CryptoPage: an Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection

Guillaume Duc, Ronan Keryell
ENST Bretagne
CS 83818, 29238 Brest Cedex 3, France
{Guillaume.Duc,Ronan.Keryell}@enst-bretagne.fr

Abstract

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.*

1. Introduction

Many computer applications need certain levels of security, confidentiality and confidence that are beyond the scope of current 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 hardware itself needs to be secure. Indeed this critical hypothesis is never verified, except for small applications that can fit onto a smartcard, for example.

During the last few years, several hardware architectures (such as XOM [16, 17, 18], AEGIS [26, 27] and CRYPTOPAGE [12, 15, 6, 5]) 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 disturb 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 [10] by sniffing the bus of the processor with a logic analyzer) or logical attacks (for example, the administrator of the machine 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 hesitant 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 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 [30] 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 [30]. However, the impact of this infrastructure on encryption and memory checking is not studied.

In this article we propose the CRYPTOPAGE extension of the HIDE infrastructure to provide, in addition to the protection of the address bus, memory encryption and memory checking. The novelty of this paper lies in this new way to combine these mechanisms with a very low performance overhead. We also describe how an untrusted operating system can take part in some of these mechanisms to decrease the hardware cost without compromising the security of the architecture.

The rest of this paper is organized as follows: Section 2 describes our proposition for implementing memory encryption and checking on the HIDE infrastructure; Section 3 presents information about the performance of this system

and Section 4 presents related work in this field.

2. Architecture

In this section, we first present the security objectives of our architecture. Next, we summarize the key concepts of the HIDE infrastructure required to understand our proposition and finally, we present our proposition to implement a memory protection system on this infrastructure.

2.1. Objectives of the architecture

The objective of our architecture is to allow the execution of secure processes. It must guarantee that these processes possess the two following 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 must be able to execute secure processes in parallel with other normal (non secure) processes with 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 spy on processor registers or to modify hardware contexts, *etc*.

However, the attacker cannot directly or indirectly probe the inside of the processor. In particular, we consider that timing attacks [13], differential power analysis (DPA [14]) attacks, *etc*. are avoided by other means beyond the scope of this article. Moreover, we do not consider denial of service attacks because they are unavoidable.

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. The plain HIDE infrastructure

Our proposition is based on the HIDE infrastructure so we will briefly summarize it here. The HIDE infrastructure, as described in [30], remembers 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. When 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 this line is locked, it cannot be replaced or written back to memory. When 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, [30] 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 [30]).

However, in [30], no information about memory encryption or memory protection is given. We now describe how to use the property that a line is written to memory and read from memory only once between each chunk memory space permutation to implement a cheap encryption and memory integrity checker on the HIDE infrastructure. First, we will describe how the encryption and verification are performed on cache lines and then, how the page information is protected.

2.3. Encryption and verification of cache lines

In the rest of this section, we assume that the chunks used by the HIDE infrastructure are the same as the pages used by the virtual memory mechanism to simplify the explanation but this assumption is not a requirement. We also introduce some notations:

- $\|$ is the concatenation of bit strings and $\oplus$ the bit-wise *exclusive or* (XOR);
- $L_{a,c} = L_{a,c}^{(0)} \| \ldots \| L_{a,c}^{(l-1)}$: the content of the cache line number a in the memory chunk c, divided into l blocks of the size of an encryption block;
- $E_K(D)$: the result of the encryption of the data block D with the symmetric key K;

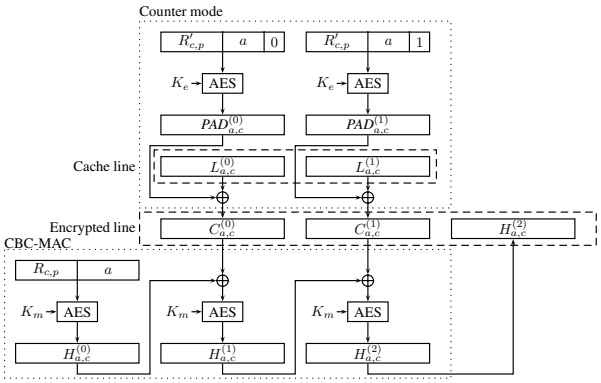


Figure 1. The encryption process of a cache line.

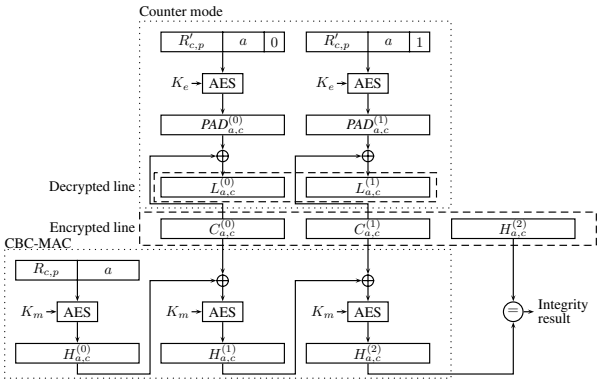


Figure 2. The decryption process of a cache line.

- K_e and K_m: the symmetric keys used to encrypt (and decrypt) the code and the data of a secure process (K_e) and to compute the *Message Authentication Codes* (MAC) used to authenticate the code and the data of a process (K_m). These keys are specific to a given secure process and are securely stored in its hardware context.

During each permutation of a given chunk, the processor chooses two random numbers, $R_{c,p}$ and $R'_{c,p}$ (where c is the corresponding chunk and p the number of the permutation), and stores them with the other information about the chunk (such as the permutation table).

After permutation, when a cache line is being written back to memory, the processor encrypts it, computes a MAC and stores $C_{a,c}\|H_{a,c}$ in memory with:

$$C_{a,c} = C^{(0)}_{a,c}\|C^{(1)}_{a,c}\|\cdots\|C^{(l-1)}_{a,c} \quad (1)$$

$$C^{(i)}_{a,c} = L^{(i)}_{a,c} \oplus PAD^{(i)}_{a,c} \quad (2)$$

$$PAD^{(i)}_{a,c} = E_{K_e}(R'_{c,p}\|a\|i) \quad (3)$$

$$H_{a,c} = H^{(l)}_{a,c} \quad (4)$$

$$H^{(i)}_{a,c} = E_{K_m}(C^{(i-1)}_{a,c} \oplus H^{(i-1)}_{a,c}), \quad i \in [1, l-1] \quad (5)$$

$$H^{(0)}_{a,c} = E_{K_m}(R_{c,p}\|a) \quad (6)$$

The cache line is encrypted using the counter mode [24] (equation 2). The pads used depend on $R'_{c,p}$ and a (equation 3). Equations 4 to 6 define a CBC-MAC [23, 11][1] computed over the encrypted cache line, $R_{c,p}$ and the address a of the cache line. The encryption mechanism is summarized in Figure 1.

The counter mode is known to be secure against chosen-plaintext attacks [2] on the condition that the counter is only used once with the same encryption key. Indeed, if two lines $L_{a,c}$ and $L'_{a,c}$ are encrypted using the same pad $PAD_{a,c}$, we have the relation $C_{a,c} \oplus C'_{a,c} = (L_{a,c} \oplus PAD_{a,c}) \oplus (L'_{a,c} \oplus PAD_{a,c}) = L_{a,c} \oplus L'_{a,c}$ so we can obtain information about the content of the two cache lines by simply comparing the two encrypted cache lines. Between two permutations, the cache line at the address a is encrypted only once, so the pad $E_{K_e}(R'_{c,p}\|a\|i)$ is only used once, except if the same random number $R'_{c,p}$ is chosen during two different permutations.

If $R'_{c,p}$ is 119-bits long [2], the birthday paradox says that the probability of a collision is high ($> 1/2$) after $\sqrt{2^{119}} = 2^{59.5}$ random draws. So after $2^{59.5}$ permutations, the risk of collision, and so of using the same encryption pad twice, is high. Even if the processor was able to perform a chunk permutation every cycle at 1 GHz, it would take about 25 years on average to find a collision, so this is not a critical security issue. In addition, to perform an attack, the attacker would have to know when the collision happened and this is not possible if $R'_{c,p}$ is encrypted in the page information data structure.

However, the counter mode does not provide any integrity protection. It is very easy to modify plaintext without being detected so we have to use another mechanism. In our proposition, the integrity protection (provided by the MAC function) is applied on the encrypted data. This scheme is called *Encrypt-Then-MAC* in the literature. Bellare *et al.* have proved in [3] that this construction does not decrease the confidentiality of the encrypted data and that, if the MAC function used is sufficiently strong, it guarantees the integrity of the data.

Before using a line read from the memory, the processor needs to decrypt and verify it. To decrypt the line, the

[1] We have chosen to use CBC-MAC because it is relatively fast and because the encryption and the integrity protection can use the same hardware, but any other good MAC function can be used instead.

[2] If we use the following parameters: the encryption algorithm is AES whose block size is 128 bits; the size of a cache line is 32 bytes (256 bits), so i is 1-bit long; the size of a memory page is 8 kB, so the number a of the cache line in the page is 8-bits long.

processor computes the pads and performs a bit-wise XOR between the content of the line and the pads (the inverse of equation 2). The pads can be computed during the memory access as they only depend on $R'_{c,p}$, a and K_e. If the time needed by the encryption unit to compute these pads is less than the time needed by the processor to fetch a line from the memory, the pads are ready before the encrypted data arrives and so, the end of the decryption process is performed in only one cycle (the XOR operation). To verify the integrity of the line, the processor computes the MAC (with equations 4 to 6) and compares it to the value $H_{a,c}$ read from the memory. If they are identical, the line has not been corrupted. This decryption mechanism is summarized in Figure 2.

This mechanism can prevent three attacks from happening. First, an attacker cannot inject a modified value in memory because he would need to compute the correct MAC for it so, if the MAC function used is strong, he would need to know the key K_m and this is not the case. The attacker also cannot copy a line and its MAC to somewhere else in memory because the MAC depends on the virtual address line. Finally, if the attacker cannot replay the value of the random number $R_{c,p}$, he cannot perform replay attacks[3] because the MAC depends on $R_{c,p}$ which changes between each permutation.

2.4. Protection of chunk information

Therefore, to prevent replay attacks, we need to prevent $R_{c,p}$ from being replayed. To do this, we protect the data structure containing the information about a chunk ($R_{c,p}$, $R'_{c,p}$, the permutation table, *etc.*), which is stored in an extension of the Translation Lookaside Buffer (TLB), with a Merkle tree [20].

The principle of this mechanism is to build a tree over all the page data structures. The leaves of the tree are the data structures themselves. The nodes of the tree contain a cryptographic hash of the content of their children. The root of the tree is stored in a secure memory inside the processor so it cannot be altered or replayed by an attacker. When the processor updates a chunk data structure (after a permutation for instance), it updates the values of the ancestor nodes up to the root. When the processor reads a chunk data structure (during a TLB miss), it checks the values of the ancestor nodes up to the root.

If we want to protect an area of n (with n a power of 2) memory elements (in our case, page data structures) represented by $b_{\log_2 n,0}$ to $b_{\log_2 n,n-1}$ (see Figure 3), the algorithms used to perform a verified read ($\mathcal{R}_{\mathcal{V}}(i,j)$) or write ($\mathcal{W}_{\mathcal{V}}(b_{i,j},i,j)$) are given in Figures 4a and 4b [15, 6], where H is a one-way hash function, $\mathcal{R}(i,j)$ the function that returns the value of the node $b_{i,j}$ from memory (except for $b_{0,0}$ which is securely stored in a special part of the processor), $\mathcal{W}(b_{i,j},i,j)$ the function that writes the value of $b_{i,j}$ to memory (except for $b_{0,0}$).

[3] A replay attack involves saving a value and its MAC and replaying them later at the same place in memory.

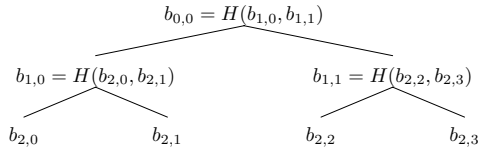


Figure 3. Merkle tree.

$\mathcal{R}_{\mathcal{V}}(i,j)$:
  $b_{i,j} = \mathcal{R}(i,j)$
  while $i > 0$:
    $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$
    $b_{i,f} = \mathcal{R}(i,f)$ { *Read the sibling node* }
    $b_{i-1,p} = \mathcal{R}(i-1,p)$ { *Read the parent node* }
    if $b_{i-1,p} \neq H(b_{i,\min(j,f)}, b_{i,\max(j,f)})$:
      error
    $i = i - 1$; $j = p$ { *Continue verification above* }
  return $b_{i,j}$

(a) Verified read algorithm.

$\mathcal{W}_{\mathcal{V}}(b_{i,j},i,j)$:
  if $i > 0$:
    $f = j \oplus 1$; $p = \lfloor \frac{j}{2} \rfloor$
    $b_{i,f} = \mathcal{R}_{\mathcal{V}}(i,f)$ { *Read & verify the sibling node* }
    $b_{i-1,p} = H(b_{i,\min(j,f)}, b_{i,\max(j,f)})$
    $\mathcal{W}_{\mathcal{V}}(b_{i-1,p}, i-1, p)$ { *Write & verify the parent node* }
  $\mathcal{W}(b_{i,j},i,j)$ { *Write the node* }

(b) Verified write algorithm.

Figure 4. Verified read and write algorithms.

This Merkle tree prevents an attacker from replaying the values of a chunk data structure and so prevents him from being able to replay data in memory. To reduce the time needed to perform the check of the Merkle tree during a TLB miss, we use a small cache that stores some nodes of the tree. During a check operation, the processor can stop as soon as the node of the tree that it is fetching is in this cache. Indeed, in order to be in the cache, a node must have been checked during a previous read operation and, as the cache is inside the processor, it cannot be altered by an attacker, so a node in the cache is necessarily correct. It is also a good idea to store a node and its sibling node together in the cache because, when we read or write a node, we will always need the sibling node.

These data structures also need to be encrypted to guarantee that an attacker cannot access the permutation table, which would destroy the information leakage protection. So

these structures are encrypted with the secret key K_p only known by the processor using a symmetric algorithm, for instance, with the AES used in the cipher-block chaining (CBC) mode with a random initialization vector (IV).

2.5. Management of the Merkle tree

In Section 2.4, we have seen that we have to maintain a Merkle tree over the data structures that store the chunk information. We also have to implement the verified read and write algorithms that check the integrity of these data structures using the Merkle tree.

To reduce hardware modifications, we want to delegate the loading of the secure chunk data structures and the storage of the Merkle tree needed for the global certification to the operating system without allowing the operating system to break the security of the running process.

The hardware provides the operating system with two new special instructions that can only be executed in the privileged mode: `LoadNode` and `HashCheck`, and a special buffer, called the *verification buffer* (VB) which can contain exactly $n-1$ double tree nodes (where $n = \log_2(\textit{maximum number of pages})$).

The first instruction, `LoadNode` takes three parameters: the physical address where the value of the node $b_{i,j}$ is stored, the depth i of this node and the horizontal position j of this node in the Merkle tree. This instruction first checks if the node $b_{i,j}$ is already available in the Merkle tree cache. If so, it does nothing except to set a special flag in the processor's status register to inform the operating system. Otherwise this instruction reads the node $b_{i,j}$ and the sibling node $b_{i,j\oplus 1}$ located at the given physical address and stores them together in line number i of the VB.

The second instruction, `HashCheck` takes only one parameter: the number of the line to check in the VB. This instruction hashes the content of the nodes $b_{i,j}$ and $b_{i,j\oplus 1}$ stored in line number i of the VB and compares the result with the parent node $b_{i-1,\lfloor \frac{j}{2} \rfloor}$ which must be already stored in the Merkle tree cache (and not in the VB). If the hash and the parent node are equal, the nodes $b_{i,j}$ and $b_{i,j\oplus 1}$ in the VB are correct so the instruction copies them to the Merkle tree cache. If the hash and the parent node are different, or if the parent node is not already in the Merkle tree cache (not the VB), the instruction triggers a security exception and aborts the execution of the secure process.

With these two instructions and the VB, some of the operations required to perform the verified read can be performed by the operating system. When the processor needs information about a page p that is not already available in the TLB, the processor generates a special exception to inform the operating system that it needs to fetch and check $b_{n,p}$. Then the operating system executes the algorithm shown in Figure 5. At the end, the operating system restarts the execution of the process. The processor expects the presence of $b_{n,p}$ in the Merkle tree cache in order to continue the execution of the process.

The load and verification algorithm given in Figure 5 performs the following operations. First, it loads into the VB, using the instruction `LoadNode`, the nodes located in the path from the page information data structure to the root of the Merkle tree, by beginning at the bottom of the tree. As soon as it tries to load a node that is already in the Merkle tree cache, it begins the verification of the nodes from the missing node, down to the bottom of the tree. If d is the depth of the last missing node in the cache, the algorithm executes the instruction `HashCheck` d which verifies that $H(b_{d,2i}, b_{d,2i+1}) = b_{d-1,i}$. If this verification succeeds, the two nodes at depth d are correct, so the instruction `HashCheck` can move them from the VB to the Merkle cache. The algorithm performs this operation several times, up to the leaves of the tree. If no security exception was raised during verification, at the end of the algorithm, the node requested by the processor is in the Merkle tree and so the secure process can restart.

The Merkle tree cache and the VB are also automatically invalidated when the processor switches from one secure process to another. With this solution, the operating system can choose by itself the best way to store and to manage the Merkle tree as it is the operating system that tells the processor where to find information about a page and the nodes of the tree.

2.6. Other security aspects

Several other aspects required to implement our architecture are not studied in this paper because they have been studied elsewhere. For instance, the confidentiality and the integrity of the hardware context of a secure process have to be maintained during an interrupt or while another process is running. The operating system, or any other process, should not have access to the state of the registers during an interrupt. In addition, the creation and the loading of an encrypted binary is also not described here. All these points have been described in depth in several articles [26, 17, 16, 5, 30].

When an interrupt occurs, the hardware context (which contains the registers, the encryption keys, the root of the Merkle tree, *etc.*) of the current running secure process is saved into one of several hardware context buffers (HCB) inside the processor. Next, all the registers are blanked in order to prevent information leakage. When the operating system wants to restart the secure process, it executes a special instruction which restores the state of the process from a given HCB. As the operating system cannot read or modify the content of the HCB, it cannot disturb or spy on the process. If there are several secure processes running, the

```
LoadPageInfo(p) :                                   { The processor needs a secured TLB of the memory page p }
   { Find the branch of the tree to check from the bottom: }
   d = n; q = p
   while d > 0 :
      A_{d,q} = getNodeAddress(d, q)                { The OS finds the address of the node by itself }
      LoadNode A_{d,q}, d, q                        { Asks the processor to load a node in the VB }
      if b_{d,q} is already in the cache:
         break                                      { We find a node that is already certified in the cache }
      d = d − 1; q = ⌊q/2⌋                          { We climb the tree }
   d = d + 1                                        { We descend to the first node missing }
   { We descend to verify the branch: }
   while d ≤ n :
      HashCheck d          { Asks the processor to verify the node in the VB and if it is correct to move it to the cache }
      d = d + 1                                     { Descends to the next missing node }
   ReturnFromInterrupt { Restarts the execution. The TLB is missing in case of an attack so denial of service is detected }
```

Figure 5. Verification by the OS of the data of a memory page.

operating system can ask the processor to encrypt the content of one HCB and can save the result of the encryption to memory. It can also load one HCB with an encrypted hardware context and ask the processor to decrypt it in order to restart the execution of the corresponding secure process.

To allow system calls, the process can choose not to blank or restore one or several registers for the next interrupt. The process can also specify the address of a signal handling function where the operating system can ask the processor to jump to with a special instruction, without triggering a security exception. To execute a secure program, the operating system copies the initial hardware context which is encrypted using the public key of the processor to a special buffer inside the processor which can decrypt it and start the execution.

3. Evaluation and results

In this section, we use the same architectural parameters as [30] to evaluate the performance of our proposition. They are summarized in Table 1. The AES unit is fully pipelined and its latency is 11 cycles.

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	4-way associative, 1 MB, LRU
L2 miss latency	12 cycles
ITLB	4-way associative, 64 entries, LRU
DTLB	4-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

Table 1. Architectural parameters used in simulations.

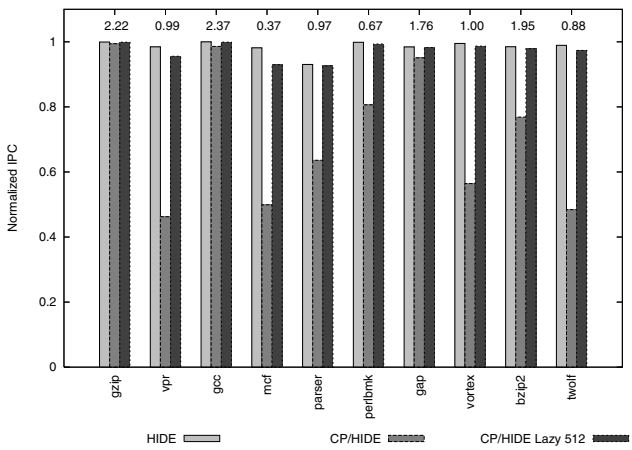
3.1. Theoretical analysis

Figure 6 describes the chronology of a read operation when there is a cache miss under the hypothesis that the information about the chunk is already available in the TLB. The slowdown due to the decryption and the verification of the line is 13 % (107 cycles instead of 95 cycles without encryption and verification).

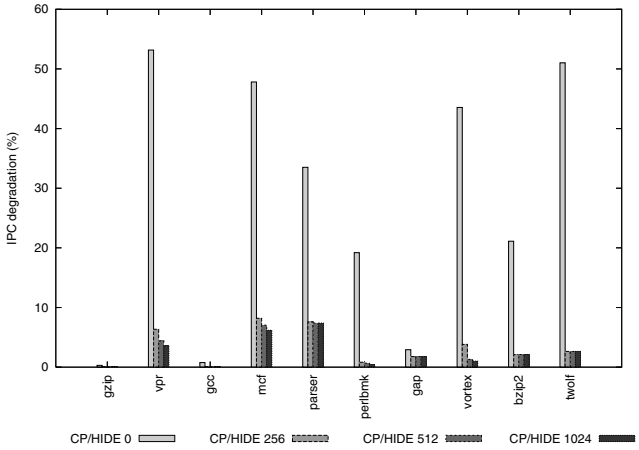
However, the decryption by itself takes only one cycle in this example. So the line can be used in a speculative way as soon as it is decrypted (with, in this case, a slowdown of only 1 %). The processor should avoid any security side-effect based on the results of this line (such as a non-encrypted write to external memory), before the integrity of the line has been checked.

During a TLB miss, the processor needs to check the Merkle tree that protects the information concerning the page. If the secure process uses its full 32-bit address space (4 Gbytes), the depth of the Merkle tree would be 19, so, in the worst case, the processor would need to compute 19 hashes and it would take 1,520 cycles[4]. However, this operation is not very frequent. In addition, the values of the intermediate nodes in the tree are cached to reduce the length of the path to check.

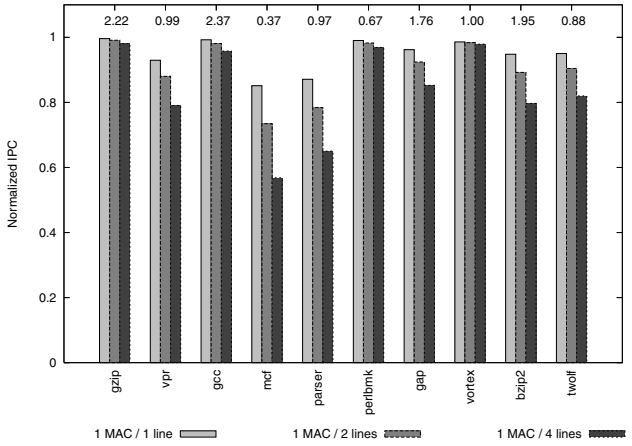
[4]By using an SHA-1 hash implementation running in about 80 cycles.



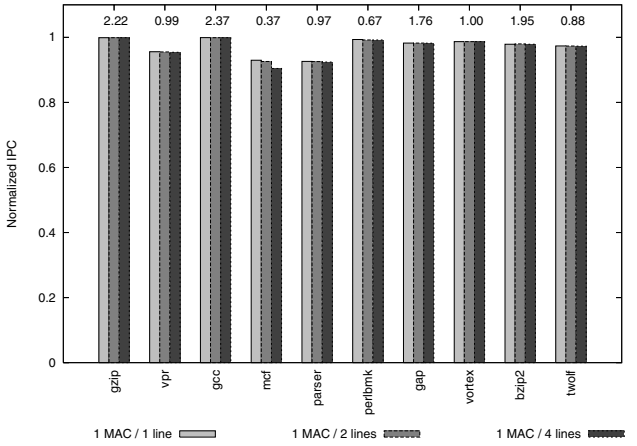
(a) Comparison for three architectures.



(b) Slowdown comparison for four Merkle cache sizes.



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



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

Figure 7. Simulation results.

3.2. Evaluation

To evaluate our propositions, we ran several SPEC2000int [9] benchmarks with the SimpleScalar [1] out-of-order simulator modified to implement our architecture. To reduce the time needed to perform the simulations, we skipped the first 1.5 billion instructions and we performed a detailed simulation for 200 million instructions.

In Figure 7a, we compare the instructions per cycle (IPC) ratio of each benchmark for three different implementations:

- our implementation of the HIDE [30] infrastructure (8K chunks, all chunks protected, no layout optimizations)
- 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 gives 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 gives good results. The average slowdown is only 3 % and the worst is 7.4 % (with the benchmark `parser`).

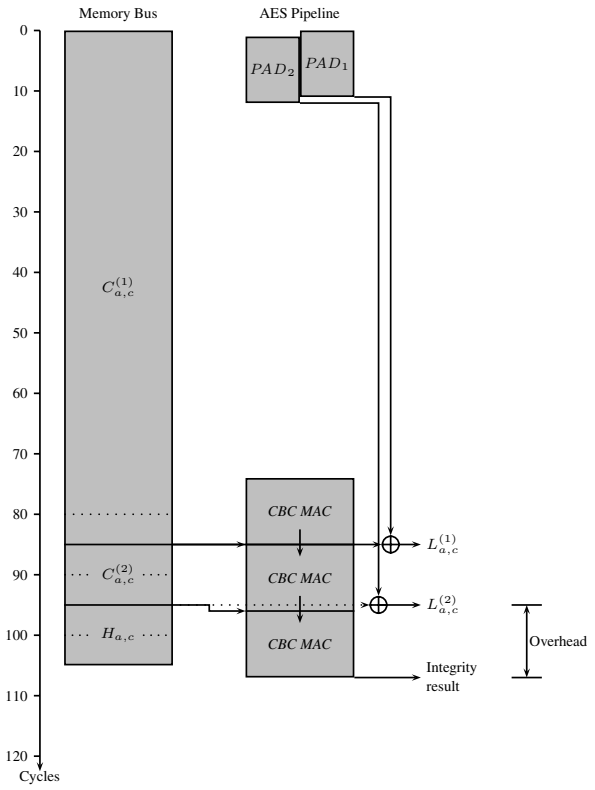


Figure 6. Timeline example of a cache line read from memory with the parameters from Table 1.

In Figure 7b, 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 1024-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 utilization. Figure 7c 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 verification, the performance penalty is very small (less than 1 %, as shown in Figure 7d).

We used SimpleScalar 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.

4. Related work

In this article, we have proposed a mechanism to implement an on-line memory checker using the HIDE infrastructure. In [7], Gassend *et al.* used a Merkle tree, computed over the whole memory space, to implement an on-line memory checker. They also used the cache hierarchy of the processor to improve the speed of the integrity checking by storing parts of the tree in the cache which is assumed to be secure. Even with the use of the cache, they get a performance penalty of 20 %. In our proposition, we combine a simple MAC mechanism and a Merkle tree computed over the page information (and not the whole memory) to build a memory checker, secure against replay attacks, with a better performance penalty.

In [7], Gassend *et al.* built an off-line memory checker using incremental hash functions. In [4], Clarke *et al.* proposed another off-line memory checker with a constant bandwidth overhead by combining hash tree and 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 verification function has to be called before each security-critical instruction. But between each verification, the instructions are executed without verification and this can cause some information leakages if this mechanism is combined with memory encryption.

The memory encryption in secure architectures has also been studied alone in several secure architecture such as XOM [16, 17, 18], AEGIS [26, 27] and CRYPTOPAGE [15, 6, 5]). For example, in [25], Shi *et al.* also used the counter mode to encrypt cache lines. However, they need to store the counter in memory, so the computation of the pads during memory access is delayed until the counter is fetched from memory. To reduce this problem, they proposed to use a prediction unit and a special cache to try to guess the correct counter required to decrypt a line. In our proposition, we also use the counter mode but the counter is deduced

from data stored in the page information structure. As this structure is always available before memory access, except if there is a TLB miss, the pads can be computed during the memory access latency, without relying on a counter cache or a counter prediction unit.

Finally, the problem of information leakage on the address bus has been studied in several works. One approach is to encrypt the address bus, which is equivalent to performing an initial permutation of the memory space. For instance, this mechanism is used in DS5000 processors. However, as this encryption does not change during the life of a process, an attacker can still see that one piece of memory is more often used than another. Goldreich *et al.*, in [8], proposed several approaches that guarantee that no information leakages can happen on the address bus but their algorithms entail a high performance penalty. Zhuang *et al.* created an infrastructure called HIDE which prevents information leakages on the address bus with a low performance penalty (1.3 % slowdown).

Recently, Yan *et al.* in [29] developed a new method to combine memory encryption and integrity protection with an IPC overhead of 5 % by using the Galois/Counter Mode of operation (GCM [19]). The encryption process is similar to our proposition, except that they still need a counter cache whereas we always have the correct counter ready before the memory access. The integrity protection is based on the GCM which is faster than our CBC-MAC because most of the computation required can be overlapped with the memory latency. However, they still use a Merkle tree computed over the whole memory and they do not provide information leakage protection.

We also have to mention the efforts of the Trusted Computing Group [28] and of Microsoft with its project *Next Generation Secure Computing Base* (NGSCB) [22, 21] 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.

5. Conclusions and future work

In this work, we described a new simple solution to implement a cheap memory encryption and protection mechanism in processors, based on the HIDE architecture and recursive hash functions, 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. We proposed to improve the performance even more with the help of a small verification cache. A few novel secure instructions are added to delegate securely the security management to the operating system and simplify the hardware secure mechanism.

Once the security is granted at the TLB level, some secret keys are ciphered 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.

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 reached thanks to the property that one line of memory is read and written only once between each chunk permutation.

We are working to extend our proposition to multicore and multiprocessor systems and to apply our architecture to secure distributed high performance computing.

6. Acknowledgment

This work is supported in part by a PhD 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] T. Austin, E. Larson, and D. Ernst. SIMPLESCALAR: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, Feb. 2002.

[2] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science (*FOCS*'97)*, pages 394–403. IEEE Computer Society, Oct. 1997.

[3] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Springer-Verlag, editor, *Advances in Cryptology - Asiacrypt 2000 Proceedings*, volume 1976 of *Lecture Notes in Computer Science*, 2000.

[4] D. Clarke, G. E. Suh, B. Gassend, A. Sudan, M. van Dijk, and S. Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 139–153. IEEE Computer Society, May 2005.

[5] G. Duc. 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`.

[6] G. Duc, R. Keryell, and C. Lauradoux. CRYPTOPAGE : Support matériel pour cryptoprocessus. *Technique et Science Informatiques*, 24:667–701, 2005.

[7] B. Gassend, G. E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (*HPCA*'03)*, pages 295–306, Feb. 2003.

[8] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[9] J. L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[10] A. Huang. Keeping secrets in hardware: the Microsoft XBox (TM) case study. Technical Report AI Memo 2002-008, MIT, May 2002.

[11] ISO/IEC 9797. Data cryptographic techniques — data integrity mechanism using a cryptographic check function employing a block cipher algorithm, 1989.

[12] R. Keryell. CRYPTOPAGE-1 : vers la fin du piratage informatique ? In *Symposium d'Architecture (*SYMPA*'6)*, pages 35–44, Besançon, June 2000.

[13] P. C. Kocher. 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)*, number 1109, pages 104–113. Springer-Verlag, Aug. 1996.

[14] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology (*CRYPTO*'99)*, number 1666, pages 388–397. Springer-Verlag, Aug. 1999.

[15] C. Lauradoux and R. Keryell. CRYPTOPAGE-2 : un processeur sécurisé contre le rejeu. In *Symposium en Architecture et Adéquation Algorithme Architecture (*SYMPAAA*'2003)*, pages 314–321, La Colle sur Loup, France, Oct. 2003.

[16] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. 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*)*, pages 168–177, Oct. 2000.

[17] D. Lie, C. A. Trekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (*SOSP*'03)*, pages 178–192, Oct. 2003.

[18] D. J. Lie. *Architectural support for copy and tamper-resistant software*. PhD thesis, Stanford University, 2004.

[19] D. A. McGrew and J. Viega. The galois/counter mode of operation. Submission to NIST Modes of Operation Process, May 2005. `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes`.

[20] R. C. Merkle. A certified digital signature. In *Proceedings on Advanced in Cryptology (*CRYPTO*'89)*, volume 435, pages 218–238. Springer-Verlag New York, Inc., 1989.

[21] Microsoft Corporation. NGSCB*: Trusted Computing Base and Software Authentication*, 2003. `http://www.microsoft.com/resources/ngscb/documents/ngscb_tcb.doc`.

[22] Microsoft Corporation. *Security Model for the Next-Generation Secure Computing Base*, 2003. `http://www.microsoft.com/resources/ngscb/documents/NGSCB_Security_Model.doc`.

[23] NIST. Computer data authentication, May 1985. Federal Information Processing Standards Publication 113.

[24] NIST. Recommendation for block cipher modes of operation, Dec. 2001. Special Publication 800-38A.

[25] W. Shi, H.-H. S. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (*ISCA*'05)*, pages 14–24. IEEE Computer Society, June 2005.

[26] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing (*ICS*'03)*, pages 160–171, June 2003.

[27] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. 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)*, pages 25–36. IEEE Computer Society, June 2005.

[28] Trusted Computing Group, Mar. 2005. `http://www.trustedcomputinggroup.org`.

[29] C. Yan, D. Englender, M. Prvulovic, B. Rogers, and Y. Solihin. Improving cost, performance, and security of memory encryption and authentication. In *Proceedings of the 33rd International Symposium on Computer Architecture (*ISCA*'06)*, pages 179–190. IEEE Computer Society, June 2006.

[30] X. Zhuang, T. Zhang, and S. Pande. 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*)*, pages 72–84. ACM Press, Oct. 2004.