# Added Redundancy Explicit Authentication at the Block Level for Parallelized Encryption and Integrity Checking on Processor-Memory Buses

Reouven Elbaz, Lionel Torres, Gilles Sassatelli

Laboratoire
d'Informatique
de Robotique
et de Microélectronique
de Montpellier
LIRMM

# Added Redundancy Explicit Authentication at the Block Level for Parallelized Encryption and Integrity Checking on Processor-Memory Buses

Reouven Elbaz, Lionel Torres, Gilles Sassatelli,

Avec la particiaption de : Pierre Guillemin, Michel Bardouillet and Albert Martinez, société STMicroelectronics

Technical Report
01/07/2007

# Added Redundancy Explicit Authentication at the Block Level for Parallelized Encryption and Integrity Checking on Processor-Memory Buses

**Abstract**— The bus between the System on Chip (SoC) and the external memory is one of the weakest points of computing systems because an adversary can easily probe this bus in order to read private data, to retrieve software code (data confidentiality concern) or to inject data (data integrity concern). The conventional way to provide data confidentiality and integrity is to implement a dedicated hardware engine for each security service. Being secured, this approach prevents parallelizability of the underlying computations. In this paper, we introduce the concept of *Added Redundancy Explicit Authentication* (AREA) *at the block level* and we describe a Parallelized Encryption and Integrity Checking Engine (PE-ICE) based on this concept. PE-ICE has been designed to provide an effective solution to ensure both security services while allowing for full parallelization on processor read and write operations and optimizing the hardware resources. Compared to standard encryption which provides only confidentiality, we show that PE-ICE additionally guarantees code and data integrity for less than 4% of run-time performance overhead and at no additional hardware cost.

**Index Terms**—Computer Architecture, Security, Integrity, Confidentiality, Encryption.

## 1. INTRODUCTION

PDAs, mobile phones, MP3 players, set-top box, digital video equipments become more and more widespread nowa-days. Also, the range of services provided by every single em-bedded system tends to widen rapidly and applications like on-line banking transactions, web browsing, email, application / game download become usual on mobile devices. As a conse-quence the amount of sensitive information such as private data – bank information, passwords, email, photos… – or intellec-tual property – software, digital multimedia content… – con-tained or transiting in those devices rapidly increases. The issue is that today's embedded systems are considered as untrust-worthy hosts [1] since the owner, or anyone else who succeeds in getting access, is a potential adversary. Thus, one of the chal-lenges for the high-technology industry in the development of pervasive computing relies on the development of secured computation and storage solutions.

The attacks conducted on embedded systems [2] challenge several security services such as data confidentiality, data integ-rity and system availability. Data confidentiality ensures that data stored in or transiting through embedded systems are only read by authorized parties while data integrity guarantees that those data are not tampered with, deleted or altered by mali-cious entities. Availability refers to the requirement of ensuring user access to the device without unexpected delay or obstacle.

Software attacks like viruses are the most famous threats be-cause they regularly affect our desktop computers. Viruses turned up in embedded systems in 2004 with the worm called Cabir which infected mobile phones running Symbian Operat-ing System (OS) and which propagated itself via Bluetooth. The industry started to work on this issue with the Trustzone Project [3] from ARM and through consortiums such as the Trusting Computing Group [4] (TCG, formerly Trusting Computing Platform Alliance, TCPA) whose goal is to define secured proc-essing architectures. However, all these efforts do not consider hardware-based (physical) attacks and work under the assumption that the communication channels between the processor chip and the other components are secure despite the fact that data exchanges are often done in clear form. The well known cracking of the Xbox gaming console shows that designing computing systems with such an

assumption leads to simple physical attacks. In [5], the hacker Andrew "bunnie" Huang, explained his approach to break the Xbox security features and demonstrated that one of the weakest points of computing systems are buses because they offer a low-cost spot for attacks.

Thus, in this work we focus on physical non-invasive attacks – or board level attacks – conducted on buses between the Sys-tem on Chip and off-chip volatile memory or directly in the memory – typically Random Access Memory (RAM). The objectives of the adversary can be the unauthorized use or the illegal distribution of intellectual properties or – concerning end users – to discover or to corrupt private data retrieved on buses or directly in memory. Thus, our goal is to ensure the confidentiality of the off-chip memory content during storage or execution to prevent the leakage of any sensitive information and its integrity to forbid execution of intentionally altered data.

Smartcards offer a countermeasure against such attacks by putting all processing and storage elements in a single chip. Another common solution is secure co-processors which encapsulate the components handling sensitive computations and data in a tamper-resistant and tamper-responsive package, such as the IBM 4578 [6]. However, these solutions are not suited for embedded systems because the latter requires an expensive and large package to provide a high performance system while the former does not allow storing a large amount of code and data and does not offer a high computing power.

A trade-off between the above mentioned countermeasures is to limit the trust boundaries to the SoC and to embed memory protection apparatus on-chip. This concept was introduced by Best with bus-encryption microprocessor [7, 8] in 1979: data are encrypted before being stored off-chip and are only decrypted once back on-chip. However, encryption only ensures data confidentiality but does not provide tamper-detection mechanisms to guarantee data integrity. Later on, several re-search works [9, 10, 11, 12] considered this additional issue to offer a private and authenticated tamper resistant environment to software execution. They achieved this task by providing both security services – data confidentiality and integrity – separately. The shortcoming of such an approach is the serialization of the computation of the underlying cryptographic algorithms on write or on read operations introducing non-parallelizable latencies on off-chip memory accesses. More-over, the hardware resources needed are not optimized since the implementation of a dedicated engine for each security service is required.

In the present work, the goal is also to provide a private and authenticated tamper resistant environment to applications running in embedded processors to prevent physical attacks (board level attacks with bus probing and memory tampering). The designed hardware mechanisms must ensure the confidentiality and the integrity of the off-chip memory content while considering the constraints relative to the processor context – particularly random access of variable data size – to optimize hardware resources, memory access latencies and the memory bandwidth consumption at run-time.

In order to reach the above mentioned objectives, we explore added redundancy at the block level during block encryption to provide the data authentication service in addition to confidentiality. We call this technique Added Redundancy Explicit Authentication [13] at the block level (Block Level AREA). The hardware mechanism PE-ICE – for Parallelized Encryption and Integrity Checking Engine – implementing this concept, per-forms the encryption and the integrity checking of the external memory content with the following properties:

i) Full parallelization of the encryption and integrity checking process on off-chip write and read operations allowing latency optimization.

ii) Hardware optimization: Implementation of a single encryption algorithm to provide both security ser-vices: data confidentiality and integrity.

The rest of the paper is organized as follow. Section 2 presents our threat model. Section 3 describes the techniques pro-viding both data confidentiality and integrity and then the scheme implemented in a SoC for memory encryption and authentication. Section 4 defines the architecture of the proposed engine PE-ICE. In this section we show how to add the integrity checking capability to block encryption with the Block-level AREA technique. Moreover we describe PE-ICE processing on read and write operations, and we provide a security analysis of PE-ICE. Section 5 evaluates a SoC implementation of PE-ICE in terms of latencies introduced on memory accesses, of hardware

resources required for its SoC implementation and of run-time performance degradation. Moreover, we compare it to a generic composition scheme in the same simulation framework to highlight its advantages. Finally Section 6 concludes this work and describes current work led to further improve the concept of PE-ICE.

## 2. THREAT AND TRUST MODELS

The device to protect is supposedly exposed to a hostile environment where physical but non-invasive attacks are feasible. As a consequence, the main hypothesis in our threat model is the fact that the System on Chip (SoC) is considered as resistant to physical attacks. A justification of such an assumption is the Xbox cracking example where the hacker, Andrew "bunnie" Huang, succeeds his attack by targetting communication buses. In [5] he justifies his choice: an ASIC is too expensive and time-consuming to shape compared to connect a custom device to buses. Thus, the SoC is considered as a trusted area. In addition to this, side-channel attacks are not taken into account in this work. As a consequence, we consider that on chip registers and memories cannot be tampered with by an adversary. More-over, software attacks are not considered: the operating system (OS) or at least the OS kernel is also trusted.

In this work, we focus mainly on board-level attacks involving Processor-Memory (PM) bus probing or memory tampering. Such attacks allow the observation of the memory contents and the injection of arbitrary data on the PM bus or directly into the memory chip. We are particularly concerned by the "Man in the middle" attacks. The corresponding protocol implementing this attack is divided into two parts:

i) First the attacker monitors the PM communications and intercepts the data on the bus (passive attacks). Another possibility is to directly read data in memory. This first step raises the issue of data confidentiality.

ii) Then the adversary may insert chosen data on the PM bus (active attack) and thus challenges data integrity. The objective of the attacker could be to take control of the system by injecting malicious code or to constrain the search space in a key or message recovery attack. There are three classes of active at-tacks – also feasible when data are encrypted – defined with respect to the attacker's possible ways to choose the inserted data:

- *Spoofing attacks*: the adversary exchanges a memory block with an arbitrary fake one. The attacker mainly alters program behavior but cannot foresee the results of his attack if data are encrypted.
- *Splicing or relocation attacks*: the attacker replaces a memory block at address A with a block from address B, where A≠B. Such an attack may be viewed as a spatial permutation of memory blocks. When data are ciphered, the benefit for an attacker of using a memory block copy as a fake is the knowledge of its behavior if this one was previously observed.
- *Replay attacks:* a memory block located at a given ad-dress is recorded and inserted at the same address at a later point in time; by doing so, the current block's value is re-placed by an older one. Such an attack may be viewed as a temporal permutation of memory blocks at a specific address location.

In [14], a detailed description of a those attacks is proposed. Markus Kuhn shows in [14] how he successfully broke the encryption scheme implemented in the DS5002FP Microcontroller by using attacks belonging to this class.

## 3. MEMORY ENCRYPTION AND AUTHENTICATION: TECHNIQUES AND RELATED WORK

In this section, we present the existing techniques providing data confidentiality and integrity, then we briefly describe the related works proposed in our field of study and finally we highlight the main sources of run-time performance degradation introduced when encryption and integrity checking are implemented.

## 3.1 Techniques

In the rest of the paper, the underlying block cipher processes *b*-bit blocks under *k*-bit keys. $E_K$ and $D_K$ are respectively the encryption and the decryption functions under the key K. The plaintext message to encrypt P is divided into *m b*-bit plaintext blocks $p_i$ with ($1 \leq i \leq m$). Similarly, C the ciphered version of P ($C = E_K(P)$) is divided into *m n*-bit ciphertext blocks $c_i$ with ($1 \leq i \leq m$); where $c_i$ is the encrypted version of $p_i$ .

### 3.1.1 The Conventional Way: the Generic Composition

The conventional way to provide both data confidentiality and integrity is to pair a data authentication technique and an encryption mode, and therefore to perform two passes on data. A first pass is dedicated to encryption and a second one is done to compute a tag with a MAC (Message Authentication Code) algorithm. The three possible schemes defined in [15] are depicted in Figure 3-1:

- *Encrypt-then-MAC* (Fig.1a) encrypts the plaintext P into a ciphertext C, and then appends to C a tag T computed with a MAC algorithm over C.
- *MAC-then-Encrypt* (Fig.1b) calculates a tag over the plaintext P, appends the resulting tag to P and then encrypts them together.
- *Encrypt-and-MAC* (Fig.1c) encrypts the plaintext P to get a ciphertext C and appends to C a tag T computed over P.

The main drawback of such techniques is that both security mechanisms (encryption and MAC computation) are non-parallelizable on either read or write operations or on both. On write operations, for the Encrypt-then-MAC scheme, the tag computation starts only at the end of the encryption process while for the MAC-then-Encrypt, encryption only terminates after the completion of the tag calculation. Concerning read operations, for the Encrypt-and-MAC and the MAC-then-Encrypt schemes, the tag reference computation begins only when the decryption process is completed.

In [15], Bellare and al. proved that the most secure way to pair an authentication technique with an encryption mode is to use the Encrypt-then-MAC scheme and to enroll a different key for each computation.

### 3.1.2 AREA: Added Redundancy Explicit Authentication

The principle of AREA [13] schemes is to insert redundancy into the plaintext message before encryption and to check it after decryption. Such a scheme is constructed with cipher mode with infinite error propagation on encryption and on decryption (infinite two-way error propagation). A cipher mode has infinite error propagation on encryption if a ciphertext block $c_i$ can be expressed as a function of all previous plaintext blocks $p_i$ to $p_1$ of the message P. Similarly, a cipher mode has infinite error propagation on decryption if a plaintext block $p_i$ can be expressed as a function of all previous ciphertext blocks $c_i$ to $c_1$ in the encrypted message C. For instance, CBC (Ciphered Block Chaining [16]) has infinite error propagation on encryption but has limited error propagation on decryption since a given plaintext block can be expressed as a function of only two ciphertext blocks.

In order to authenticate a message in addition to encrypt it, a value $p_{m+1}$ – the redundancy – is

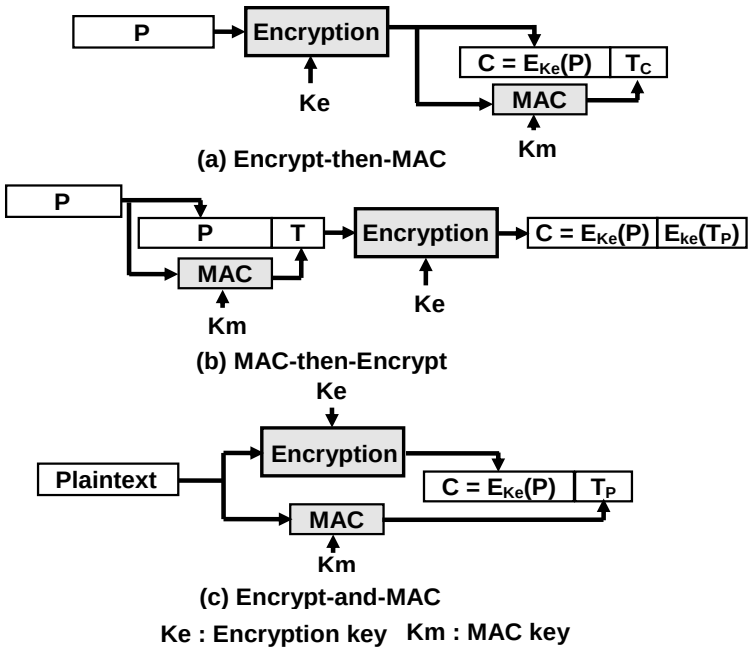



Fig. 1. The Generic Composition Schemes

appended at the end of the plaintext message before encryption. In this way, $c_{m+1}$ -the result of the encryption of $p_{m+1}$ - will depend on all plaintext blocks $p_i$ composing the message to authenticate. $p_{m+1}$ and $c_{m+1}$ are sent along with the encrypted message. This way, on decryption, the corruption of one bit in any ciphertext block $c_i$ will impact the decryption of $c_{m+1}$ since it depends on all previous ciphertext blocks. The recipient can detect a malicious modification by comparing the result of the decryption of $c_{m+1}$ with $p_{m+1}$. Note that, if the error propagation of the underlying cipher mode is not infinite on encryption, distinct messages leading to the same $c_{m+1}$ can be easily found. On the other hand, if the error propagation of the underlying cipher mode is not infinite on decryption, an adversary can modify or swap all blocks $c_i$ in C which are not impacted by the error propagation without detection.

AREA schemes seem really efficient since it is the only existing technique which performs only one pass over the data on encryption and decryption (and by requiring one encryption algorithm) to provide both data confidentiality and authentication. However, the infinite error propagation is usually achieved by chaining encryption (e.g. CBC, PCBC[17], and PCFB[18]) or decryption (e.g. PCBC, PCFB) operations, making parallelization impracticable.

### 3.1.3 Authenticated Encryption Modes

As shown in [15] and [17], providing both data confidentiality and authentication with schemes based on generic composition or on AREA could be a risky task. Hence, an important effort led by the cryptographic research community through the NIST's modes-of-operation [54] activities is deployed to define Authenticated Encryption (AE) modes. It aims at proposing a secure way to provide the confidentiality and authenticity security services to data. However, those encryption modes are based on AREA (PCFB) or on a generic composition schemes (GCM, CCM, OCB, IAPM…) and thus suffer from the same drawbacks.

## 3.2 Related Works

The works presented in this section aims at providing trusted processor architecture with different objectives. The AEGIS [10, 11] processor objective is to offer a trusted computing platform for a large set of applications with a special focus on distributed computation. XOM [9] aims to provide the same security features as the AEGIS processor with countermeasures against software and physical attacks. The objectives are also similar: software copy protection and tamper-resistant software distribution. SP [12] (Secret Protected) focuses on the protection of critical information such as cryptographic keys contained in mobile devices. Lee and al. propose the concept of concealed execution where a Trusted Software Module (TSM) handles the management of the sensitive information. TSM runs in a dedicated mode protecting its code and data from software attacks.

However, all those three projects need to protect the processor-memory transaction to thwart board level attacks. To fulfil this objective, they implement a generic composition scheme (Encrypt-then-MAC). Thus they all suffer from the drawback highlighted above: no parallelization, hardware expensive.

In AEGIS, XOM and SP the computation of tags is done over data block and address (addressed-MAC), this way providing a countermeasure against spoofing and splicing attacks. This ensures the authentication of Read Only (RO) data, but Read Write (RW) data are still sensitive to replay. XOM [19] failed in preventing replay attacks since it treats RW data as RO. Store hash of RW data block on-chip is a countermeasure against replay but is expensive in term of on-chip memory overhead. The hash tree [19] is a technique that allows reducing this on-chip memory overhead to only one hash. Suh and al. and Lee and al. implement hash algorithm technique combined with cached hash tree to efficiently thwart replay attacks. However techniques allowing to save on-chip memory consummed by replay attack countermeasures are not considered in this paper, for more details on those techniques, refer to [19, 20, 21]. In this work, we focus on the mechanisms which provide confidentiality and authentication services of off-chip memory content.

## 3.3 Run-Time Performance Degradation Considerations

In addition to the non-parallelizability of encryption and integrity checking, the run-time performance slowdown induced by the implementation of cryptographic algorithms, has mainly two sources:

- The intrinsic latencies of cryptographic functions involved in encryption and tag computation,
- The memory bandwidth pollution generated by the loading of meta-data such as tags and by the size of the atomic block loaded for decryption or for integrity checking. Such a block is called *chunk* in the following.

However, the processing of data by the security engines implies specific operations which represent another source of degradation. Whatever the size of the chunk a performance overhead must be expected on:

1) *Read operations of data smaller than a chunk*: such operations occur mainly for non-cacheable data and require to:

i) load the whole matching chunk from external memory with its tag
ii) decipher it and check its integrity
iii) forward the requested data to the CPU.

In addition to the latencies introduced by the security mechanisms, such a processing pollutes the memory bandwidth by loading data not necessarily needed.

2) *Write operations of data smaller than a chunk* require to:

i) load the matching chunk with its tag from off-chip
ii) decipher it and check its integrity
iii) modify the corresponding sequence in the chunk
iv) re-cipher it and re-compute its tag
iv) write it back into memory with its new tag.

This chain of operations is called in the rest of the paper a Read Modify Write (RMW). The additional performance slowdown implied by such an operation is mainly due to the generation of a read/decryption/checking process. Therefore, to reduce the run-time performance overhead introduced by RMW, the chunk size should ideally be defined as small as possible without affecting security.

The proposed Parallelized Encryption and Integrity Checking Engine, PE-ICE, is a mechanism dedicated to guaranteeing data integrity and confidentiality on off-chip read and write. The objective is to decrease the latencies introduced by the underlying cryptographic engine and its related hardware cost.

# 4. PE-ICE

In this section, we first describe the technique on which PE-ICE relies to provide data integrity and confidentiality using a single block encryption algorithm. Then we give a general overview of PE-ICE. The subsequent sub-sections deal with PE-ICE SoC implementation issues and provide a security analysis.

## 4.1 The Block-Level AREA Technique

The proposed technique relies on the diffusion property identified by Shannon [34] for block ciphers to be considered as secure. Theoretically a block cipher must be indistinguishable – from an adversary standpoint – from a random permutation with equiprobable outputs i.e. the redundancy in the statistics of the plaintext block $p$ has to be dissipated in the statistics of the ciphertext block $c$. Once a block encryption is performed, the resulting position and value of each bit in $c$ are a function of all bits of the corresponding $p$, thus providing the property of infinite error propagation at the block level.

In our scheme, we propose to leverage the diffusion property of block cipher to add the integrity checking property to this type of encryption. To do so, $p$ is composed of two fields: an $l_p$-bit field $P_L$ – hereafter called a payload – and an $t$-bit field T – hereafter called the tag – such as $b=t+l_p$ ($p = P_L||T$). Considering the diffusion property presented above, after encryption with a block cipher, it is impossible to identify the ciphered versions of $P_L$ and T within the ciphertext block $c = E_K(p)$. Moreover, if a $c$' is derived by flipping a single bit in $c$, there is a large probability that the last $t$-bit of the plaintext $p' = D_K(c')$ will be different from the content of T in $p$. This probability depends on the tag size $t$. The number of possible plaintext blocks with the same T resulting from the decryption of a tampered $c$ is equal to $2^{b-t}$. Hence the probability that T remains the same after decryption is $1/2^t$ ($= 2^{b-t}/2^t$). We call this authentication technique *block-level AREA*: the redundancy is added in each plaintext block before encryption and checked for each ciphertext block after decryption. The diffusion property of block cipher provides the two-way infinite error propagation at the block level.

## 4.2 General Overview

PE-ICE is the implementation of the proposed block-level AREA technique to authentify and encrypt data during off-chip memory operations. It is located on-chip between the last level of cache memory and the memory controller.

On write operations (Fig. 2a), a payload $P_L$ provided by the processor is concatenated with a

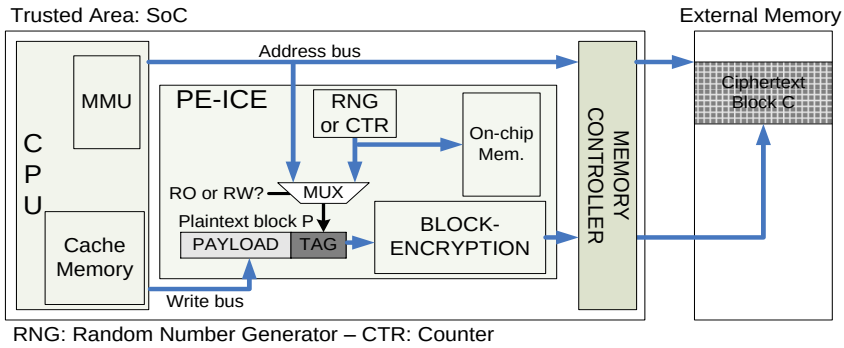
tag T to produce each plaintext block *p* to be processed by the block cipher. After encryption, the resulting ciphered block *c* is written in the external memory.

On read operations (Fig. 2b) *c* is loaded and decrypted. The tag T issued from the resulting plaintext block is compared to an on-chip re-generated tag called the tag reference T'. If T does not match T', it means – as explained in section 4.1 – that at least one bit of *c* has been modified during transmission on the bus or in the off-chip memory (spoofing attack), and PE-ICE raises an integrity checking flag to prevent further processing.

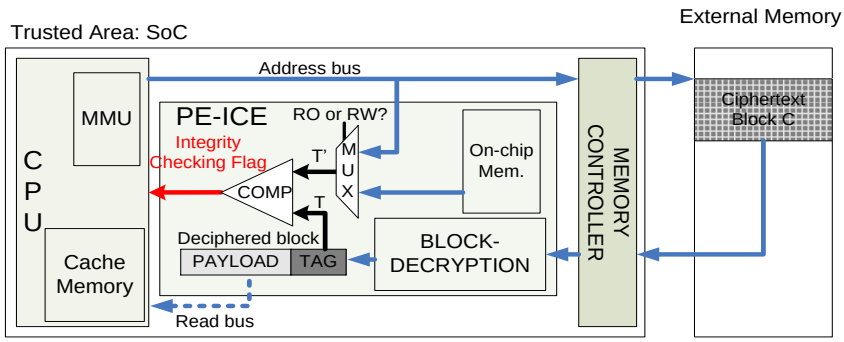
Thus the block encryption provides the data confidentiality service and the Block-level AREA technique allows for data authentication using the same block cipher.

The general overview of PE-ICE presented above assumes that we are able to re-generate a tag reference on read operations which must matches the tag retrieved after decryption. Moreover, this description of PE-ICE offers only a countermeasure against spoofing. That is why the next sub-section describes the tag generation and its composition which allows preventing replay and splicing attacks.

## 4.3 The Tag Generation




(a) Write Operations: Tag insertion and Block encryption




(b) Read Operations: Decryption and Integrity checking process (Tag matching process)

Fig. 2. PE-ICE: Encryption and Integrity Checking Process

In the processor-memory communication context, the SoC alone is performing both the encryption and the decryption. Therefore, the SoC has to hold the tag value T of each ciphered block between the encryption and the decryption or must be able to regenerate it on read operations to retrieve the correct reference tag. The challenge is to reach this objective by storing as little tag information as possible on the SoC to optimize the on-chip memory usage. The composition of the tag is different for each kind of data, RO (Read Only) and RW (Read Write), and depends on the properties of the respective data types.

### 4.3.1 Tag for Read Only data

RO data are only written once in external memory and are not modified at run-time. Therefore, such payloads are only sensitive to spoofing and splicing attacks. Thus, the tag contained in each plaintext block of RO data can be fixed for a payload $P_L$ stored at a given address. Moreover, it can be public because an adversary needs the secret encryption key to create an accepted ciphertext block $c=E_K(P_L||T)$. This secret key is stored in an on-chip register where storage is trusted. However, an adversary must not be able to choose the reference tag T' or to influence its generation. Hence, PE-ICE uses the address of the ciphered block as a tag (Fig.2a and 3a). If an attacker performs a splicing attack, the address used by the processor to fetch a block and by PE-ICE to generate the reference tag T' will not match the last *t*-bit (T) of the plaintext issued from the decryption of the fake ciphered block (Fig.2b).

### 4.3.2 Tag for Read Write data

RW data are modified during software execution and are consequently sensitive to replay attacks. Using only the address as tag does not prevent such an attack because the address will not relate changes between write operations at run-time at a given location in memory and thus, the processor cannot verify that the data stored at a given address is the most recent one (temporal permutation). For that reason the tag T must be changed on each off-chip write operation; this can be achieved in two different ways:

1) T is a nonce, a Number used ONCE, which can be simply generated with a counter (T = CTR) incremented on each write operation (Fig. 2a). The use of a nonce as tag prevent replay and splicing since the processor never produces two ciphertext blocks using the same tag. However, when the counter reaches its limit we must change the encryption key and re-encrypt the corresponding memory region for T to be a true nonce. Otherwise, an adversary can perform what we call a *periodic replay/splicing attack*: he can record a ciphered block *c* in memory and predict when *c* can be replayed or relocated by counting the number of store operations performed by the processor. Each time the number of store is a multiple of the period[1] of the counter, a periodic replay/splicing of *c* succeeds at the targeted address by the processor.

2) Re-encryption requirement can be inconvenient and frequent particularly when we choose a small counter size. Thus to avoid re-encryption we propose an alternative solution where T is generated with a random value generator assumed embedded on-chip (T = RV; Fig. 2a and Fig. 3b). Using a random value provides unpredictability and thus prevents the periodic replay/splicing attack to be successful: the tag value is unpredictable from an adversary's standpoint, so he is unable to know when two encrypted blocks have the same tag and thus when to perform a periodic replay/splicing. However, since a random value is not a nonce, this implies that replay and splicing attacks may succeed. With random values, the security relies on the difficulty for an adversary to find two ciphered blocks processed by PE-ICE (at the same address for a replay and at different address for a splicing) with the same random values. The probability to overcome this difficulty is the same for replay and splicing attacks and is defined in section 4.1. This point is further developed in section 4.6 which presents the PE-ICE security analysis.

Regardless of the tag composition, the SoC must be able to retrieve the correct random or counter values – called in the following the reference random values RV' or reference counter values

CTR' – to generate the reference tag T' during the integrity checking process. On the other hand, the set of RV' and of CTR' must be tamper-proof, otherwise the attacker could choose the data to replay by replaying it with the corresponding RV' or CTR'. In order to solve this issue the random or counter values generated on write operations are stored on-chip (Fig. 2a) as reference values in a dedicated memory. This way, these values are tamper-proof since the SoC is considered as trusted and can be easily retrieved on read operations (Fig. 2b).

The size of RV or CTR fixes a trade-off between the strength of the countermeasure against replay and the on-chip memory overhead. An alternative tag configuration of the tag is proposed for RW data in which the least significant bits of the address addressing each ciphered block are concatenated with an RV or a CTR (T=RV||ADD - Fig. 3c - or T=CTR||ADD). Such a configuration decreases the strength against replay (by decreasing the size of the value which changes on each off-chip store) but maintains a countermeasure against splicing and reduces the on-chip memory cost.

However, the on-chip memory cost may remain too high so we developed tree techniques to allow for the secure external storage of the set of RV' [20] or CTR' [21]. This paper focuses on providing data confidentiality and integrity; for more details on those tree techniques, refer to the corresponding references.

For sake of clarity, in the following PE-ICE is described only for tag composition using random values (fig.3b and c).

## 4.4 Encryption Mode and Chunk Definition

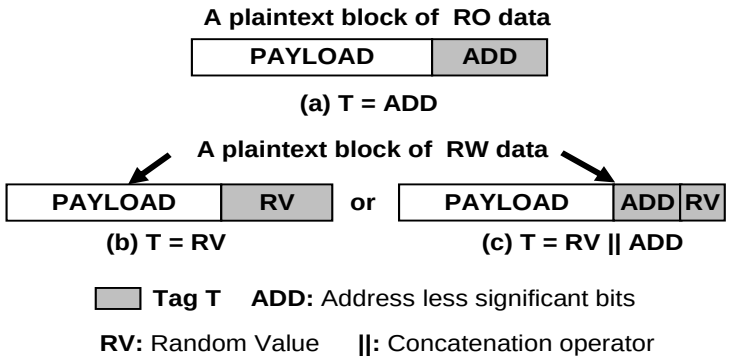



Fig. 3. Plaintext blocks and tag composition before encryption

As highlighted in Section 3.3 the definition of the granularity of encryption and of integrity checking – the chunk size – is essential to limit the memory bandwidth pollution. The chunk must be as small as possible. When block encryption is implemented the smallest possible chunk size is that of the ciphered block if encryption is done in ECB (Electronic Code Book). Thus, PE-ICE implements ECB mode. Such a mode also enables per-block integrity checking in PE-ICE since the tag is generated independently for each plaintext block. This is the finest granularity of integrity checking allowed by PE-ICE; this way, when a RMW is required, only the matching ciphered block is loaded for decryption and integrity checking. Hence, in the following, the chunk size is defined as the length of a block processed by the block cipher implemented in PE-ICE.

## 4.5 Physical Address vs Virtual Address Space Protection

[1] The period of a counter is defined by the number of distinct value it generates (e.g. for a *ctr*-bit counter the period is of $2^{ctr}$).

In XOM [22], the OS is untrusted and responsible for controlling the MMU (Memory Management Unit); hence, an adversary can choose a block to load from memory by modifying the entries of the table which manages the virtual/physical address translation. Thus, he performs a splicing attack within the virtual address space. To prevent this attack the tag must be produced from the virtual address; otherwise if the tag is generated from the physical address, PE-ICE will not detect splicings because from its point of view, the data is fetched from the correct physical address.

The main advantage of using the virtual address in tags comes from the fact that the application can be loaded in memory already encrypted off-line by PE-ICE. However, the use of the virtual address requires some deep processor core modifications. As mentioned above, PE-ICE is localized between the cache memory and the memory controller. Therefore, for physically-addressed cache memory it will be necessary to store the virtual address - to be able to write-back dirty cache blocks in the main memory – implying an additional on-chip memory overhead. Finally, for shared libraries or data, the fact that several virtual addresses map to the same physical address complicates [23] or prevents [24] their protection. In previous works [24], this point remains an open question.

Nevertheless, considering our threat model (a secure kernel is trusted), an adversary can only perform a splicing attack on the physical address space (physical relocation of memory block on the bus or in the off-chip memory); as a consequence the address used in the tag is the physical one.

The direct implication of such a choice when a MMU is implemented is the need to decrypt and re-encrypt the program image if it is received already ciphered. Indeed, the physical address is unknown until the translation of the address is known (when the application is loaded in the off-chip memory). Hence, the program must be decrypted first to be able to insert the physical address in plaintext blocks and then re-encrypted. The cost of such a computation is negligible since it occurs once at the application load-time. In case programs are statically mapped in the memory (XIP, eXecute In Place) the programmer can use the physical address to encrypt the software with PE-ICE prior to installing it in memory and thus avaiding any performance overhead at load-time.

On the other hand the use of the physical address solves the issue of shared data and of shared libraries encryption and integrity checking. Two physical sections of memory can be reserved, one for shared data and one for shared libraries, and handled by PE-ICE with a dedicated encryption key (or keys, see section 4-6.3).

## 4.6 Security Analysis

### 4.6.1 Active Attacks

The security of PE-ICE against the three active attacks (spoofing, splicing and replay) described in the threat model and carried out on a chunk is quantified in Table 1 depending on four parameters:

- *t* the bit width of the tag,
- *a* the number of address bits in the tag,
- *r* the bit width of the random value RV,
- and *b* the ciphered block length (in bytes).

Our threat model considers that an adversary can only access off-chip data. Thus, for spoofing attacks the adversary can only modify ciphertext blocks. As mentioned in Section 4.1, attacks consisting in the insertion of random ciphertext or the tampering with certain ciphertext bits succeed with probability $1/2^t$. Moreover, an adversary able to predict – without knowledge of the key – the effect of ciphertext bit manipulations on chosen bits in the plaintext (e.g. tag bits) implies the underlying block cipher is broken.

TABLE 1
SECURITY LIMITATIONS OF PE-ICE REGARDING THE DEFINED ACTIVE ATTACKS (EVALUATED IN LIKELIHOOD OF SUCCESS)

| Attack | | RO data | RW data | |
|---|---|---|---|---|
| | | | t = a + r | t = r |
| Spoofing attack | | $1/2^t$ | $1/2^t$ | $1/2^t$ |
| Splicing attack | Inside a splicing segment | 0 | 0 | $1/2^t$ |
| | Outside a splicing segment | 0 | $1/2^r$ | |
| Replay attack | | N/A | $1/2^r$ | $1/2^t$ |

Concerning replay, the outputs of the random number generator implemented to produce RVs are assumed equiprobable; therefore the probability of success for a replay is equal to $1/2^t$ and is the same for a splicing attack when $t = r$ (Fig. 3b). When the address is used in the tag ($t=a+r$ – Fig. 3c), the physical address space protected against splicing attacks is determined by $a$ and $b$. It is equal to ($2^a$ x $b$) and is called in the following a *splicing-segment*. However, $a$ might have a size which could be insufficient to cover the whole address space[1]; hence a different key must be attributed to each splicing segment contained in the application addressing space. The key is thus said *splicing-segment-dependent*. In this way an adversary which swaps two memory blocks with the same address bits in the tag from a splicing segment to another will be detected since the decryption of the fake block will be performed with the wrong key. Such a requirement for the key only applies to RO data since the tag for RW data already includes a countermeasure against replay which protects against this attack. As a consequence, considering that the keys used to encrypt the RO memory section are *splicing-segment-dependent*, an adversary cannot perform a splicing attack on such a memory section. Thus, we consider that it is impossible to perform a splicing attack on a chunk of RO data while for RW data a splicing attack carried out on a chunk from a splicing segment to another has $1/2^r$ chance to succeed and 0 inside a splicing-segment.

### 4.6.2 Confidentiality and Passive Attacks

An interrogation might arise when considering the encryption scheme in PE-ICE: is confidentiality affected by the insertion in the plaintext blocks of data potentially known by the adversary (e.g. the data address)?

Considering our threat model (SoC trusted), the adversary might only perform two passive attacks to challenge data confidentiality: ciphertext-only attacks – the eavesdropper tries to deduce the secret key or the plaintext by observing the ciphertext – and known plaintext attacks – the adversary additionally knows a part of the plaintext. Therefore, the choice of the block cipher algorithm is essential and must be secure against these two kinds of attacks. However, this is the minimum requirement for a block encryption algorithm, and in the following the block cipher implemented in PE-ICE fulfills this necessary condition.

Moreover, PE-ICE implements ECB mode to encrypt chunks. The weak point of this encryption mode relies on the fact that a given plaintext block always yields the same ciphertext block upon encryption with a given key. For RO data, the tag – address bits – is a true nonce. Thus, nonces make each plaintext chunk different from the others; this ensures that the encryption of the same payload twice yields to two different ciphertext blocks in ECB. Such reasoning is not true for RW chunks when a random value is used to thwart replay attack. However, the random value strengthens the robustness of ECB modes by ensuring that most of the time the adversary cannot figure out that a given payload is processed twice by the processor. Indeed, statistically there is little

[1] For instance for 32-bit processor architecture with 4GB of address space, the minimum size for $a$ to cover the whole address space is 28-bit if $b$ = 16 (AES – Advanced Encryption Standard [25]).

probability that the same plaintext block (payload + tag) appears twice during application execution. This probability depends on the size of *r* and on the way the payload is generated. If the same plaintext does occur for a given address, the eavesdropper might only deduce that a same data has been written at this given address at two different times.

### 4.6.3 PE-ICE's Encryption Key Requirements

As previously mentioned, for PE-ICE to be secure an encryption key must be splicing-segment-dependent. However, such a key must respect further requirements.

Different program images may be stored in the same physical splicing segment at different times. As a consequence, the tag inserted in each RO chunk will be the same at a given address for all those program images. Thus we need to avoid replay attacks of RO data from one application to another. Therefore for detecting such a kind of replay the encryption key must also be *application-dependent* meaning that the key must be different for each application loaded in memory.

Nevertheless, another attack remains feasible: an adversary may install an application in a splicing segment and record the resulting RO ciphered blocks in the memory with their corresponding address. Then, he resets the corresponding splicing-segment and re-installs the same application but at a different starting address in the same splicing-segment. Hence, since the secret-key is only application-dependent and splicing-segment-dependent, he could replay the RO ciphered blocks from the first installation at the address they had been stored at, without detection. To thwart such an attack the key must be changed on each loading of an application, in this case the key is said *installation-dependent*. Note that the notion of installation-dependent includes the notion of application-dependent.

Those encryption key requirements only apply to RO data. Indeed a countermeasure against replay during run-time is already provided for RW data and is valid for the different kinds of attacks presented above. Hence, the same key could be used for all RW data. In order to fulfill all requirements for the secret keys – splicing-segment-dependent and installation-dependent – dedicated to RO data, an encryption key is randomly generated on-chip on each application loading and for each splicing segment contained in the physical addressing space consumed by the application code. A PE-ICE implementation example is given in section 5 which shows that in practice the number of keys required is low.

With static application mapping, key management is further simplified since an application is always stored at the same address; hence the encryption keys only need to be splicing-segment-dependent and application-dependent. In this case, the application might be encrypted off-line with a PE-ICE scheme using a set of encryption keys respecting the two latter requirements and stored as such in the off-chip memory.

For the shared libraries, the solution proposed in section 4.5 must be carefully implemented. In order to fulfil the key requirements, an encryption key must be dedicated to each splicing segment contained in the reserved memory section, and each time a new library is installed, the concerned splicing segment(s) must be re-encrypted with a new key.

## 5. PE-ICE Implementation and Evaluation

In the following, a PE-ICE configuration is defined as the implementation of PE-ICE with a given block cipher. A PE-ICE configuration is denoted PE-ICE-*bw* where *bw* is the bit width of the block processed by the underlying block cipher.

In this section a PE-ICE configuration is first described. Then performances of several PE-ICE configurations at run-time are evaluated. Finally a comparison of PE-ICE with a generic composition scheme implemented in the Encrypt-then-Mac fashion is proposed.

## 5.1 PE-ICE-160 – A PE-ICE Configuration

The Rijndael algorithm is the block cipher who won the NIST contest for a new block encryption standard. The related standard is called AES [25] (Advanced Encryption Standard). AES processes 128-bit blocks and enrolls 128, 192 or 256-bit key. However, the original Rijndael [26] block cipher supports any key and block sizes that is a multiple of 32, between 128 and 256. This leads to several configurations for PE-ICE based on this block cipher. We studied three of them PE-ICE-128, PE-ICE-160 and PE-ICE-192, which use the Rinjdael algorithm processing respectively 128-bit (AES), 160-bit (Rijn-160) and 192-bit (Rijn-192) blocks. For sake of clarity, we only detail PE-ICE-160 configuration in this paper; for a description of the other configurations, refer to [20].

### 5.1.1 Address Computation and Layout of a chunk

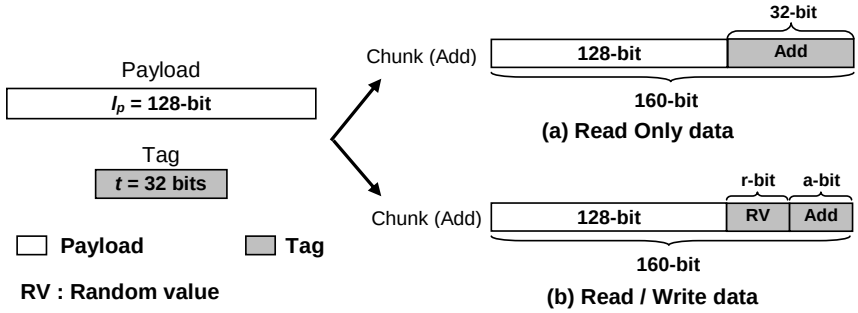



Fig. 4. Layout of a PE-ICE-160 chunk before encryption

PE-ICE shifts the physical addressing by inserting tags between payloads. This shift must be transparent for the CPU, thus PE-ICE handles the address translation. In the following $t_b$ and $l_{pb}$ denotes respectively the size in bytes of the tag and of the payload contained in a chunk. Moreover, we consider that PE-ICE protects the whole physical memory space. Hence, to retrieve an address $A_P$ of a PE-ICE chunk from the address $A_{CPU}$ provided by the CPU, PE-ICE first computes the position $P$ of a payload in the address space seen by the CPU:

$$P = \frac{A_{CPU}}{l_{pb}}$$

Then, $A_P$ can be computed as follows:

$$A_P = A_{CPU} + P \times t_b$$

Hence, $l_{pb}$ and $t_b$ must be a power of 2 to allow a simple computation of $P$ and $A_P$ in hardware. For PE-ICE-160 we choose $l_{pb} = 16$ and $t_b = 4$. The composition of the resulting PE-ICE-160 chunk is depicted in Fig. 4 (where $a$ may be 0).

### 5.1.2 Security limitations

The security limitations of PE-ICE-160 are directly deduced from Table 1. Concerning RO data, the 32-bit address of each ciphered block is used as tag (Fig. 4a). Hence, a splicing segment is of the size of the addressing space (4GB) and thus only one encryption key dedicated to RO data is required per application. $K_{RO}$ denotes such a key.

For RW data, the strength of the proposed countermeasure against replay and splicing attacks depends on the designer's choice of the values for $r$ and $a$. For example if $a = 24$ and $r = 8$, a splicing segment is 256 MB long (224 x 16B); thus a replay attack has a $1/2^8$ chance to succeed, while an adversary has a $1/2^8$ chance to succeed with a splicing from a splicing-segment to another

and 0 with a splicing inside a splicing-segment. When $t = r = 32$, for both replay and splicing attacks, the chance to succeed is of $1/2^{32}$. One key is required for all RW data stored off-chip and is denoted $K_{RW}$.

When $r$ is chosen small to save on-chip memory, a simple trick enabling to improve the strength against splicing and replay is to foresee in PE-ICE a counter which memorizes the number of detected intrusions. When this counter reaches a threashold value determined by the designer or by the software programmer, the external memory space dedicated to RW data is zeroized and a new $K_{RW}$ is generated. This implementation countermeasure can also be implemented to prevent brute force spoofing attack where the adversary inject random chunks until finding one passing the integrity checking process.

For both kinds of data the tag is at least 32-bit long, hence an adversary has a $1/2^{32}$ chance to succeed with a spoofing attack. Table 2 summarizes the security limitations of PE-ICE-160 evaluated in likelihood of success.

TABLE 2
SECURITY LIMITATIONS OF PE-ICE-160

| Attack | | RO data | RW data | |
|---|---|---|---|---|
| | | | t = a + r | t = r |
| Spoofing attack | | $1/2^{32}$ | $1/2^{32}$ | $1/2^{32}$ |
| Splicing attack | Inside a splicing segment | 0 | 0 | $1/2^{32}$ |
| | Outside a splicing segment | N/A | $1/2^{r}$ | |
| Replay attack | | N/A | $1/2^{r}$ | $1/2^{32}$ |

### 5.1.3. Memory Consumption

The amount of memory consumed by PE-ICE comes from the tag storage for the off-chip memory and from the storage of the reference random values for the on-chip memory. The off-chip memory overhead is defined by the ratio $R_{OF}$ between the tag and the payload bit widths ($R_{OF} = t/l_p$). For PE-ICE-160 the off-chip memory overhead is 25%. The on-chip memory overhead is defined by the ratio $R_{ON}$ between the bit-length of a random value used to protect a RW chunk against replay and the corresponding protected payload bit-length ($R_{OF} = r/l_p$). For PE-ICE-160 the on-chip memory overhead is respectively 6.25% and 25% depending if $r$ is 8-bit or 32-bit long. However, we proposed in [20, 21] a scheme reducing this overhead to a single $r$-bit value.

### 5.1.4 Latencies

In this section, we present the additional latencies introduced on off-chip memory accesses by PE-ICE-160 and AES-ECB encryption/decryption. The underlying CPU considered in this study is the ARM9E for which the optimum frequency in the 0.18µm CMOS process is around 200MHz with a 32-bit AMBA AHB bus running at 100MHz [27].

Our implementation of the AES algorithm is non-pipelined and takes 11 cycles to encrypt one 128-bit block of plaintext in ECB (1 cycle per round plus 1 cycle to bufferise the result). The AES implementation (0.18µm CMOS) presented in [28] shows that such latency is valid until 330 MHz. Hence, in the following we consider a realistic case for the ratio $R_{E/B}$ between the AES frequency ($F_{AES}$) and the bus frequency ($F_{AHB}$): $R_{E/B}=2$ - $F_{AHB}=F_{AES}/2$. When $R_{E/B}=2$ the intrinsic latency of the AES encryption seen on the AHB bus is of 6 cycles. The difference between all Rijndael versions relies in the number of rounds required to output a ciphertext block (or a plaintext block). This

number of rounds *Nr* is defined in [26] and is equal to: *Nr* = max(*Nk*; *Nb*) + 6; where *Nk* is the number of 32-bit words in the key and *Nb* the number of 32-bit words in the block processed. For Rijn-160 - processing 160-bit block and enrolling a 128-bit key - *Nr* is equal to 11. Hence the intrinsic latency of Rijn-160 seen on the AHB bus is of 6 cycles.

PE-ICE has been design in VHDL to be compliant with the AHB bus. Table 3 sums up the additional latencies[1] (expressed in bus cycles) introduced by PE-ICE-160 and by the AES-ECB encryption. The overhead of PE-ICE compared to AES-ECB encryption gives the cost of providing data authentication in addition to data confidentiality. On average this cost is of 22%. This latency overhead is partially due to the increase of the intrinsic latency of the underlying block cipher. Moreover, on read operations the tag generates wait cycles by polluting the PE-ICE-160 throughput while on write operations the fact that a plaintext block is collected in 4 cycles – instead of 5 for a 160-bit block - provokes bus resource conflicts.

TABLE 3
ADDITIONAL LATENCIES INTRODUCED BY PE-ICE-160 AND BY AES-ECB ON AN AHB BUS FOR THE OPERATIONS REQUESTED BY AN ARM9E CORE

| Operations | AES-ECB | PE-ICE-160 | |
|---|---|---|---|
| | Latencies (AHB cycles) | Latencies (AHB cycles) | Overhead vs. AES-ECB |
| 8 to 32-bit Write | 28 | 30 | 7% |
| 8 to 32-bit Read | 10 | 11 | 10% |
| 4-word Write | 10 | 11 | 10% |
| 4-word Read | 10 | 11 | 10% |
| 8-word Write | 10 | 12 | 20% |
| 8-word Read | 10 | 12 | 20% |
| 16-word Write | 10 | 14 | 40% |
| 16-word Read | 10 | 14 | 40% |

### 5.1.5 Silicon Area Usage

The hardware implementation of the same round is required for all versions of Rijndael. Therefore, the hardware resources needed for PE-ICE-160 and for the AES-ECB engine to obtain the latencies listed above can be estimated in numbers $N_{AES}$ of AES cores implementing such a round.

The processor can read or write a 32-bit data per AHB bus cycle. Thus, considering that a plaintext (or ciphertext) block is collected in 4 bus cycles and that the AES intrinsic latency seen on the AHB bus (6 cycles), the AES-ECB engine must implement two AES cores to reach the optimum throughput of 32-bit per cycle.

For PE-ICE-160, on write operations, 4 cycles are required to collect a plaintext block on the 32-bit AHB bus and 5 cycles to output a ciphertext block to the memory controller whereas on read operations, 5 cycles are required to collect a ciphertext block and 4 to output a plaintext block. The maximum throughput is theoretically higher on write operations than on read operations. However, the available bandwidth between PE-ICE and the memory controller limits the throughput on write operations; hence on both read and write operations a ciphered text block is processed only every 5 cycles. Considering the intrinsic latencies of the Rijn-160 (6 cycles), to reach the optimum throughput $N_{AES}$ has to be of two, as for the AES-ECB engine.

For an atomic bus transfer the same key is shared by all implemented AES cores, hence only one key expander core is needed for PE-ICE-160 and for the AES-ECB engine. By considering the figures provided by Ocean Logic [29] an AES encryption/decryption core with 11 cycles of latency

[1] Rijndael algorithm requires a decryption key which is computed from the encryption key. We make the realistic assumption that this decryption key is computed once and then stored in a dedicated register on-chip.

takes 24 Kgates and the corresponding key expander core 32 Kgates in the 0.18µ technology. This means that considering the chosen ratio $R_{E/B}$ = 2, the hardware cost of PE-ICE-160 and of the AES-ECB can be approximated to 80 Kgates.

At no additional hardware cost and low latency overhead, we showed that PE-ICE: i) strengthens AES-ECB encryption – the tag inserted before encryption prevents an adversary from detecting when the same data is transferred twice by monitoring bus transactions – ii) provides data authentication in addition to data confidentiality.

## 5.2 Performance Evaluation

### 5.2.1 Simuation Framework

In order to evaluate the performance during run-time of the studied PE-ICE configurations the SoC designer tool set [33] is used. This toolset provided by ARM consists in two separate applications: SoCDesigner - used to integrate custom components modeled in SystemC (CABA, Cycle Accurate Bit Accurate) into complex SoC platforms - and SoCExplorer - a cycle accurate simulator allowing to run benchmarks and to profile the platforms defined with SoCDesigner.

The SoC platform designed to evaluate the performance overhead implied by encryption and by PE-ICE is depicted in Fig.5. The libraries required to model the processor core (ARM9E), the AHB-bus and the external memory (PMEM for RO data and DMEM for RW data) have been provided by ARM with the SoC Designer tool set. The component called Latency_comp is the modeling in SystemC of the off-chip memory access latencies. It allows to set the number of wait cycles seen on the AHB bus for each kind of operations requested by the CPU. In the following we refer to the Base platform to denote the SoC platform which does not include hardware mechanisms for data security (encryption and integrity checking engine). To define the Base latencies we use the figures provided in the datasheet of an AHB compliant memory controller, the PL172[30]. We choose the lower read (9 cycles) and write (1 cycle) latencies assuming the following parameters for the underlying SDRAM memory: Precharge latency = 2, Activate latency = 2 and CAS latency = 2. This way, ideal memory accesses are defined and PE-ICE is pushed in the worst simulation case. One simulation platform per security engine has been designed: PE-ICE-128, PE-ICE-160, PE-ICE-192 and AES-ECB. Latency_comp parameters for each platform are set by adding the latencies their underlying computations introduce on the bus to the Base ones.

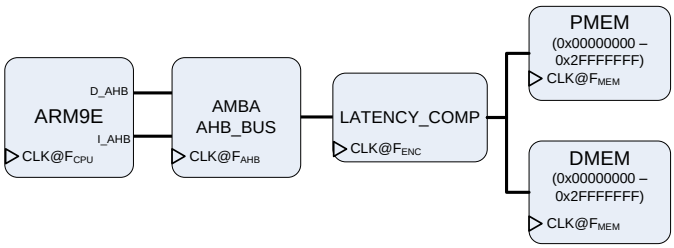



Fig. 5. Generic architecture of the simulation platforms

In [31] the proposed evaluation considers that the CPU, the AHB bus and the hardware encryption core run at the same frequency i.e. $F_{CPU} = F_{AHB} = F_{AES}$. In this paper another realistic case for embedded systems is explored where $F_{CPU} = 2*F_{AHB} = F_{AES}$ ($R_{E/B} = 2$) meaning that the latencies seen by the CPU are twice as big as the ones seen on the AHB bus. The architectural parameters defining the simulation frameworks are summarized in Table 4.

TABLE 4
ARCHITECTURAL PARAMETERS FOR SIMULATION

| | |
|---|---|
| Processor Core | ARM9E |
| Processor-memory bus width | 32-bit |
| AHB Clock ratio ($F_{CPU}$ / $F_{AHB}$) | 2 |
| Cache line size | 256-bit |
| Cache policy | Write-back |
| $R_{E/B}$ ($F_{AES}$ / $F_{AHB}$) | 2 |
| Base off-chip Read latency (AHB bus cycles) | 9 |
| Base off-chip Write latency (AHB bus cycles) | 1 |

### 5.2.2 Results

Eight benchmarks [32] designed for embedded systems were used in this evaluation. The simulation results for the Base platform serve as reference and are shown in IPC (Instruction Per Cycles) in Fig.6 for two different sizes of data cache and instruction cache (4KB and 128KB). We observed that the performance slowdown of the studied hardware mechanisms for data security is mainly related to the data cache miss rate; Fig.7 gives this cache miss rate for each benchmark and for both sizes of the data cache. Note that considering the low Base latencies and the fact that all applications are entirely protected, the worst case results are presented in this section. Indeed, all data processed during software execution do not require to be necessarily encrypted and integrity checked.

In order to illustrate the impact of the studied hardware mechanisms for data security we show in Fig.8 the simulation results of the platforms emulating the AES-ECB engine, PE-ICE-128, PE-ICE-160 and PE-ICE-192, in IPC normalized to the Base platform performance. The AES-ECB engine performance clearly highlights that the overhead is mainly due to encryption; it is 50% in the worst case (CJPEG – 4KB) and, 31.5% and of 14.3% on average respectively for 4KB and 128KB data cache. This quite important performance cost can be drastically reduced by using a wider processor-memory bus (e.g. 64-bit) or by running the encryption algorithm at its maximum frequency. Moreover, we did not explore the exploitation of the waiting time in the write buffer and of the Base write latency. The latencies introduced by the different security engines could be partially hidden on write operations by starting the encryption before storing data in the write buffer or at least at the same time as the memory access request. Increasing the data cache size decreases the overhead of the security engines by reducing the number of off-chip memory accesses, except for the DES benchmark for which the data cache miss rate remains almost the same. Nevertheless the interesting point is the low overhead implied by PE-ICE compared to the AES engine. We evaluate the performance slowdown of the integrity checking mechanisms proposed by PE-ICE when compared to AES encryption alone by normalizing the IPCs of the PE-ICE platforms to the AES-ECB engine performance (Fig.9). The best results are obtained with PE-ICE-128 since on average the degradation is 1.9% and 1.1% respectively for a data cache size of 4KB and of 128KB and in the worst case it is 4.1% (DES – 4KB). However PE-ICE-160 results are close since on average it implies a performance slowdown of 3.3% for a data cache of 4KB and of only 1.7% for a data cache of 128KB.

We evaluated the implementation of PE-ICE with several block ciphers and we showed that it provides data integrity in addition to data confidentiality with no additional hardware cost a negligible performance overhead when compared to standard encryption. In the next section we further highlight the advantage of PE-ICE through comparaison we a conventional scheme with the same goal as PE-ICE: providing data confidentiality and integrity.

## 5.3 Comparison with a Generic Composition Scheme

In this section a generic composition scheme - referred to as GC - is first described and then compared to PE-ICE-160.

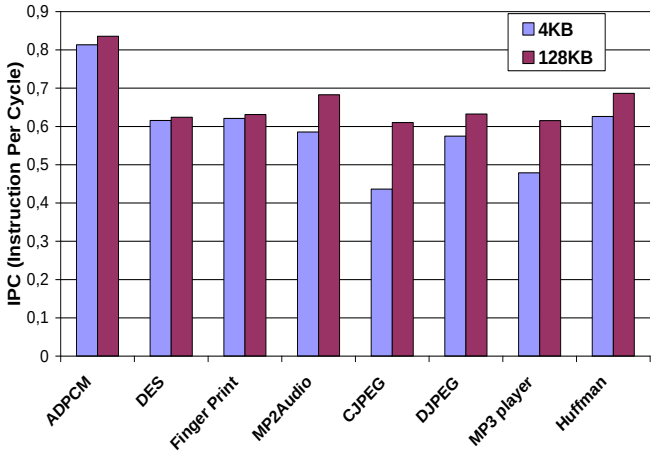



Fig. 6. Simulation Results for the Base Platform for two different data cache sizes (4KB and 128KB)

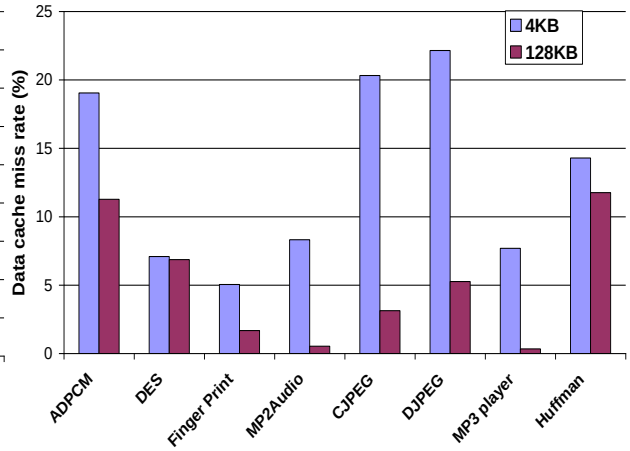
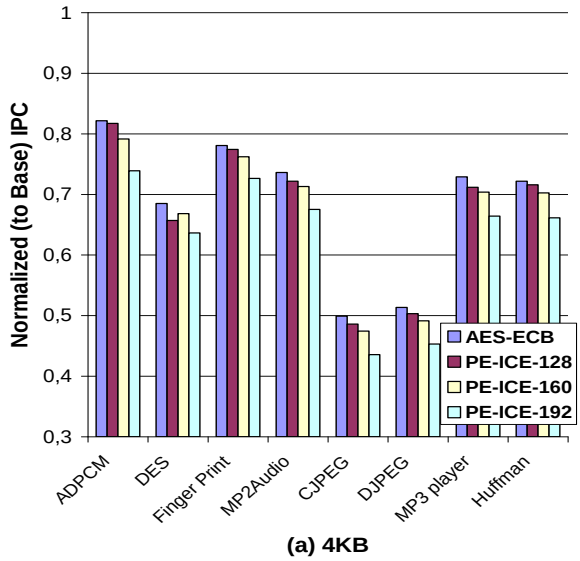



Fig. 7. Data cache miss rate for the set of benchmarks used for the performance evaluation and for two different data cache sizes (4KB and



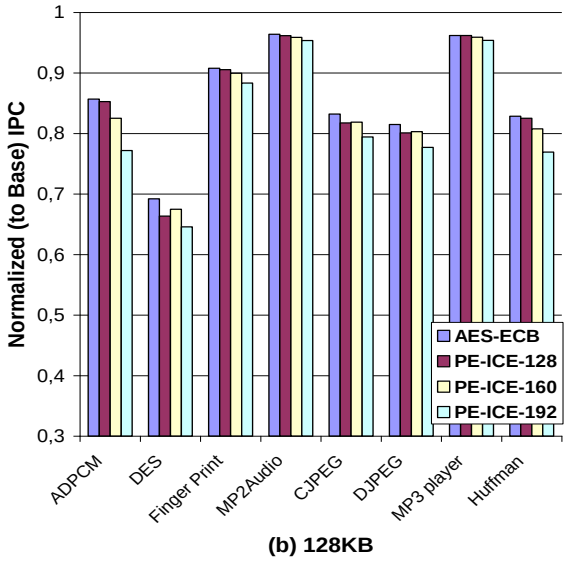
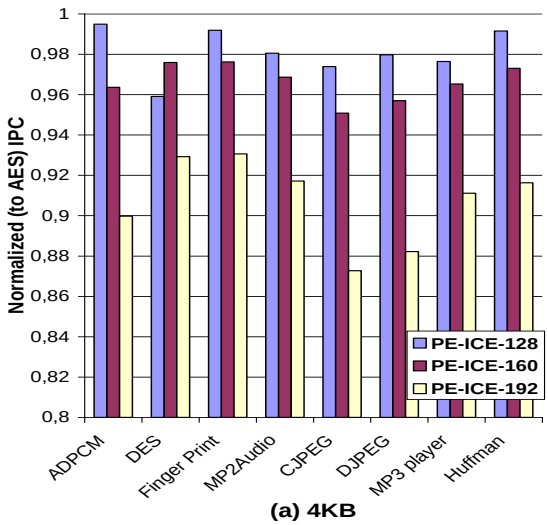




Fig. 8. Run-time overhead of AES-ECB encryption and of PE-ICE configurations for two data cache sizes (4KB - 128KB)



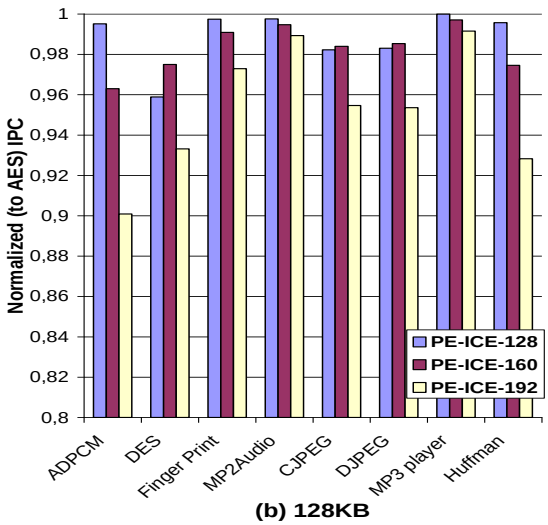




Fig. 9. Run-time overhead of the integrity checking mechanism of PE-ICE configurations compared to AES-ECB encryption alone for two data cache sizes (4KB and 128KB)

### 5.3.1 Description and Evaluation of the Generic Composition Scheme

**Description of GC.** GC is the association of an AES encryption with a CBC-MAC [16] algorithm in the Encrypt-then-MAC fashion. Encrypt-then-MAC is chosen because it is the most secured conventional method to pair an authenticated mode and an encryption mode as proved in [15]. As for PE-ICE the encryption mode is ECB in order to perform a fair comparison between GC and PE-ICE concerning the integrity checking overhead. We consider ECB encryption secure inGC even if the same block encrypted twice always yields to the same ciphertext block, which is not the case for PE-ICE. The tag required for the integrity checking process is computed over a chunk with the CBC-MAC algorithm. The underlying block cipher $E_k$ for our CBC-MAC implementation is AES. The Encrypt-then-MAC construction is implemented therefore the tag is computed over the ciphered chunk composed of *m* AES blocks ($C_1$, $C_2$, …, $C_m$) by using a different key than the one required for the encryption [15]. A 128-bit vector N is additionally enrolled in the CBC-MAC computation to thwart replay and splicing attacks. N is composed of the 32-bit chunk address concatenated with an *r*-bit vector RV and padded with zeroes to be 128-bit. The address serves to thwart splicing attacks by making N different for each ciphered chunk stored off-chip. For RW data, RV is the countermeasure against replay attacks, it is an *r*-bit random value generated on-chip, with its reference RV' stored also on-chip. Thus it can be retrieved for integrity checking on read operations while making it secret and tamper-proof from an adversary point of view. For RO data RV is padded with zeroes. In the literature [12, 13] the chunk size is defined by the cache line length. However, this choice for the CBC-MAC is inefficient for big cache blocks in terms of latency due to the inherent recursiveness of such a MAC algorithm. Thus, the tag is computed over a chunk *M* composed of two ciphered blocks – $M = (N, C_1, C_2)$ – independently of the size of the cache line. This tag is then truncated to 32-bit to decrease the memory bandwidth pollution generated by its transmission on the bus and to optimize the off-chip memory overhead. The resulting CBC-MAC implemented is depicted Fig 10.

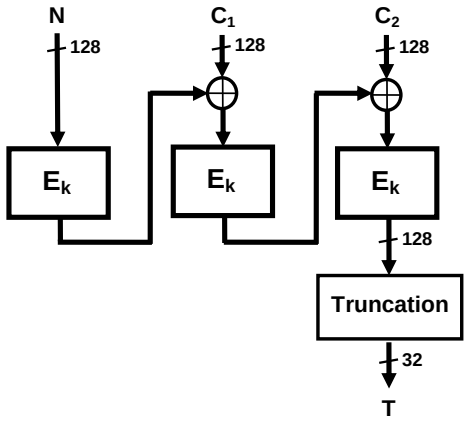



Fig. 10. CBC-MAC implemented in GC

**Security analysis.** In the following, the defined CBC-MAC implementation is evaluated relatively to the three attacks exposed in the threat model (spoofing, splicing and replay) and carried out on a chunk. CBC-MAC is based on block cipher encryption; therefore its outputs are equiprobable from an adversary's standpoint. Concerning spoofing and splicing attacks, the chance to succeed for an attacker depends on the size of the tag and is equal to $1/2^{32}$. The strength of the countermeasure against replay depends on *r*: the chance for replay to succeed is equal to $1/2^r$. The latter probability is limited by the size of the tag, meaning that there is no sense in choosing $r > 32$.

**Memory overhead.** The off-chip memory overhead of GC is of 12.5% since it requires to store 32-bit of tag for 256-bit of payload. The on-chip memory overhead depends on the size of RV and is defined by the ratio between *r* and the protected payload.

**Latencies.** In this section we consider an implementation of GC on the AHB bus to evaluate the latencies introduced on read and write operations. On read operations the encryption of N is parallelized with the memory access latency, hence the CBC-MAC latency seen on the AHB bus is only due to two consecutives AES encryptions plus 1 cycle of buffering, resulting in 11 cycles (for $R_{E/B}$ = 2) with 4 additional cycles to collect the first block. The integrity checking process is parallelizable with decryption process since they are both performed on the ciphertext, therefore the resulting latency for GC on read operations is of 15 cycles. On write operations, the CBC-MAC has the same latency (11 cycles) since the enrollment of N is hidden by the AES encryption. However, the data encryption process and the tag computation latencies are only partially parallelized. Moreover, the CBC-MAC generates a RMW write on the tag for the 8 to 32-bit and 128-bit write operations; this means that the chunk and the corresponding tag are loaded and checked, to be recomputed by enrolling the new 128-bit value in the CBC-MAC computation. Table 5 sums up the additional latencies introduced by GC on the AHB bus. On average the overhead of GC compared to AES encryption alone is of 52% for $R_{E/B}$ = 2.

TABLE 5
ADDITIONAL LATENCIES INTRODUCED BY GC ON AN AHB BUS FOR THE OPERATIONS REQUESTED BY AN ARM9E CORE

| Operations | GC (AES + CBC-MAC) | |
|---|---|---|
| | Latencies (AHB cycles) | Overhead vs. AES-ECB |
| 8 to 32-bit Write | 38 | +36% |
| 8 to 32-bit Read | 15 | +50% |
| 128-bit Write | 20 | +100% |
| 128-bit Read | 15 | +50% |
| 256-bit Write | 14 | +40% |
| 256-bit Read | 15 | +50% |
| 512-bit Write | 14 | +40% |
| 512-bit Read | 15 | +50% |

**Hardware cost.** On read operations the decryption and integrity checking processes are parallelized; thus despite the fact that they are both based on the AES algorithm, the AES-ECB engine and the CBC-MAC scheme of GC cannot share hardware. As shown in section 5.1.5, the AES-ECB encryption requires two AES encryption/decryption cores when $R_{E/B}$ = 2. Considering the intrinsic latency of the CBC-MAC (11 cycles) algorithm three AES cores are required to reach the maximum throughput of 32-bit/cycle. However, only the encryption process is involved in the CBC-MAC computation, therefore the hardware cost can be optimized by using an AES core implementing the encryption process only. The silicon area consumed by such a core is estimated at 16 Kgates by Ocean Logic [29] in the 0.18µ technology. Moreover, the AES-ECB encryption and the CBC-MAC computation require separated key expansion cores since they enroll two different keys. The resulting hardware cost for GC is of 160 Kgates when $R_{E/B}$ = 2.

**Run-time performance.** A simulation platform has been designed for GC by configuring the Latency_comp component with the latencies given in Table 5 added to the Base ones. The simulation framework is the same as the one described in section 5.2.1. GC has also been evaluated for two data cache sizes: 4 KB and 128 KB. Similarly to PE-ICE configurations the overhead of GC compared to the Base performance is mainly due to the encryption as shown in Fig.12. Nevertheless the additional performance slowdown of the integrity checking mechanisms of GC (CBC-MAC) is non negligible (Fig.13) when compared to the AES-ECB engine since it is of 18% in the worst case scenario (DJPEG – 4KB), and of 13.7% and 7.8% respectively in average for a data cache of 4 KB and of 128 KB.

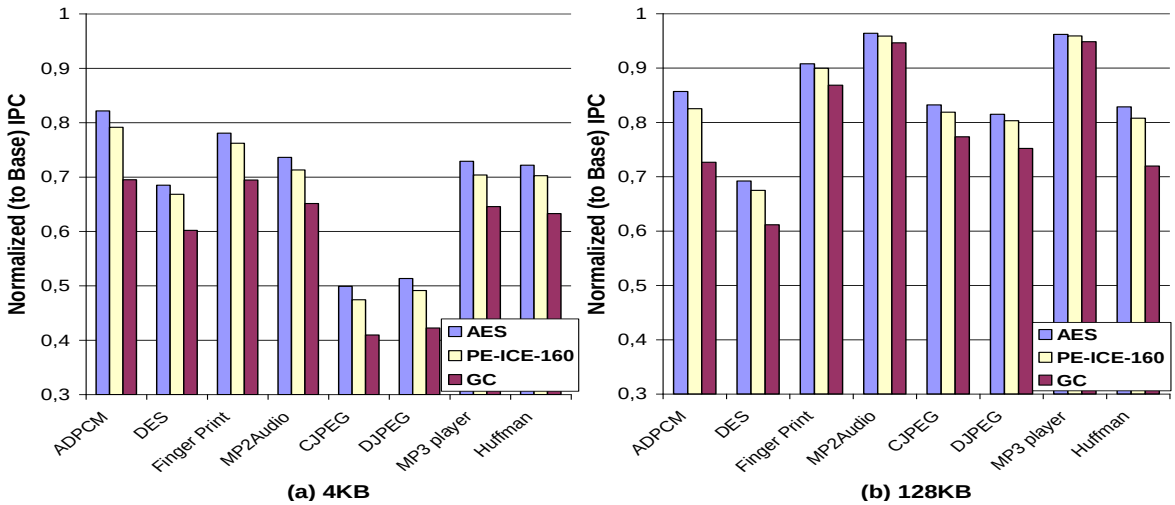



Fig. 11. Run-time overhead of GC, of the AES-ECB engine and of PE-ICE-160 for two data cache sizes (4KB and 128KB)

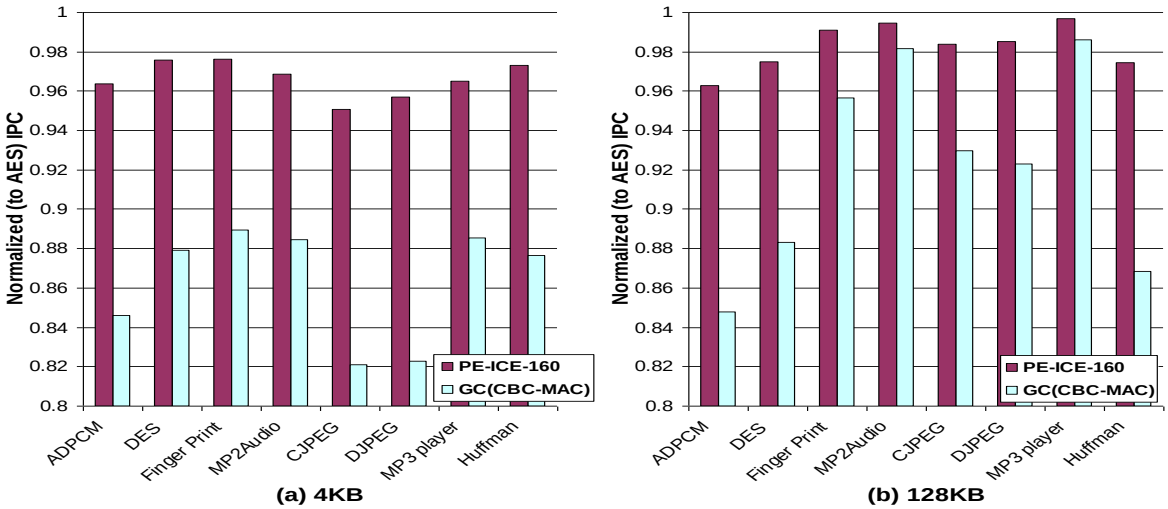



Fig. 12. Run-time overhead of GC, of the AES-ECB engine and of PE-ICE-160 for two data cache sizes (4KB and 128KB)

### 5.3.2 Comparison between GC and PE-ICE

To compare GC and PE-ICE-160, we evaluate their respective cost to ensure data integrity in addition to data confidentiality. The AES-ECB engine is used as the reference cost to provide data confidentiality since PE-ICE-160 and GC implement both the ECB encryption mode. Table 6 shows the overhead implied by the integrity checking mechanisms in PE-ICE-160 and in GC in terms of hardware area, latencies and run-time performance hit. PE-ICE-160 does not require additional silicon area to achieve the integrity checking process while GC implies an overhead of 100%. In term of latencies the overhead of GC reaches up to 52% on average while for PE-ICE-160 it remains always under 22% on average. The additional performance slowdown when compared to the AES-ECB encryption is roughly four times lower for PE-ICE-160 than for GC. Concerning security, PE-ICE-160 has the same security limitations as GC regarding the defined active attacks when $r = 32$. Moreover, PE-ICE-160 increases the robustness of the ECB mode by introducing a random value – for RW data – or a nonce – for RO data. Hence for RO data a same plaintext block encrypted twice never produces the same ciphertext block while for RW data there is a little probability that the same plaintext block ciphered twice leads to the same ciphertext block. This is not ensured by GC. The

main advantage of GC concerns the memory consumption since for the same value of *r*, GC implies an on-chip and off-chip memory overhead twice smaller than PE-ICE-160. Indeed, to maintain a strong security level with a fine granularity of integrity checking (per-block) PE-ICE requires having a dedicated tag to each processed ciphered block.

TABLE 6
SUMMARY OF THE COST OF THE INTEGRITY CHECKING MECHANISMS OF GC AND PE-ICE COMPARED TO AES-ECB

| | | GC (AES-ECB + CBC-MAC) | PE-ICE-160 |
|---|---|---|---|
| Hardware cost | | 160 Kgates ⇔ +100% | 80 Kgates ⇔~0% |
| Latencies | | +52% | +22% |
| Run-time slowdown | DC=4KB | +13.7% | +3.3% |
| | DC=128KB | +7.8% | +1.7% |
| Off-chip memory | | +12.5% | +25% |
| On-chip memory | | *r*/256 | *r*/128 |

# 6. CONCLUSION

We introduced in this paper the concept of Added Redundancy Explicit Authentication at the block level. We highlighted its relevance and its efficiency to ensure data integrity in addition to data confidentiality in the the context of processor-memory transaction. The proposed engine PE-ICE based on this concept, provides integrity checking in addition to encryption for no additional hardware cost and for a low run-time performance hit (less than 4%). We also showed that PE-ICE is more efficient at run-time and in terms of hardware than a generic composition scheme. Compared to PE-ICE, a generic composition scheme can require 100% of additional hardware and almost 14% of run-time performance overhead to provide integrity checking in addition to encryption.

Thus implementing PE-ICE in commercial devices to provide memory encryption and integrity checking is a more realistic solution than generic composition schemes.

On-going works include the integration of PE-ICE in a LEON[36] processor while current research deals with reducing the on-chip memory overhead induced by the countermeasure against replay.

# REFERENCES


[1] P.Kocher, R.B. Lee, G.McGraw, A.Raghunathan, and S.Ravi, "Security as a New Dimension in Embedded System Design", Proceedings of the Design Automation Conference (DAC), pp. 753-760, June 2004.

[2] S. Ravi, A. Raghunathan and S. Chakradhar, "Tamper Resistance Mechanisms for Secure Embedded Systems," IEEE Intl. Conf. on VLSI Design, January 2004.

[3] T.Alves and D.Felton. "Trustzone: Integrated hardware and software security", ARM white paper, July 2004.

[4] Trusted Computing Group. "TCG Specification Architecture Overview Revision 1.2." April 2004, available at: https://www.trustedcomputinggroup.org/groups/TCG_1_0_Architecture_Overview.pdf.

[5] A. Huang. "Keeping secrets in hardware the microsoft xbox case study". MIT AI Memo, 2002.

[6] S. W. Smith and S. H. Weingart, "Building a High-Performance, Programmable Secure Coprocessor", in Computer Networks (Special Issue on Computer Network Security), volume 31, pages 831–860, April 1999.

[7] R. M. Best, "Microprocessor for Executing Enciphered programs", U.S. Patent No. 4 168 396, September 18, 1979.

[8] R. M. Best, "Crypto Microprocessor for Executing Enciphered Programs", U.S. Patent No. 4 278 837, July 14, 1981.

[9] 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 9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pages 168–177, November

2000.
[10] 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 Int’l Conference on Supercomputing, June 2003.
[11] Gookwon Edward Suh, “AEGIS: A Single-Chip Secure Processor”, PhD thesis, Massachusetts Institute of Technology, September 2005.
[12] R.B. Lee, P.C.S. Kwan, J.P.McGregor, J.Dwoskin, and Z.Wang, “Architecture for Protecting Critical Secrets in Microprocessors”, in Proceedings of the 32nd International Symposium on Computer Architecture (ISCA 2005), pp. 2-13, June 2005
[13] C. Fruhwirth, “New Methods in Hard Disk Encryption”, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005.
[14] M. G. Kuhn, “Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP”, IEEE Trans. Comput., vol. 47, pp. 1153–1157, October. 1998.
[15] M. Bellare and C. Namprempre, “Authenticated Encryption: Relations among Notions and Analysis of the Generic Construction Paradigm”, In T. Okamoto, editor, Asiacrypt 2000, volume 1976 of LNCS, p. 531-545. Springer-Verlag, Berlin Germany, December 2000.
[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.
[17] C.J.Mitchell, “Cryptanalysis of Two Variants of PCBC Mode When Used for Message Integrity”, ACISP 2005: 560-571.
[18] H. Hellström, “Propagating Cipher Feedback”, 2001, available at:http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/pcfb/pcfb-spec.pdf.
[19] B.Gassend, G.E.Suh, D.Clarke, M.vanDijk, and S.Devadas, “Caches and Merkle Trees for Efficient Memory Integrity Verification”, In Proceedings of Ninth International Symposium on High Performance Computer Architecture, February 2003.
[20] Reouven Elbaz – “Hardware Mechanisms for Secured Processor Memory Transactions in Embedded Systems”, PhD Thesis, University of Montpellier -LIRMM, 2006.
[21] R.Elbaz, D.Champagne and R.B.Lee, “TEC-Tree: A Low Cost and Parallelizable Tree for Efficient Defense against Memory Replay Attacks” Princeton University Department of Electrical Engineering Technical Report CE-L2007-002, March 2007
[22] D.Lie, C.Thekkath and M.Horowitz, “Implementing an Untrusted Operating System on Trusted Hardware”, in Proc. of the 19th ACM Symposium on Operating Systems Principles, October, 2003.
[23] D. Lie, “Architectural Support for Copy and Tamper-Resistant Software”, Ph.D Thesis, Stanford University, December 2003.
[24] J.Yang, L.Gao, and Y.Zhang, “Improving Memory Encryption Performance in Secure Processors”, IEEE Transactions on Computers. pp. 630-640, Vol. 54, No. 5, May 2005.
[25] National Institute of Science and Technology (NIST), FIPS PUB 197: “Advanced Encryption Standard (AES)”, November 2001.
[26] J.Daemen, V.Rijmen, “AES Proposal: Rijndael”, March 1999, available at: http://csrc.nist.gov/CryptoToolkit/aes/rijndael/ Rijndael-ammended.pdf.
[27] http://www.arm.com/pdfs/ARM9E_flyer_063_4.pdf
[28] A. Hodjat, David Hwang, B.C. Lai, K. Tiri and I. Verbauwhede, “A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18-µm CMOS technology” ACM Great Lakes Symposium on VLSI 2005: p.60-63.
[29] http://www.arm.com/products/DevTools/MaxSim.html
[30] ARM PrimeCell MultiPort Memory Controller PL172 - Technical Reference Manual, available at: http://www.nalanda.nitc.ac.in/industry/appnotes/arm/soc/ DDI0215B_MPMC_PL172.pdf.
[31] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and A. Martinez, A parallelized way to provide data encryption and integrity checking on a processor-memory bus. DAC 2006: p.506-509
[32] The Embedded Microprocessor Benchmark Consortium (EEMBC). http://www.eembc.org/.
[33] http://www.arm.com/products/DevTools/MaxSim.html
[34] Claude Shannon, “Communication theory of secrecy systems”, Bell System Technical Journal, 28, 1949.
[35] C S R C (Computer Security Resource Center) - Modes of Operation at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/
[36] http://www.gaisler.com/.