the adoption of the grids.

If all the nodes of the grid have a CRYPTOPAGE processor, the integrity and the confidentiality of the applications that run on the grid can be guaranteed against attacks, and so, it would be possible to perform truly secured distributed computing as we are going to detail.

First, the user who needs to do some remote computations enciphers the program with a secret session key and constructs an enciphered binary with as many enciphered execution contexts as remote CRYPTOPAGE processors available with each public key.

In this way, the user can run opaque computations remotely and the computer owner cannot read the code nor the data, neither modify them in an unnoticed way, since nobody (even the owner) knows the secret key of a CRYPTOPAGE processor. But an attacker could build a virtual CRYPTOPAGE with a public key pair and claim it is a genuine CRYPTOPAGE and give the public key to the user. If the user ciphers the session key with this public key, the attacker will recover the session key and will be able to access the code and the data or even modify them. To avoid this attack, a *Public Key Infrastructure*

(PKI) must be deployed to attest that a CRYPTOPAGE public key really corresponds to a genuine CRYPTOPAGE processor.

The user can verify the correct execution of the program by adding an authentication protocol in the enciphered program. All the communications between the user and the remote nodes are encrypted using some cryptographical algorithm implemented in the enciphered program and cannot be tapped since the secure execution mode of CRYPTOPAGE avoids information leakage during execution.

If the user does not need to hide the data or the code, the program can run in the execution environment that only protects the integrity. But then, the attacker, with full control of the remote computer, can peek in the authentication protocol instructions and secrets and thus attests anything to the user. To avoid this attack, we use the attestation mechanism added in CRYPTOPAGE to attest the correct execution of *the* program in the integrity mode.

In this way, we can assure the user of a correct and opaque execution of the program. But in a collaborative distributed computing, the remote computer owner may want to know what is running on its computer and not to give resources to anybody for any usage. To attest the identity of a running process even in secure mode, the operating system asks the processor to compute a fingerprint of the process that can be used by the computer owner to verify, in a global directory for instance, that it is a given program and not another one.

As special interesting cases of remote secure executions, we can cite:

- anti-virus where the code must execute securely on the remote computer of an end-user. The end-user cannot peek into the anti-virus binary because it runs in enciphered mode to protect the intellectual property of the anti-virus company but must be sure that the program is really the anti-virus it claims to be. For this purpose, CRYPTOPAGE attests with the fingerprint of the program that it is really an anti-virus of the anti-virus company;
- one can build open-source DRM (Digital Right Management) systems, since the execution is enciphered by CRYPTOPAGE. In this way, the public keys are secretly generated by the program with the secure execution mode and can be used to decipher the media content according to the license conditions granted by the copyright holder. But the user must be sure in exchange that the DRM application does not contains malicious code such as spywares. If the DRM program is open-source, the user can inspect the code to verify that it is correct. But to guarantee that secrets of the DRM application remains secret, the process must run in secure mode and thus, the binary must be enciphered by the producer of the DRM and not the user. But then, how the user can be sure that the secure process has really the binary produced by the source code since it cannot peek in the process? By reproducing the compilation chain and fingerprinting the code. First the user compile the open-source DRM application according to exactly the same process (same compiler, same libraries...) as documented by the DRM producer. Afterward, it builds an enciphered binary for its CRYPTOPAGE by using its public key. It starts execution of this process and asks its operating system to ask the CRYPTOPAGE processor to compute the fingerprint of the process. If it is the same as the fingerprint of the DRM process, it is the same program.

3.2 Virtual smart cards

Smart cards are more and more used, at least in Europe, in several applications from electronic banking to health insurance, telecommunications, etc.

Thus, a person has more and more smart cards to carry. In addition, only few computers are equipped with smart card readers and so we cannot take into account the security of the smart cards during computer based operations (online purchase for instance).

With the CRYPTOPAGE architecture, it is possible to create virtual smart cards. The idea is to turn each smart card into a secure application that will run only on this instance of the CRYPTOPAGE architecture. The security mechanisms, which protect the integrity and confidentiality, guarantee the same security properties than smart cards. The secrets (asymmetric keys for instance) that are normally embedded in the smart card can be directly integrated in the encrypted executable file. The non-volatile memory of the smart card can be replaced by the secure storage mechanism. Finally, the communication channels between the outside world (the reader) and the card are replaced by the different input/output channels that can be used by the applications (inter-process communication, network sockets, etc.).

Thus, the smart cards of a user can be totally dematerialized and, for instance, be put together and used on a computer equipped with a CRYPTOPAGE processor (for non-mobile applications) or on a small personal digital assistant or even on a mobile phone with a CRYPTOPAGE processor (for mobile applications).

The program identification feature of CRYPTOPAGE can also be used in this case to allow the user to access the source code of the smart cards and thus verify that they do what they have to do.

3.3 Other applications

Several other applications can use the two secure execution environments provided by the CRYPTOPAGE architecture, such as electronic voting systems, online commercial applications (business, banking...), etc.

4 Benefits for the computer virology field

In this section, we will present the relationships between the CRYPTOPAGE secure architecture and the field of computer virology and the benefits of this architecture to fight against viruses or malicious codes.

4.1 Processes isolation

In the CRYPTOPAGE architecture, the secure processes are totally isolated from the operating system or from the other processes (secure or not). The code and the data of a secure process cannot be altered, its execution cannot be disturbed (if an attack that may have an impact on the execution of a secure process is detected, the secure process is stopped so it cannot produce false results) and, if it is running in the execution environment that also guarantees its confidentiality, it cannot be wiretapped.

So, if an anti-virus program runs as a secure process, its execution cannot be altered by a virus that would try to prevent the anti-virus from being able to detect it. However, the virus can still alter the operating system to prevent it from launching the anti-virus program but this attack can be mitigated if the anti-virus has a method to show to the user that it is still running. A simple case is that a secure application, that wants to be sure that the anti-virus runs, explicitly communicates with an enciphered channel with the anti-virus. To have this communication working, the operating system must run the anti-virus process to avoid denial of service on the application and then the anti-virus can do its job.

In addition, a malicious software cannot wiretap a secure process to steal some information such as the encryption keys generated during the creation of a SSL/TLS tunnel since all the data are covered.

The integrity and the confidentiality of the data stored by a secure process using the secure storage feature of the CRYPTOPAGE architecture are also protected. This secure storage can, for instance, be used to store user's certificates, secret keys and passwords, the encryption keys and licenses used by digital rights management applications, etc.

However, if the input/output devices (keyboard, mouse, graphics card) are not modified, a malicious program (a keylogger for instance) can still wiretap the communications between a secure process and the user of the computer and so retrieves passwords for instance.

4.2 Code injection

The CRYPTOPAGE architecture also prevents an attacker from injecting a piece of code though data channels into a secure process because the encryption key and the verification keys used to encrypt or protect the code and the data of a secure process are different. So if an attacker tries to inject a piece of code with an overflow buffer for example, it would be encrypted using the key dedicated to the data and read again and decrypted and verified with the key dedicated to the code.

Most processors have already a mode to disable execution in data area at a memory page level that can mitigate direct code injection (unfortunately, this mode is far too recent on *Intel* or *AMD* PC processors with *No-eXecution* bit, compared to other processors) but this feature adds another level of protection.

However, this feature does not protect a secure process from jumping to a piece of code that is already in it (for instance *return to libc* attacks). It also does not prevent buffer overflows.

4.3 Reverse engineering

As we have seen, the CRYPTOPAGE architecture has several dual features that can mitigate the threat of computer viruses. However, it has also drawbacks.

If a malicious program is running in the execution environment that protects its confidentiality, it is impossible to analyze it to discover what it does. This can be problematic, for instance, for anti-virus companies that would not be able to study an encrypted virus. However, some information still leak from a secure process such as system calls or network operations.

The program identification feature of the CRYPTOPAGE architecture allows the user to be sure that an encrypted binary file correspond to a given source code. It can decide not to execute a secure processes if it do not have the corresponding source code and so that could be a malicious application. However, this is only possible if the user is provided with the source code of the secure applications and the same building environment that the one used to compile them.

5 Evaluation and results

In this section, we use the same architectural parameters as [39] to evaluate the performance of our architecture. They are summarized in Table 1. The AES unit is fully pipelined and its latency is 11 cycles.

To evaluate the CRYPTOPAGE architecture, we ran several SPEC2000int [18] benchmarks with the SimpleScalar [1] out-of-order simulator modified to implement our architecture. To reduce the time required to perform the simulations, we skipped the first 1.5 billion instructions and we performed a detailed simulation for 200 million instructions.

In Fig. 2a, we compare the instructions per cycle (IPC) ratio of each benchmark for three different implementations:

- our implementation of the HIDE [39] infrastructure (8K chunks, all chunks protected, no layout optimizations);

Table 1 Architectural parameters used in simulations

Architectural parameter	Specifications
Clock frequency	1 GHz
Cache line size	32 bytes
Data L1	Direct mapped, 8 kB, LRU
Instruction L1	Direct mapped, 8 kB, LRU
L1 miss latency	1 cycle
Unified L2	Four-way associative, 1 MB, LRU
L2 miss latency	12 cycles
ITLB	Four-way associative, 64 entries, LRU
DTLB	Four-way associative, 128 entries, LRU
TLB miss latency	30 cycles
Memory bus	200 MHz, 8 bytes wide
Memory latency	80 (first), 5 (inter) cycles
Page size	8 kB
Fetch/decode width	32/8 per cycle
Issue/commit width	8/8 per cycle
Load/store queue size	64
Register update unit size	128
Encryption algorithm	AES
Encryption block length	128 bits (so $l = 2$)
Encryption latency	11 cycles

- a basic implementation of our architecture (without a MERKLE tree cache, the instructions have to be checked before being executed);
- an advanced implementation of our architecture (with a fully-associative MERKLE tree cache of 512 entries, speculative execution of instructions during integrity checking).

All the IPC are normalized to the original value obtained when we run the benchmarks on a normal, unmodified, architecture without any security features.

Our basic architecture has exhibited bad results (up to 50% slowdown on some benchmarks). This is partly due to the high cost of the verification of the MERKLE tree at each TLB miss, especially with the benchmark `mcf` which has a high TLB miss ratio.

The advanced version of our architecture has exhibited good results. The average slowdown is only 3% and the worst is 7.4% (with the benchmark `parser`).

In Fig. 2b, we compare the effect of four different sizes for the MERKLE tree cache: without a cache, with a 256-entry cache, with a 512-entry cache and with a 1,024-entry cache. We see that the introduction of a cache, even a small one, has a great impact on the performance of the applications. However, the increase in the size of this cache has only a minor impact because the number of memory pages used by these applications is far greater than the number of entries in the cache. The ideal solution would be that the size of this cache would be sufficient to store all the page information and all the nodes of the MERKLE tree.

The storage of the MAC increases the memory usage of the secure program by 50% using our parameters. To reduce this memory footprint, the MAC can be computed over several lines instead of only one, to the detriment of the time needed to check this MAC and of the memory bus use. Figure 2c shows the impact of computing the MAC over one, two and four cache lines without speculative execution of instructions (CRYPTOPAGE architecture with a 512-entry MERKLE tree cache). The mean performance penalty is 4.4% for computing the MAC over two cache lines and 11.7% over four cache lines. If we execute the instructions before their control, the performance penalty is very small (less than 1%, as shown in Fig. 2d).

We used the *SimpleScalar* micro-architecture simulator to obtain performance results that can be compared with the other architectures that also use *SimpleScalar* or similar micro-architecture simulators. However, it only simulates the execution of a process itself and does not take into account the impact of the operating system or other processes running on the processor. For instance, the kernel of the operating system is normally called frequently with a clock interrupt in a multitask operating system, and this execution can increase the cache miss rate of the process which was running.

We are currently working to estimate the hardware cost (number of logic gates) of our modifications.

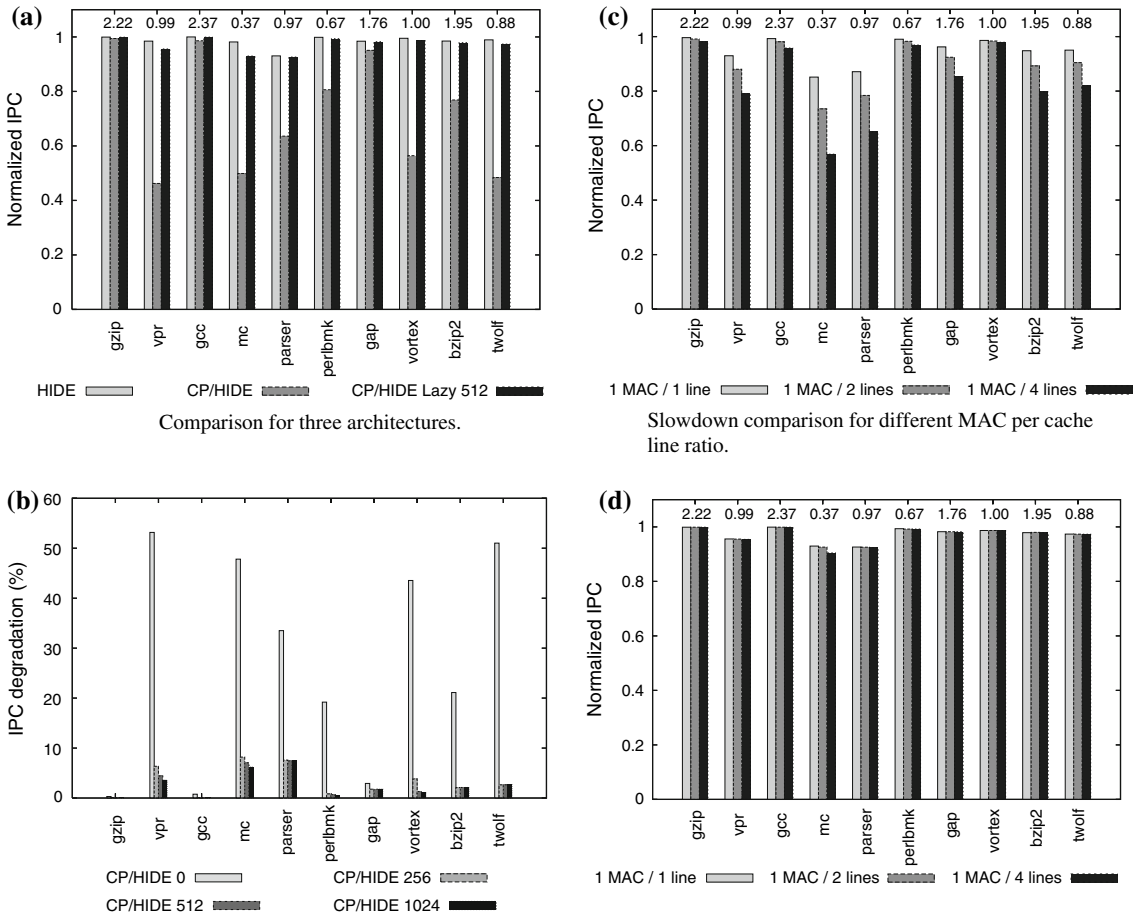
6 Related work

Several secure architectures have been proposed in the past. We can basically divided them into two categories: the secure architectures that use bus encryption and those that use an external co-processor.

The idea of using an encrypted bus has been introduced by Best in [2–5]. The processor has a secret key that it uses to decrypt a program stored in memory. The encryption guarantees the confidentiality of the program. However, the processor supports the execution of only one process and does not protect its integrity.

In [24], Kuhn has proposed to integrate an asymmetric key pair in the secure processor in order to facilitate the diffusion of the secure applications. He has also proposed a mechanism that allows the execution of an operating system and several secure processes.

In [27], Lie et al. has proposed the XOM (*eXecute-Only Memory*) architecture that guarantees the confidentiality of secure processes but also their integrity by computing an authentication value (MAC) for each line stored in memory. Thus, any attempt to modify the content of the memory would be detected by the processor.



Comparison for three architectures.

Slowdown comparison for different MAC per cache line ratio.

Slowdown comparison for four Merkle cache sizes.

Slowdown comparison for different MAC per cache line ratio (speculative execution).

Fig. 2 Simulation results

In [36], Suh et al. has introduced the AEGIS architecture. This architecture use two new methods to protect the integrity of the memory against replay attacks (XOM was vulnerable to these attacks). The first method [16] is a MERKLE tree computed over the whole memory space (this method was also proposed by Lauradoux and Keryell in [26]). The cache hierarchy of the processor is used to improve the speed of the control. However, even with the use of the cache, the performance penalty of this control mechanism is 20%. The second method is an off-line memory checker built using incremental hash functions. An off-line memory checker only checks the integrity of a set of transactions whereas on-line memory checkers check the integrity of memory during each memory transaction. Off-line memory checkers are faster but the control function has to be called before each security-critical instruction. But between each control, the instructions are executed without verification and this can cause some information leakages in case of this mechanism is combined with memory encryption.

The works on the domain of bus encryption architectures are mainly academic. Only few architectures was commercialized. One of them is the DALLAS DS5000 family [6]. The processors of this family can execute an encrypted program stored in memory. However, they do not offer a protection of the integrity and they were attacked by Kuhn in [25].

Another way to allow the execution of secure processes is to use a co-processor. This is for instance the case with the IBM 4758 [20,34,35] which is a shielded PCI card that can be added to a standard PC. The card contains a processor, some cryptographic hardware accelerators, standard and flash memory. The programs can run inside the card. If an attacker tries to tamper the card, an auto-destruction mechanism is triggered.

We also have to mention the efforts of the Trusted Computing Group [38] and of Microsoft with its project *Next Generation Secure Computing Base* (NGSCB) [30,31] in order to secure general purpose computers. However, their objectives in terms of security are different from ours. We want

to protect the integrity and confidentiality of the code and data of the processes that are running on our architecture, even in the case of a hardware attack, whereas they want to provide strong isolation at the software level between processes, remote attestations that the hardware and software are in a given state, sealed storage and a secure path between the input devices to the application. Our objectives cannot be reached by simply using a *Trusted Platform Module* (TPM) and a secure operating system.

7 Conclusions and future work

In our computer and network centric world, computer reliability and security is a bigger concern than ever. Security must be addressed at all levels and software integrity is not the easiest part to ensure with the increasing complexity going far beyond what can be automatically proved as secure. Up to now, software integrity has been warrantied by other software procedures and it is a source of intractability.

Hardware support to increase software security seems a promising approach to complete a pure software solution. Research groups have already given solutions such as the TPM from the Trusted Computing Group [38] but there are strong hypothesis on the behaviour and the security of the software and operating system on the platform. Any software security breach can be exploited by a computer virus and hardware attacks are not considered outside the TPM. Most other secure architectures are in the smartcard area but even if smartcards have increasing performance, they cannot cope with the performance of general purpose processors. Thus there is a need to go further in the choice between performance and security and it is time to use more transistors from the MOORE's law stock for security instead of only throwing all of them at performance.

In this work, we described the CRYPTOPAGE, an advanced secure architecture which implements a cheap but secure memory encryption and protection mechanism, based on the HIDE architecture and recursive hash functions with aggressive caching, which prevents information leakage from applications on the data and address bus and can resist to active external attacks, even in presence of a malicious operating system or logic analyzer.

The basic idea is to add a MERKLE hash tree only at a coarse grain level (the operating system page descriptor TLB) to protect the architecture against replay attack at a low penalty cost. A few novel secure instructions are added to delegate securely the security management to the untrusted operating system and simplify the hardware secure mechanism.

Once the security is granted at the TLB level, some secrets are enciphered and added to the TLB to efficiently implement a memory encryption and address shuffling inside each memory page based on counter mode encryption and an anti-replay protection based on a CBC- MAC.

We already have a GNU/Linux version supporting preliminary secure mechanisms of CRYPTOPAGE. This kernel can run in the plain insecure mode and manage insecure or secure processes. Even if the secure processes are under control of the operating system, the secure architecture ensures process integrity and confidentiality and stops execution if the operating system or other piece of software does not play its role.

The performance penalty of these mechanisms, compared to a normal non-secured architecture, turns out to be only about 3% on average on some SPEC2000int benchmarks, which is far smaller than that achieved by previous works. This result is achieved thanks to the property that one line of memory is read and written only once between each chunk permutation.

We described some useful applications of the CRYPTOPAGE architecture to solve some common security problems and we listed some impacts of CRYPTOPAGE on the computer virology field. Secure processes can run user applications in a safe way even in presence of virus or external hardware attacks or, worse, operating system corruption. A classical anti-virus software will take obvious advantage to run in a secure mode since it is a critical part of many security policies. CRYPTOPAGE provides also some secure storage accessible in a secure way that cannot be modified by an attacker without uncovering the forgery. This integrity and confidentiality of this secure storage is a base to global secure distributed infrastructure. At last, process authentication and signature of results can prove that a given program ran correctly and not another one instead, warrantying no process hijack. This architectural support can leverage the classical security by obscurity approach (code obfuscation...). Since CRYPTOPAGE can run a process in a secure mode with identity attestation, even critical components (secure routing, antivirus, DRM...) can be open-source since everybody can verify that the running program is correct, without being able to peek in or modify the data. So we can envisage a true collaborative secure distributed computing, with users securely running their secure processes on remote untrusted computers but with CRYPTOPAGE, without bothering about results tampering or confidentiality issues. The owners or administrators of the remote computers will be able to accept or refuse execution of these opaque processes according classical trust rules or, more interestingly, on the attestation by CRYPTOPAGE that the processes are executing some given source programs and not some others.

To go further, we are now working to extend our proposition to multicore and multiprocessor systems and to apply our architecture to secure distributed high performance computing. We need to go on porting GNU/Linux to the latest version of CRYPTOPAGE to deal with secure storage, memory

and process authentication. To ease secure virtualization, we are also considering to virtualize CRYPTOPAGE itself to be able to run various CRYPTOPAGE domains with various operating systems securely protected without relying on a trusted virtualization monitor.

Acknowledgments This work is supported in part by a Ph.D. grant from the *Délégation Générale pour l'Armement* (DGA, a division of the French Ministry of Defense), and funded by the French National Research Agency (ANR) under contract ANR-05-SSIA-005-03 SAFESCALE. The authors wish to thank Jacques Stern for his valuable comments on this project, Sylvain Guilley and Renaud Pacalet for their insightful discussions in the GET- TCP project.

References

1. Austin, T., Larson, E., Ernst, D.: SIMPLESCALAR: An infrastructure for computer system modeling. Computer **35**(2), 59–67 (2002)
2. Best, R.M.: Microprocessor for executing enciphered programs. Technical Report US4168396, US Patent, Sept 1979
3. Best, R.M.: Preventing software piracy with crypto-microprocessors. In: IEEE Spring CompCon'80, pp. 466–469. IEEE Computer Society, February 1980
4. Best, R.M.: Crypto microprocessor for executing enciphered programs. Technical Report US4278837, US Patent, July 1981
5. Best, R.M.: Crypto microprocessor that executes enciphered programs. Technical Report US4465901, US Patent, August 1984
6. Dallas Semiconductor. DS5002FP Secure Microprocessor Chip, July 2006. http://datasheets.maxim-ic.com/en/ds/DS5002FP.pdf
7. Duc, G.: CRYPTOPAGE—an architecture to run secure processes. Diplôme d'Études Approfondies, École Nationale Supérieure des Télécommunications de Bretagne, DEA de l'Université de Rennes 1, June 2004. http://enstb.org/~gduc/dea/rapport/rapport.pdf
8. Duc, G.: Support matériel, logiciel et cryptographique pour une exécution sécurisée de processus. Ph.D. thesis, École Nationale Supérieure des Télécommunications de Bretagne (2007). http://enstb.org/~gduc/these/these.pdf
9. Duc, G., Keryell, R.: Portage d'un systéme GNU/LINUX sur l'architecture sécurisée CRYPTOPAGE/x86. Technical report, ENST Bretagne, December 2004. http://info.enstb.org/projets/cryptopage/documents/techreport_200412.pdf
10. Duc, G., Keryell, R.: The concept of secure processes for LINUX on the CRYPTOPAGE/x86 secure architecture. Technical report, ENST Bretagne, April 2005. http://info.enstb.org/projets/cryptopage/documents/techreport_200504.pdf
11. Duc, G., Keryell, R.: Portage de l'architecture sécurisée CRYPTOPAGE sur un microprocesseur x86. In: Symposium en Architecture nouvelles de machines (SympA'2005), pp. 61–72, April 2005
12. Duc, G., Keryell, R.: CRYPTOPAGE: an efficient secure architecture with memory encryption, integrity and information leakage protection. In: Proceedings of the 22th Annual Computer Security Applications Conference (ACSAC'06), pp. 483–492. IEEE Computer Society, December 2006
13. Duc, G., Keryell, R.: CRYPTOPAGE/HIDE: une architecture efficace combinant chiffrement, intégrité mémoire et protection contre les fuites d'informations. In: Symposium en Architecture de Machines (SympA'2006), October 2006
14. Duc, G., Keryell, R., Lauradoux, C.: CRYPTOPAGE: Support matériel pour cryptoprocessus. Techn. Sci. Inform. **24**, 667–701 (2005)
15. Folding@home distributed computing, May 2007. http://folding.stanford.edu/
16. Gassend, B., Suh, G.E., Clarke, D., van Dijk, M., Devadas, S.: Caches and hash trees for efficient memory integrity verification. In: Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03), pp. 295–306, February 2003
17. Grid'5000, May 2007. http://www.grid5000.fr
18. Henning, J.L.: SPEC CPU2000: measuring CPU performance in the new millennium. IEEE Comput. **33**(7), 28–35 (2000)
19. Huang, A.: Keeping secrets in hardware: the Microsoft XBox (TM) case study. Technical Report AI Memo 2002–2008, Massachusetts Institute of Technology, May 2002
20. IBM PCI cryptographic coprocessor, May 2007. http://www.03.ibm.com/security/cryptocards/pcicc/overview.shtml
21. Keryell, R.: CRYPTOPAGE-1: vers la fin du piratage informatique? In: Symposium d'Architecture (SympA'6), pp. 35–44, Besanton, June 2000
22. Kocher, P.C.: Timing attacks on implementations of DIFFIE-HELLMAN, RSA, DSS, and other systems. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'96), vol. 1109, pp. 104–113. Springer, Heidelberg, August 1996
23. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'99), vol. 1666, pp. 388–397. Springer, Heidelberg, August 1999
24. Kuhn, M.: The TRUSTNO1 cryptoprocessor concept. Technical Report CS555, Purdue University, April 1997
25. Kuhn, M.G.: Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP. In: IEEE Transaction on Computers, vol. 47, pp. 1153–1157. IEEE Computer Society, October 1998
26. Lauradoux, C., Keryell, R.: CRYPTOPAGE-2: un processeur sécurisé contre le rejeu. In: Symposium en Architecture et Adéquation Algorithme Architecture (SympAAA'2003), pp. 314–321, La Colle sur Loup, France, October 2003
27. Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), pp. 168–177, October 2000
28. Lie, D., Trekkath, C.A., Horowitz, M.: Implementing an untrusted operating system on trusted hardware. In: Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP'03), pp. 178–192, October 2003
29. Lie, D.J.: Architectural support for copy and tamper-resistant software. Ph.D. thesis, Stanford University (2004)
30. Microsoft Corporation. NGSCB: Trusted Computing Base and Software Authentication (2003). http://www.microsoft.com/resources/ngscb/documents/ngscb_tcb.doc
31. Microsoft Corporation. Security Model for the Next-Generation Secure Computing Base (2003). http://www.microsoft.com/resources/ngscb/documents/NGSCB_Security_Model.doc
32. NIST. Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, November 2001
33. NIST. Recommendation for block cipher modes of operation. Special Publication 800-38A, December 2001
34. Smith, S.W.: Trusted Computing Platforms: Design and Applications. Springer, Heidelberg (2004)
35. Smith, S.W., Weingart, S.: Building a high-performance, programmable secure coprocessor. Comput. Netw. **31**(9), 831–860 (1999)
36. Suh, G.E., Clarke, D., Gassend, B., van Dijk, M., Devadas, S.: AEGIS: Architecture for tamper-evident and tamper-resistant processing. In: Proceedings of the 17th International Conference on Supercomputing (ICS'03), pp. 160–171, June 2003

37. Suh, G.E., O'Donnell, C.W., Sachdev, I., Devadas, S.: Design and implementation of the AEGIS single-chip secure processor using physical random functions. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05), pp. 25–36. IEEE Computer Society, June 2005
38. Trusted Computing Group, February 2007. http://www. trusted-computinggroup.org
39. Zhuang, X., Zhang, T., Pande, S.: HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI), pp. 72–84. ACM Press, October 2004