Mécanismes Matériels pour des TransfertsProcesseur Mémoire Sécurisés dans lesSystèmes Embarqués

Reouven Elbaz

UNIVERSITE MONTPELLIER II
SCIENCES ET TECHNIQUES DU LANGUEDOC

T H E S E

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE MONTPELLIER II

Discipline : Génie Informatique, Automatique et Traitement du Signal
Formation Doctorale : Systèmes Automatiques et Microélectroniques
Ecole Doctorale : Information, structures, Systèmes

présentée et soutenue publiquement

par

Reouven ELBAZ

Le 12 Décembre 2006 à Montpellier

Mécanismes Matériels pour des Transferts Processeur Mémoire Sécurisés dans les Systèmes Embarqués

Hardware Mechanisms for Secured Processor-Memory Transactions in Embedded Systems

JURY

M. Jean Claude Bajard, Professeur, Université de Montpellier II,	Président du jury
M. Viktor Fischer, Professeur, Université de St Etienne,	Rapporteur
M. Olivier Sentieys, Professeur, Université de Rennes I,	Rapporteur
M. Joan Daemen, Docteur, Société STMicroelectronics,	Examinateur
M. Pierre Guillemin, Société STMicroelectronics,	Examinateur
M. Jean Baptiste Rigaud, Maître de Conférence, Centre Microélectronique de Provence,	Examinateur
M. Gilles Sassatelli, Chargé de Recherche CNRS, LIRMM – UMII,	Examinateur
M. Lionel Torres, Professeur, Université de Montpellier II,	Directeur de Thèse

UNIVERSITE MONTPELLIER II
SCIENCES ET TECHNIQUES DU LANGUEDOC

T H E S E

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE MONTPELLIER II

Discipline : Génie Informatique, Automatique et Traitement du Signal
Formation Doctorale : Systèmes Automatiques et Microélectroniques
Ecole Doctorale : Information, structures, Systèmes

présentée et soutenue publiquement

par

Reouven ELBAZ

Le 12 Décembre 2006 à Montpellier

Mécanismes Matériels pour des Transferts Processeur Mémoire Sécurisés dans les Systèmes Embarqués

Hardware Mechanisms for Secured Processor-Memory Transactions in Embedded Systems

JURY

M. Jean Claude Bajard, Professeur, Université de Montpellier II,	Président du jury
M. Viktor Fischer, Professeur, Université de St Etienne,	Rapporteur
M. Olivier Sentieys, Professeur, Université de Rennes I,	Rapporteur
M. Joan Daemen, Docteur, Société STMicroelectronics,	Examinateur
M. Pierre Guillemin, Société STMicroelectronics,	Examinateur
M. Jean Baptiste Rigaud, Maître de Conférence, Centre Microélectronique de Provence,	Examinateur
M. Gilles Sassatelli, Chargé de Recherche CNRS, LIRMM – UMII,	Examinateur
M. Lionel Torres, Professeur, Université de Montpellier II,	Directeur de Thèse

A Tiffanie, à mes Parents,

Et à ma Grand-mère Myriam,

Remerciements

Je souhaite remercier en premier lieu « ma » Tiffanie, qui a su me supporter (dans tous les sens du terme) au cours de ces trois ans. Il n'y a pas de mots pour exprimer l'importance que tu as eu dans l'élaboration de ces travaux, et je ne parle même pas des heures passées sur des démonstrations mathématiques.

Ensuite mes remerciements vont à mes parents, Marie France et Haïm Elbaz, qui ont toujours su me témoigner leur amour inconditionnel, et bien évidemment à mes frères et sœurs, Shmouël, Yossi, Déborah, Tsyona et Meyer qui malgré la distance ont toujours marqué leur présence et leur soutien.

Ces travaux de thèse ont été réalisés dans le cadre d'un projet régional (PACA 2003-08) impliquant la société STMicroelectronics, le LIRMM (Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier) et le CMP (Centre Microélectronique de Provence – Ecole des Mines de St Etienne). Dans ce contexte, j'ai été amené à travailler avec de nombreuses personnes, j'espère ne pas en oublier.

Un immense merci à Pierre Guillemin qui au cours de ces trois ans a été mon responsable chez STMicroelectronics. Toujours présent, il n'a jamais économisé son temps et ses précieux conseils qui m'ont permis de poursuivre mes travaux de manière autonome. J'ai vraiment apprécié travailler à ses côtés et je vais regretter notre collaboration. Merci Pierre.

Je remercie également mon directeur de thèse, Lionel Torres, ainsi que Gilles Sassatelli du LIRMM qui ont su me faire confiance et me donner ma chance. Leurs encouragements et leurs remarques constructives m'ont permis de valoriser efficacement mes travaux. La bonne ambiance qui régnait lors de nos réunions a énormément contribué à ma motivation.

Ensuite, je remercie Michel Bardouillet de STMicroelectronics pour ces discussions particulières et animées qui m'ont amené à pousser ma réflexion toujours plus loin.

Merci à mon maître en design (et en jeux de mots), Albert Martinez de STMicroelectronics, qui a fait preuve d'une patience et d'une pédagogie époustouflante. Toujours disponible, j'ai énormément appris à ses côtés.

Je remercie également Claude Anguille de STMicroelectronics, avec qui j'ai travaillé durant la première année de thèse. Une année certes, mais durant laquelle il m'a beaucoup appris, en particulier concernant le travail d'équipe.

Je remercie le Professeur Viktor Fischer du Laboratoire Hubert Curien de St Etienne et le Professeur Olivier Sentieys de l'Université de Rennes 1 qui ont accepté d'être les rapporteurs de mes travaux de thèse.

Je suis particulièrement reconnaissant envers Joan Daemen de STMicroelectronics de m'avoir fait l'honneur de sa participation à mon jury de thèse. Par ailleurs, ses conseils prodigués au cours de ma thèse m'ont été d'une grande aide dans la compréhension de la cryptographie.

Je remercie grandement le Professeur Jean Claude Bajard du LIRMM de m'avoir fait l'honneur de présider mon jury de thèse.

Je remercie Jean Baptiste Rigaud du CMP pour avoir pris part à mon jury de thèse ainsi que pour ses encouragements au cours de ma thèse.

Le contexte « multi labo » dans lequel s'est déroulé ma thèse m'a amené à beaucoup voyager entre Rousset, Marseille et Montpellier, et je tiens à remercier particulièrement deux personnes qui ont tout fait pour me simplifier la vie. La géniale Evelyne Arrighi de STMicroelectronics : on a à peine le temps de penser aux démarches à entreprendre qu'elle a déjà tout fait !!!! Et Elisabeth Petiot du LIRMM, qui, même lorsqu'elle n'était plus responsable des missions du département microélectronique, était toujours présente pour répondre à mes questions dans la bonne humeur. Encore merci à vous deux, si tout le monde pouvait être comme vous….

Ensuite je tiens à remercier toute l'équipe d'AST (STMicroelectonics) : Stéphan Courcambeck (merci pour ton temps), William Orlando, Christian Schwarz, Sarah Hoffman, Jean Nicolaï, Klaus Rischmuller (merci pour tous tes conseils en termes de communications), Bernard Kasser et Norah Cowman.

Une « spéciale dédicace » à mes collègues de pause AST : Sophie Gabriele, Guillaume Petitjean et Lionel Martin ainsi qu'à mes amis thésards AST : Simon Conseil et William Ketchantang. Je crains de ne jamais retrouver une telle ambiance.

Je remercie les thésards du LIRMM ; je commence par mes colocataires de box : Robin et Nicolas qui ont su durant mon absence, défendre vaillamment mon bureau dans un hall où les places se font rares. Je pense également à Nicolas, Jean Baptiste (tu n'es pas thésard mais ça passe pour cette fois), Alin, Nabil, Fabrice, Benoit, Daniel, Jean Denis, Jean Etienne, Lionel, Julien, Annissa, Alex, Alex, Alex, Abdellah, Olivier, Zequin (Eric), Nicolas, Mickael, Laurent et Marion. Merci également à Nurten pour son aide de dernière minute.

Je remercie également mon compagnon de toujours et mon complice « musical » Vincent Kerzérho ainsi que son amie Anne Laure. Je remercie mon ami Wael Gouja ainsi que tous les autres : Boris, Daniele, Severine, Nadine et Franck, Arnaud, Maxime, Nizar, Romain, Christian, Seb…

Je ne peux terminer ces remerciements, sans remercier infiniment Dany et Alexandre Herrmann pour leur accueil toujours chaleureux lors de mes nombreux séjours à Montpellier.

Résumé

Les systèmes embarqués actuels (téléphone portable, assistant personnel…) ne sont pas considérés comme des hôtes de confiance car toute personne y ayant accès, sont des attaquants potentiels. Les données contenues dans ces systèmes peuvent être sensibles (données privées du propriétaire, mot de passe, code d'un logiciel…) et sont généralement échangées en clair entre le Système sur Puces (SoC – System on Chip) et la mémoire dans laquelle elles sont stockées. Le bus qui relie ces deux entités constitue donc un point faible : un attaquant peut observer ce bus et récupérer le contenu de la mémoire, ou bien a la possibilité d'insérer du code afin d'altérer le fonctionnement d'une application s'exécutant sur le système. Afin de prévenir ce type d'attaque, des mécanismes matériels doivent être mis en place afin d'assurer la confidentialité et l'intégrité des données. L'approche conventionnelle pour atteindre cet objectif est de concevoir un mécanisme matériel pour chaque service de sécurité (confidentialité et intégrité). Cette approche peut être implantée de manière sécurisée mais empêche toute parallélisation des calculs sous-jacents.

Les travaux menés au cours de cette thèse ont dans un premier temps, consisté à faire une étude des techniques existantes permettant d'assurer la confidentialité et l'intégrité des données. Dans un deuxième temps, nous avons proposé deux mécanismes matériels destinés à la sécurisation des transactions entre un processeur et sa mémoire. Un moteur de chiffrement et de contrôle d'intégrité parallélisé, PE-ICE (Parallelized Encryption and Integrity Checking Engine) a été conçu. PE-ICE permet une parallélisation totale des opérations relatives à la sécurité aussi bien en écriture qu'en lecture de données en mémoire. Par ailleurs, une technique basée sur une structure en arbre (PRV-Tree – PE-ICE protected Reference Values) comportant la même propriété de parallélisation totale, a été spécifiée afin de réduire le surcoût en mémoire interne impliqué par les mécanismes de sécurité.

Abstract

Today's embedded systems are considered as non trusted hosts since the owner, or anyone else who succeeds in getting access, is a potential adversary. The bus between the System on Chip (SoC) and the external memory is one of the weakest points of such systems because external memories contain sensitive data (end users private data, software code…) which are usually exchanged in clear form over the bus. Therefore an adversary may probe this bus in order to read private data or to retrieve software code (data confidentiality concern). Another possible attack relies on code injection (data integrity concern). Thus, hardware mechanisms must be designed to ensure data confidentiality and integrity. The conventional way to reach such a goal is to implement a dedicated hardware engine for each security service. Being secured, this approach prevents parallelizability of the underlying computations.

In this thesis, after a study of existing techniques and engines guaranteeing data confidentiality and integrity, two hardware mechanisms dedicated to the security of processor-memory transactions are proposed. First, a Parallelized Encryption and Integrity Checking Engine (PE-ICE) has been designed to provide an effective solution to ensure both security services to data. PE-ICE allows full parallelizations on processor read and write operations while optimizing the hardware resources required. Then, a technique based on a tree structure (PRV-Tree – PE-ICE protected Reference Values) with the same property of full parallelization, is specified to decrease the on-chip memory overhead implied by security mechanisms.

Contents

List of Figures

List of Tables

Introduction

Not a day passes in our lives without someone using an embedded system: PDA, mobile phones, MP3 players, set-top box, video equipments… The range of services provided by every single embedded system tends to widen rapidly and applications like on-line banking transactions, web browsing, email, application / game download become common on mobile devices. As a consequence the amount of sensitive information such as private data – bank information, passwords, email, photos… – or intellectual property – software, digital multimedia content… – contained or transiting in those devices also increases. The issue is that today's embedded systems are considered as untrustworthy hosts [1] since the owner, or anyone else who succeeds in getting access, is a potential adversary. Thus, one of the challenges for the high-technology industry in the development of pervasive computing is the ability to ensure secured computation and storage.

The attacks conducted on embedded systems [2] challenge several security services such as data confidentiality, data integrity 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 malicious entities. Availability refers to the requirement of ensuring the access to the device to the user without unexpected delay or obstacle.

Software attacks like viruses are the most famous threat because 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 Operating System (OS) and which propagated itself via Bluetooth. According to McAfee Virus Library, there were 120 types of Cabir variants [3] in 2005. The industry started to work on this issue with the Trustzone Project [4] from ARM and through consortiums such as the Trusting Computing Group [5] (TCG, formerly Trusting Computing Platform Alliance, TCPA) whose goal is to define

secured processing 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. The well known cracking of the Xbox gaming console shows that designing computing systems with such an assumption leads to simple physical attacks. In [6], 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 thesis we focus on physical non-invasive[1] attacks – or board level attacks – conducted on buses between the System 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 – what is more inconvenient for end users – to discover or to corrupt private data retrieved on buses or directly in memory. Thus, our goal is to provide a private and authenticated tamper resistant environment for application execution. This means we must 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 [7]. However, these solutions are not suited for embedded systems because the latter require an expensive and large package to be able to provide a high performance system while the former do not allow storing a large amount of code and data and do 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 [8, 9, 10] in 1979: data are encrypted before being outputted off-chip and are only decrypted once on-chip. However, encryption only ensures data confidentiality but does not provide tamper-detection mechanisms to guarantee data integrity. Later on, several research works [11, 12, 13, 14] considered this additional

[1] In this work we do not consider the non-invasive attacks called side-channel attacks. They are based on the analysis of the system behavior (power consumption, execution time, supply voltage, temperature, radiation …) to draw conclusion on the cryptographic functions or on the secret values on which the security of the underlying computing systems rely and which require specific countermeasures.

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. Moreover, the hardware resources needed are not optimized since the implementation of a dedicated engine for each security service is required.

Objectives: In this thesis, the goal is to provide a private and authenticated tamper resistant environment to applications running in embedded processors. The attacks targeted are the physical attacks conducted at the board level, more specifically bus probing and memory tampering (code and data injection). 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 sizes – to optimize hardware resources, memory access latencies and the memory bandwidth consumption at run-time.

Contributions: In order to reach the above mentioned objectives, we explore added redundancy and randomness to block encryption to provide data integrity in addition to confidentiality. Two hardware mechanisms are proposed:

(1) PE-ICE – *Parallelized Encryption and Integrity Checking Engine* – is a dedicated solution to guarantee data confidentiality and integrity of the external memory content with the following features:

- Full parallelization of the encryption and integrity checking process on write and read operations.
- Implementation of a single encryption algorithm to provide both security services: data confidentiality and integrity.

In this thesis, the architecture of PE-ICE and its security limitations are defined. Moreover PE-ICE is evaluated in terms of latencies introduced on memory accesses, of hardware resources required for its SoC implementation and of run-time performance degradation.

(2) PRV-Tree - *PE-ICE protected Reference Values Tree* – is a scheme allowing to securely store off-chip the meta-data used by PE-ICE and thus to propose an efficient solution against a specific kind of attacks (replay). PRV-Tree offers the following advantages when compared to existing solutions by using the same hardware as PE-ICE:

- Full parallelization of the underlying operations on read and write operations
- Reduction of the memory bandwidth consumption

In this thesis the concept of PRV-Tree is detailed and its advantages are highlighted through concrete application examples.

The security of our schemes relies on the use of a well known and studied block encryption algorithm (Rijndael [15]) and on the essential assumption that the System on Chip is trusted.

The thesis is organized as follows.

Chapter 1 introduces the cryptographic tool by defining the security service objectives and by describing the cryptographic function useful to provide such security services and discussed in the rest of the thesis.

Chapter 2 provides the threat model and defines the considered attack scenarios. Moreover we show through an example of attack that performing memory encryption only is not sufficient and could lead to simple attacks challenging data confidentiality. We also present the bus encryption and the memory content (integrity) verification principles. Finally this chapter deals with the issue of implementing cryptographic functions in a System on Chip.

Chapter 3 first relates the existing memory encryption engines and the memory verification schemes embedded on-chip. Then the techniques providing both data confidentiality and authentication are described and the related implementations in our application domain considering the same security perimeter (SoC trusted) are presented.

Chapter 4 defines the architecture of the proposed engine PE-ICE (Parallelized Encryption and Integrity Checking Engine). In this chapter we show how to add the integrity checking capability to block encryption. Moreover we describe PE-ICE processing on read and write operations, and the PE-ICE security limitations and memory consumptions are discussed.

Chapter 5 proposes several PE-ICE configurations which depend on the underlying block cipher and evaluates their implementation on a commercial on-chip bus and processor core.

Moreover, a comparison with a conventional scheme (data confidentiality and integrity separately provided by dedicated engines) is discussed.

Chapter 6 proposes a scheme called PRV-Tree (PE-ICE Protected Reference Value Tree) to eliminate the on-chip memory overhead implied by the PE-ICE countermeasure against a specific kind of attacks (replay). The advantages of PRV-Tree compared to existing solutions (Hash Trees) are finally discussed.

Chapter 7 concludes this thesis by summarizing the contributions. Further works are also detailed and a new research idea is proposed to improve the mechanisms reported in the thesis.

Chapter 1: The cryptographic tool

Cryptology is the science of secret. Two sub-disciplines compose cryptology: **cryptography** which is the science of keeping secrets and **cryptanalysis** which is the science of breaking those secrets.

This chapter presents the techniques provided by cryptography to allow secured communications and the existing attacks proposed by cryptanalysis to circumvent such techniques. The first section gives general term definitions. Then, section 1-2 exposes Kerckhoffs' principles which define what a secure cryptosystem is. Section 1-3 and section 1-4 respectively describe techniques for data encryption and for data integrity checking.

1-1. Definitions

Cryptography is a study of techniques dedicated to provide security services to implement information security. Such security services are confidentiality, authentication, data integrity and non-repudiation (definition taken from [16]):

Confidentiality is a service used to keep the content of information from all but those authorized to have it. Secrecy is a term synonymous with confidentiality and privacy. There are numerous approaches to providing confidentiality, ranging from physical protection to mathematical algorithms which render data unintelligible.

Data integrity is a service which addresses the unauthorized alteration of data. To ensure data integrity, one must have the ability to detect data manipulation by unauthorized parties. Data manipulation includes such things as insertion, deletion, and substitution.

Authentication is a service related to identification. This function applies to both entities and information itself. Two parties entering into a communication should identify each other. Information delivered over a channel should be authenticated as to origin, date of origin, data content, time sent, etc. For these reasons this aspect of cryptography is usually subdivided into two major classes: entity authentication and data origin authentication. Data origin authentication implicitly provides data integrity (for if a message is modified, the source has changed).

Non-repudiation is a service which prevents an entity from denying previous commitments or actions. When disputes arise due to an entity denying that certain actions were taken, a means to resolve the situation is necessary. For example, one entity may authorize the purchase of property by another entity and later deny such authorization was granted. A procedure involving a trusted third party is needed to resolve the dispute.

In the context of this thesis, non-repudiation is not applicable as the system on chip is the only active entity in a processor-memory communication system; therefore such a service issue is not dealt with in this dissertation. In the following, when the term data integrity is used, data origin authentication is implicitly checked. In this chapter, differences between these two notions are highlighted when required.

Encryption is the transformation process which makes a message, called **plaintext**, unintelligible. The resulting text of such a transformation is called **ciphertext**. Encryption operations are mathematical functions used to ensure the confidentiality of a message (but not only). **Decryption** is the operation which allows to retrieve the plaintext from the ciphertext. Both encryption and decryption primitives use a **key** which specifies the particular transformation respectively from plaintext to ciphertext and from ciphertext to plaintext.

In the following, the encryption function using the key K_e will be noted as E_{Ke}. Hence if P is the message to encrypt, then $E_{Ke}(P)$ is the encryption of P using the key K_e. It results in a ciphertext C such as $C = E_{Ke}(P)$. Similarly, the decryption function using the key K_d will be

noted as D_{Kd} and the plaintext P is recovered by applying D_{Kd} on the ciphertext C: $P = D_{Kd}(C)$. Encryption techniques are exposed in section 1-3.

The **integrity checking process** is the set of operations which allows to verify the data integrity of a message during its storage or transmission. Such a process is described in section 1-4.

A **cryptosystem** is a set of algorithms used to provide the above mentioned security services and particularly confidentiality. It is usually composed of three algorithms: one for the key (or keys) generation, one for encryption noted *E* and one for decryption noted *D*.

A **cryptanalyst** studies techniques allowing to retrieve secret information on which the robustness of a cryptosystem - e.g. the key - relies. In the following a malicious cryptanalyst is referred to as an **adversary** or an **attacker**. The term **eavesdropper** is used when the attack only involves monitoring of the communication channel.

Two families of attacks have to be considered:

- **Active attacks** which allow message deletion or corruption, data injection or replay.
- **Passive attacks** which consist in observing ciphertext on the communication channel (eavesdropping).

1-2. Kerckhoffs' Principles

In 1883, Auguste Kerckhoffs defined six principles to design a secure cryptosystem [17]:

1. The system should be, if not theoretically unbreakable, unbreakable in practice;
2. It must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience;
3. Its key must be communicable and retainable without the help of written notes, and changeable or modifiable at the will of the correspondents;
4. the cryptogram should be transmissible by telegraph;
5. the encryption/decryption apparatus should be portable and operable by a single person;

6. it is necessary - given the circumstances that command its application - that the system be easy to use, requiring neither mental strain nor the knowledge of a long series of rules to observe.

All of Kerckhoffs' requirements are still valid today. Among them we can notice that the second principle states that all algorithms used in a cryptosystem should be public knowledge.

1-3. Encryption Techniques

The first purpose of encryption techniques is to provide the confidentiality security service. The second and the third of Kerckhoffs' principles implicitly mean that the robustness of an encryption scheme relies on a secret: the key. If this secret is shared by all persons who use a cipher to communicate in a private way, this cipher is called symmetric or secret-key cipher. On the other hand, if the secret is only known by the intended recipient of the encrypted message, the cipher used is called asymmetric or public-key cipher.

1-3.1. Secret-Key Cipher (a.k.a. Symmetric-Key Cipher)

1-3.1.1. Principle

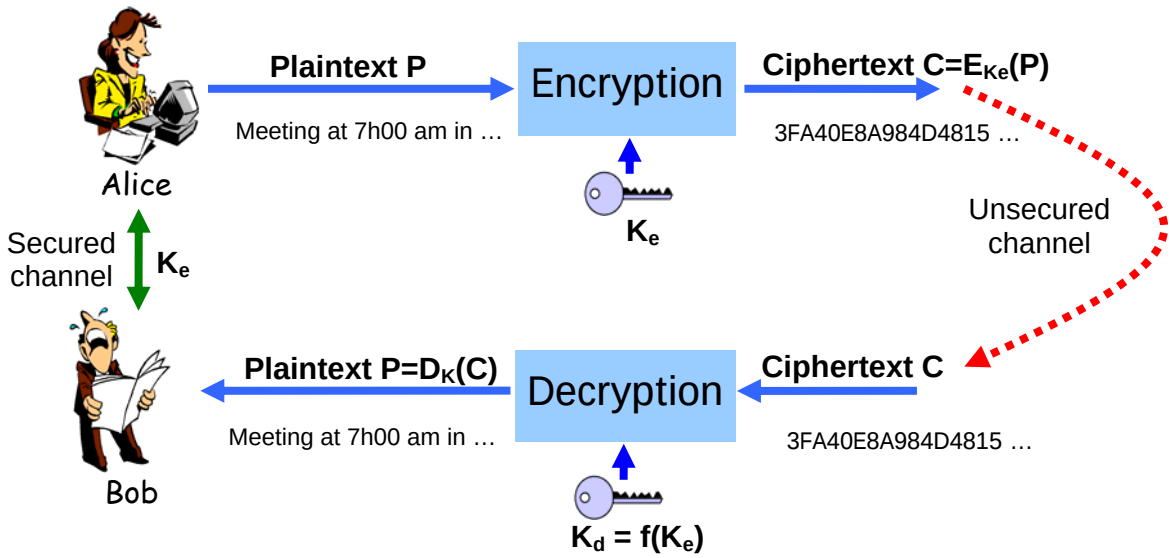


Figure 1-1 Secret-key (a.k.a. symmetric key) cipher principle

The entities using cryptosystems based on this family of ciphers to communicate must agree on a secret. This secret is the encryption key K_e and the decryption key K_d. This is why they are called secret-key ciphers. Moreover, such ciphers are also referred to as symmetric-key cipher since they mostly use the same key for encryption and for decryption ($K_e = K_d$) or because K_e is easily computable from K_d and vice-versa.

Example of utilization: Alice and Bob want to communicate over an unsecured channel (Figure 1-1). They first choose together a secret key K_e. Then Alice encrypts the message with a symmetric-key cipher and K_e and sends it to Bob. Bob receives the ciphertext and uses the same symmetric-key cipher and K_e (or computes K_d from K_e) to retrieve the plaintext.

Symmetric ciphers are designed to provide fast computation on encryption and on decryption. However the previous example highlights the main drawback of symmetric-key cryptosystems which is to securely agree on a secret key prior to establishing the communication.

Secret-key algorithms are divided into two families: stream ciphers and block ciphers.

1-3.1.2. Stream Ciphers

1-3.1.2.1. Principle

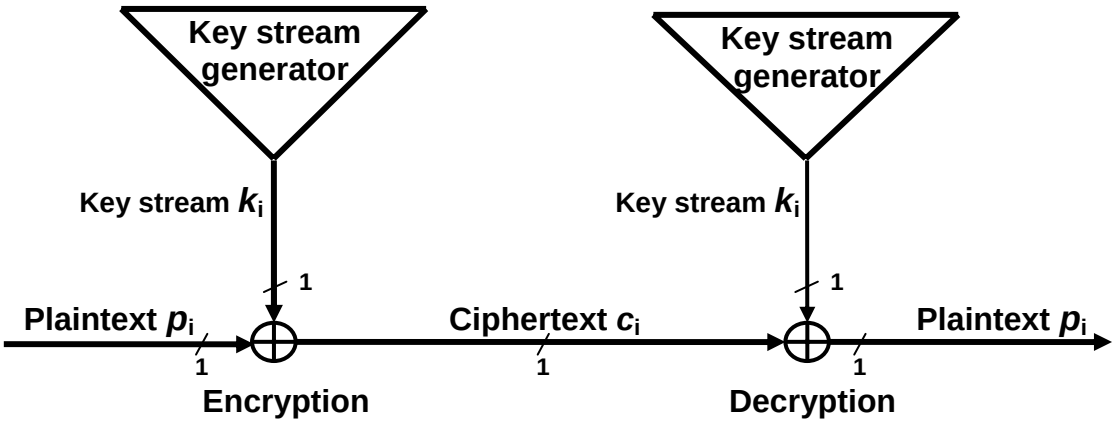


Figure 1-2 Stream cipher principle

The concept of stream ciphers was first introduced by Gilbert Vernam [18] in 1917. He proposed to encrypt a message character by character with a previously-prepared key of the

same size as the message. The reverse procedure should be applied to the ciphertext to retrieve the plaintext; i.e. the same key has to be applied character by character on the ciphertext.

Modern stream ciphers are inspired by such a principle and are composed of two parts (Figure 1-2): a key stream generator and a XOR operator. The key stream generator provides a bit stream k_i; the encryption proper is performed by the XOR operator which combines k_i and the plaintext p_i one bit at a time. It results in the ciphertext c_i as follows:

$$c_i = p_i \oplus k_i.$$

The decryption process is also a XOR operation between the same bit stream as the one used for encryption and the ciphertext c_i. In the light of the XOR operator property ($k_i \oplus k_i = Id$[2]), the result of the decryption is the plaintext:

$$p_i = c_i \oplus k_i.$$

1-3.1.2.2. One Time Pad: The Perfect Stream Cipher

One type of stream ciphers has been proved theoretically unbreakable by Shannon [19]: the One-Time Pad (OTP). OTP uses a secret key stream randomly generated and at least as long as the plaintext. Hence the sole information that a cryptanalyst could guess, is the length of the plaintext. However, generating a random bit stream as long as the plaintext and transmitting it to the intended recipient of the message are two difficult tasks which make OTP implementation unaffordable.

1-3.1.2.3. Modern Stream Ciphers

In order to overcome the OTP issues, key stream generators of modern stream ciphers use algorithms which produce a pseudo random key stream k_i from an internal state S_i and a secret key as depicted in Figure 1-3. The internal state is initialized by a small data called "seed" and all the cryptographic complexity of such a scheme resides in the output algorithm. The objective is to make a bounded adversary think that the key stream generator output is

[2] Id is the identity function.

random. However the resulting key stream is obviously not truly random and the unbreakable provability of OTP is no longer valid for such stream ciphers.

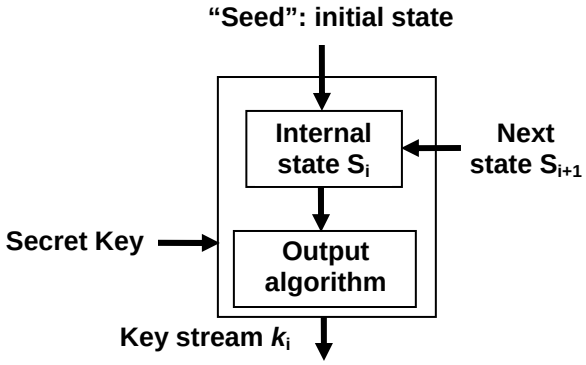


Figure 1-3 Key stream generators model of modern stream ciphers

Modern stream ciphers are also called state ciphers since the generated key stream sequences depend on the internal state of the key stream generator. The internal state is initialized with the seed and the way it is updated leads to define two kinds of stream ciphers: synchronous stream ciphers and self-synchronous stream ciphers.

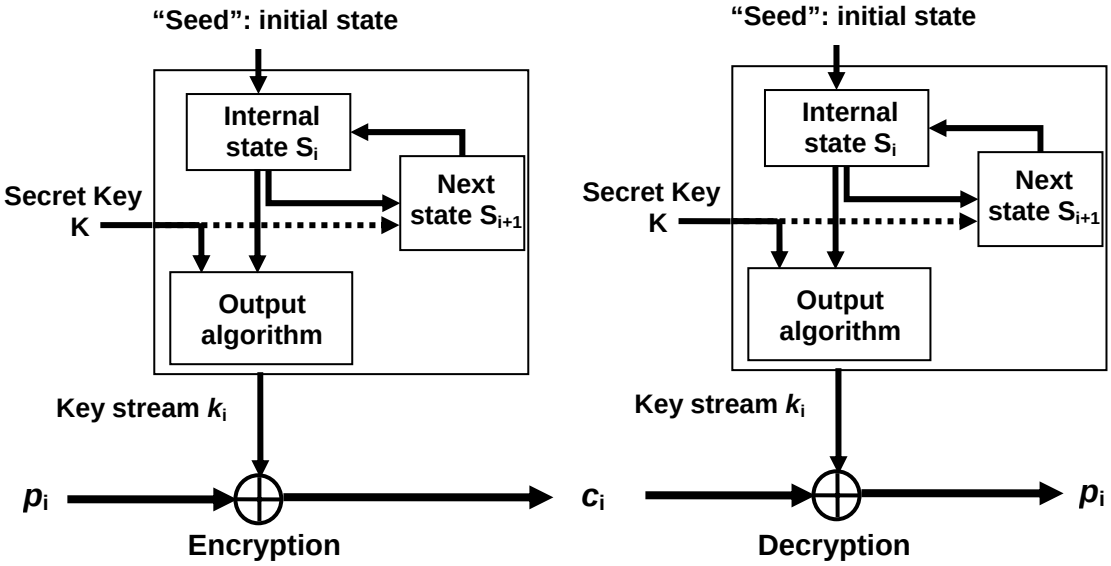


Figure 1-4 Synchronous stream ciphers

Concerning *synchronous stream ciphers* the internal state of the key stream generator changes independently of the ciphertext, it is only a function of the previous state and possibly of the secret key (Figure 1-4). Therefore, the sender and the recipient of an encrypted message must be synchronized: each state - defining a given key stream sequence - must correspond to the same section of the message during encryption and decryption. If some

encrypted message bits are lost during transmission, the key stream will be shifted in comparison to the ciphertext and the result of the decryption will be wrong. Additional mechanisms must be foreseen to re-synchronize the system and to avoid impacting the whole message deciphering. An advantage of such a class of stream ciphers concerns error propagation. If an error occurred on one or several bits during transmission, only the decryption of this or those bits will produce a wrong result. Unfortunately, this advantage becomes a drawback from a security point of view. Indeed an adversary, who changes bits in the ciphertext, knows how the plaintext is impacted.

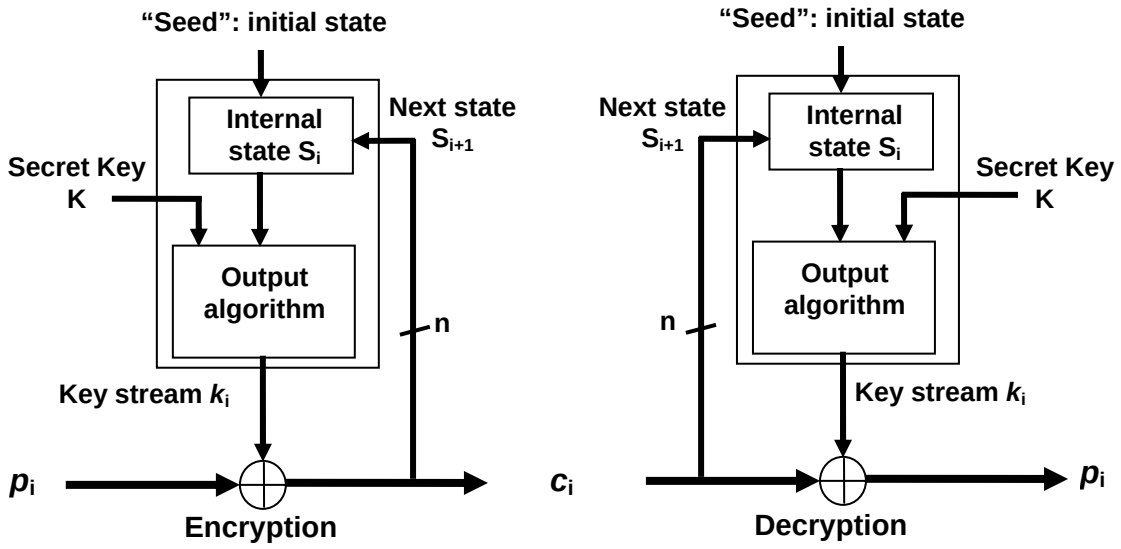


Figure 1-5 Self -synchronous stream ciphers

Concerning *self-synchronous stream ciphers*, the evolution of the internal state of the key stream generator depends on the *n* previous ciphertext bits (Figure 1-5). Therefore on decryption, the key stream generator will be automatically synchronized with the key stream generator used during encryption after receiving *n* bits of ciphertext. Such a class of stream ciphers propagates transmission errors but in a limited way. Suppose that an error occurred and that the erroneous bits were used to decide the next internal state of the key stream generator. As internal states depend only on the *n* previous ciphertext bits, only the next key stream sequence and, as a consequence, the decryption of the corresponding ciphertext will be impacted by the error.

Stream ciphers are usually constructed by using block ciphers as output algorithms. Examples of stream ciphers are therefore presented in the section dedicated to block cipher modes of operation (3.1.3.6.).

1-3.1.2.4. Advantages and Drawbacks

The main advantages of stream ciphers are their speed and their low hardware complexity. In addition, they are relevant for applications for which error propagation is not acceptable (synchronous stream ciphers) or must remain limited (self-synchronous stream ciphers). However designing a secure stream cipher is a difficult task. The fact that two key streams must never be used twice in the same cryptosystem is one security critical point. Suppose that two plaintext messages P_1 and P_2 are encrypted with the same key stream K_S, the resulting ciphertexts will be respectively $C_1 = P_1 \oplus K_S$ and $C_2 = P_2 \oplus K_S$. Hence, $P_1 \oplus P_2 = C_1 \oplus C_2$, which gives information to cryptanalysts to perform attacks [16]. Moreover, a key stream generator using a secret key K and always initialized with the same seed will produce each time the same key stream. Therefore, an attacker may replay an old ciphertext with the certainty that it will be correctly decrypted. Such attacks highlight the fact that the seed or the secret key must be changed on each initialization of the output algorithm.

1-3.1.3. Block Ciphers

1-3.1.3.1. Principle

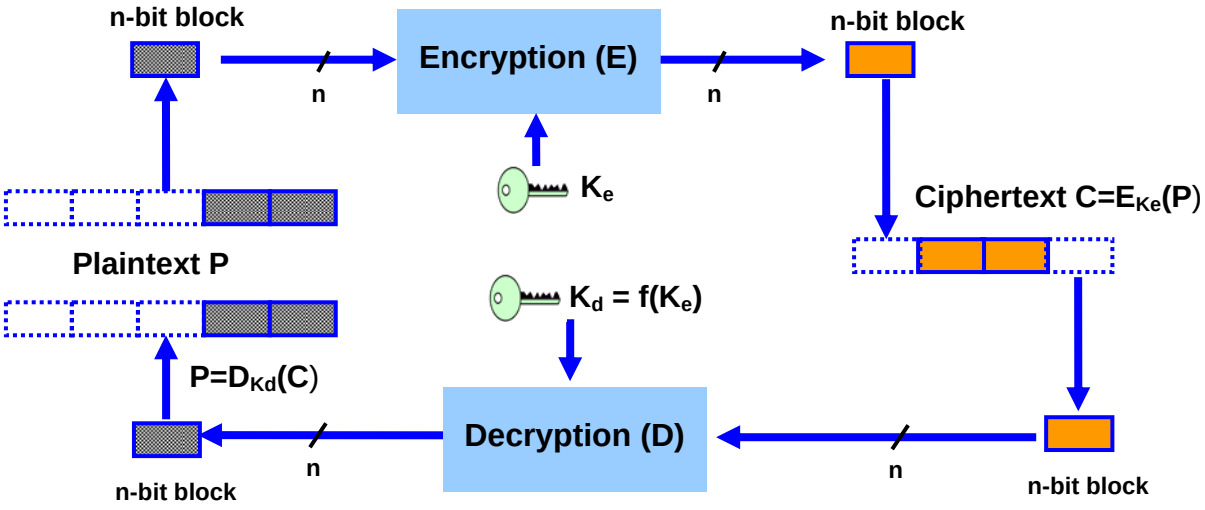


Figure 1-6 Block ciphers principle

A block cipher is an encryption scheme which first splits the plaintext into n-bit blocks, and then encrypts each block separately using a secret key K_e (Figure 1-6). The result of the encryption of an n-bit block of plaintext is a ciphertext block of the same length. The

decryption process works analogously deciphering the ciphertext *n*-bit block by *n*-bit block using K_d.

1-3.1.3.2. The Shannon Principles

Shannon [19] identified two properties for cipher operations to be secure: *confusion* and *diffusion*. In Shannon's original definitions, *confusion* makes the relation between the key and the ciphertext as complex as possible. An ideal confusion makes each bit of the ciphertext dependent on all bits of the key. *Diffusion* refers to the property that redundancy in the statistics of the plaintext is dissipated in the statistics of the ciphertext. Theoretically, to reach such a goal, each bit of the ciphertext must depend on all bits of the plaintext.

In the same paper, Shannon introduced the notion of substitution-permutation networks also called product ciphers. Product ciphers iterate simple operations like substitution – to add confusion - and permutation – to add diffusion. The objective of combining simple transformations which do not individually provide a high level of security is to obtain strong cipher.

Most modern block ciphers are based on the product cipher principle.

1-3.1.3.3. Block Cipher Structures

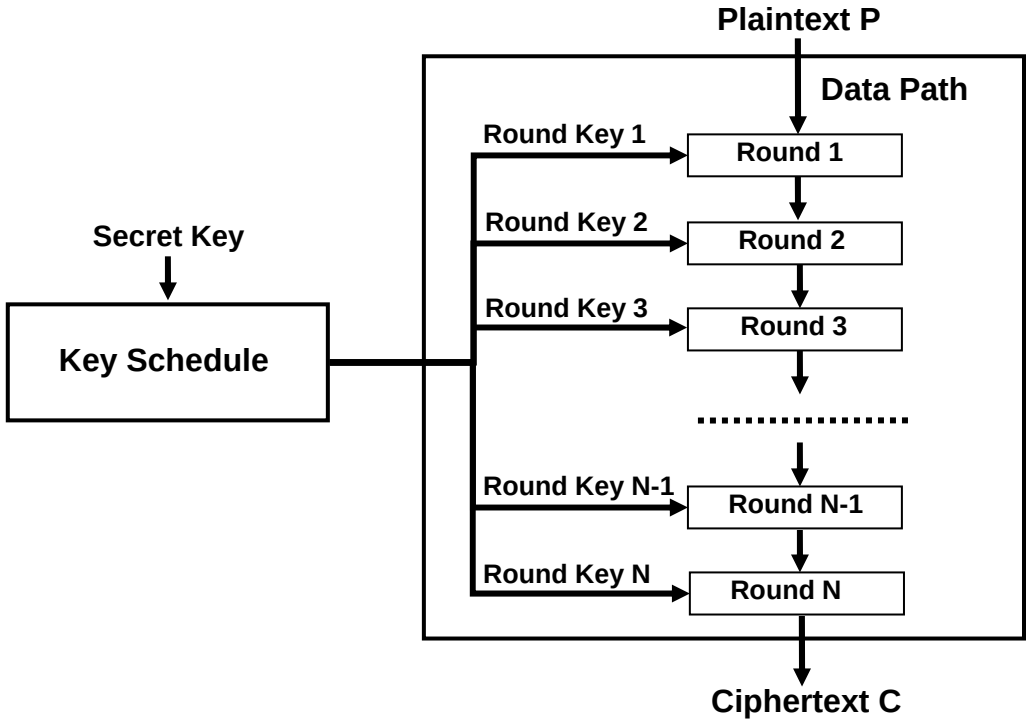


Figure 1-7 Block ciphers structure

A block cipher is divided into two parts: the key schedule and the data path.

The data path follows the product cipher principle: it is composed of a function (or a set of functions) called a round that is repeated a fixed number of times. It takes the plaintext as input, and outputs the ciphertext after the fixed number of rounds *Nr*.

The key schedule processes the secret key and derives from it the round keys used in each data path round. The derivation of the key is necessary to add confusion: it increases the dependency of each bit of the ciphertext upon all bits of the secret key.

The first round of the data path (Figure 1-7) takes the plaintext and the first round key as inputs, and then the following rounds take the previous round output and the corresponding round key as inputs.

In order to illustrate block cipher theory, the following section describes the last block encryption algorithm standardized by the NIST (National Institute of Standard and Technology): AES (Advanced Encryption Standard).

1-3.1.3.4. Example: AES

The Rijndael [15] algorithm was developed by Joan Daemen and Vincent Rijmen to answer the call for bock cipher standardization from the NIST. It was adopted as the new standard after a standardization process of five years. It was then called AES [20] (Advanced Encryption Standard) and replaced the famous DES [21] (Data Encryption Standard) for which the key length became too small to resist a brute force attack[3] [22].

The AES algorithm processes 128-bit data blocks using cipher keys with lengths of 128, 192 and 256 bits. In the following, the AES structure is detailed only for 128-bit keys; such a configuration is referred to as AES-128. For more information on the other configurations refer to [15, 20].

The encryption data path:

The AES-128 data path is an iteration of 10 rounds after a first XOR operation between the plaintext block and the secret key. A round is made of four operations: SubBytes, ShiftRows, MixColumns and AddRoundKey, except the last round of the data path which does not include the MixColumns transformation (Figure 1-8).

[3] A brute force attacks consists in trying all possible keys to find out all plaintext / ciphertext pairs, see section 3.3

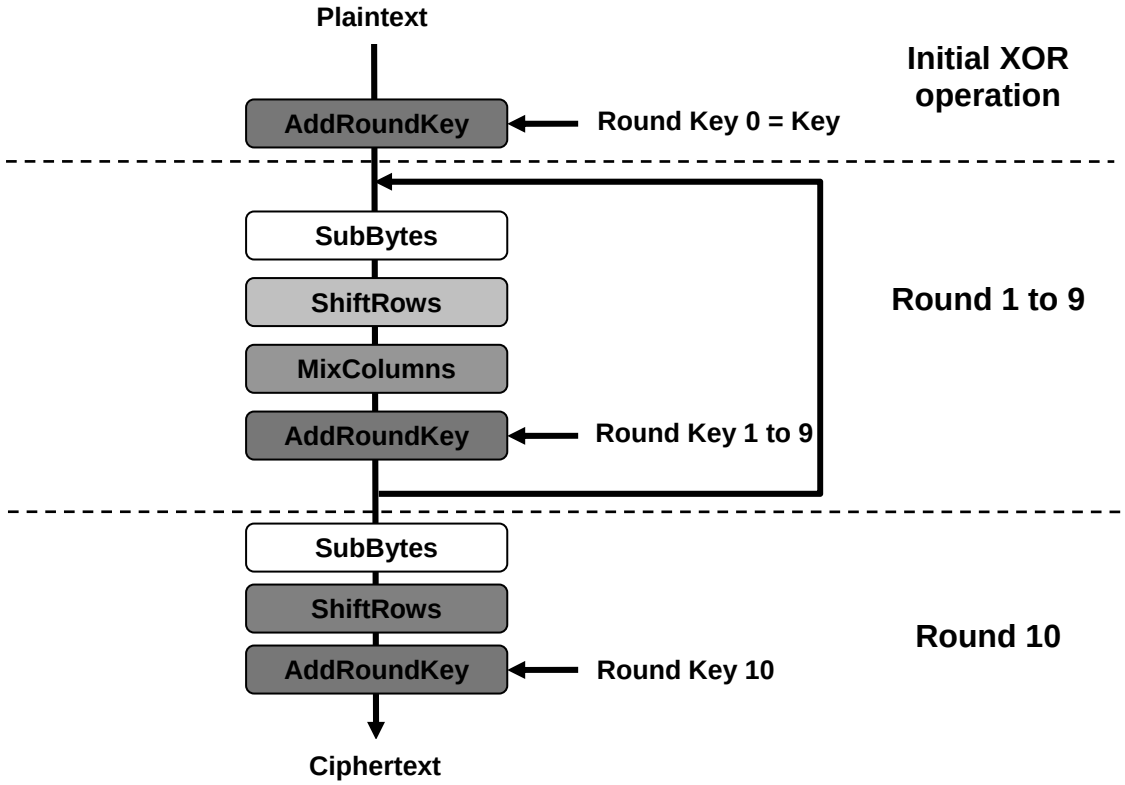


Figure 1-8 AES Encryption process: 10 rounds of 4 operations: SubBytes - ShiftRows - MixColumns - AddRoundKey

Those operations are performed on a 4x4 array of bytes called the State. At the start of the cipher the State array is filled by the plaintext bytes – in_0, in_1, in_2, … , in_{15} – as shown in Figure 1-9 (taken from [20]). $S_{r,c}$ is a byte of plaintext in the State array, the suffixes *r* and *c* refer respectively to the row and the column to which it belongs. Then the State is processed by the data path and at the end of the last round it is copied to the output array (Figure 1-9) to produce the ciphertext - out_0, out_1, out_2, … , out_{15}.

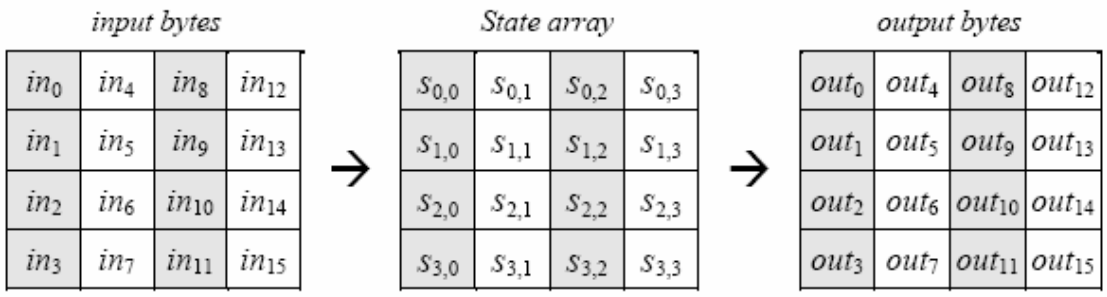


Figure 1-9 State array input and output

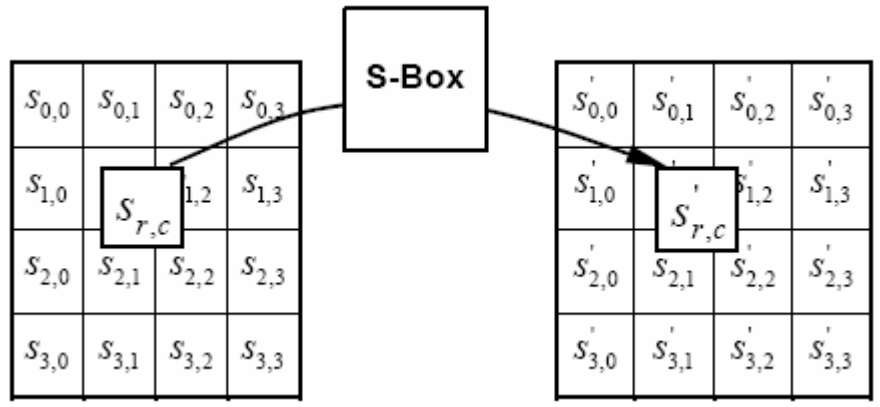


Figure 1-10 SubBytes: the S-box is applied on each byte of the State array

		y														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 1-11 The S-box: substitution value for a byte *xy*

SubBytes is a non-linear byte substitution applied on each byte of the State array using a substitution table, the S-box (Figure 1-10 - taken from [20]). Each byte $S_{r,c}$ of the State array is represented by two digits *xy* which are used to address the S-box (Figure 1-11 - taken from [20]). The pointed value replaces $S_{r,c}$ in the State array. The S-box is constructed by performing two transformations: taking the multiplicative inverse of the State array in the Galois Field GF(2^8) and applying an affine transformation over GF(2). Non-linearity is provided by the first one which minimizes the correlation between the input and the output and the second one increases the algebraic structure complexity of the operation. Both transformations are invertible; hence the S-box is invertible.

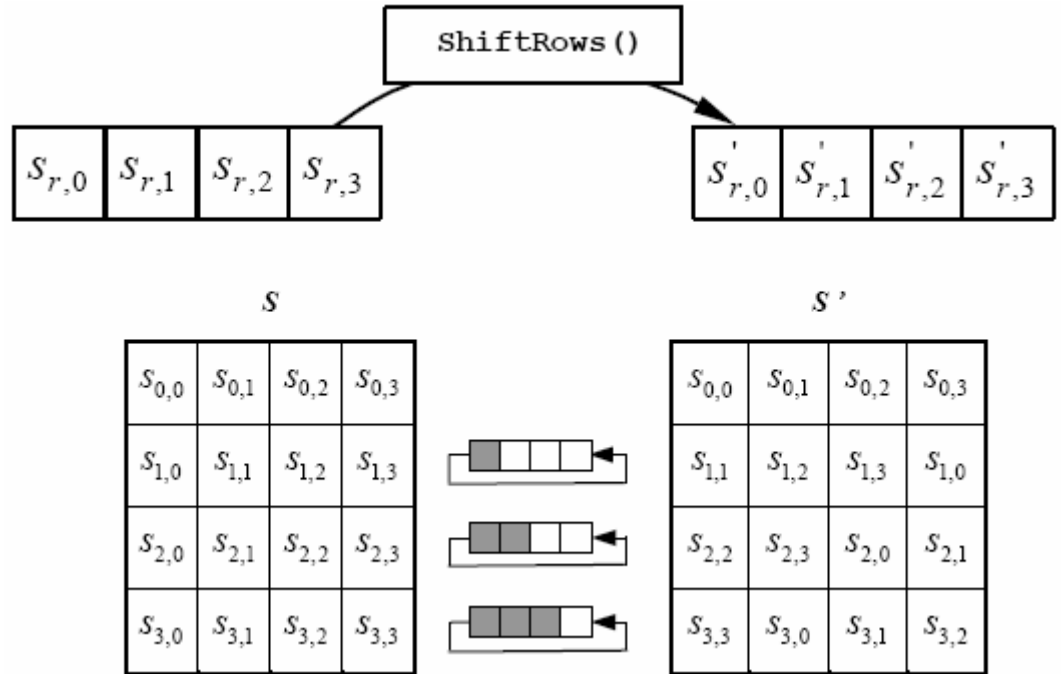


Figure 1-12 ShiftRows: the last three rows of the state array are cyclically shifted of *r*-byte

The *ShiftRows* transformation acts on the last three rows of the State array. They are cyclically shifted of a number equal to the value of the suffix *r* as shown in Figure 1-12 (taken from [20]). The objective of ShiftRows is to introduce diffusion through rows.

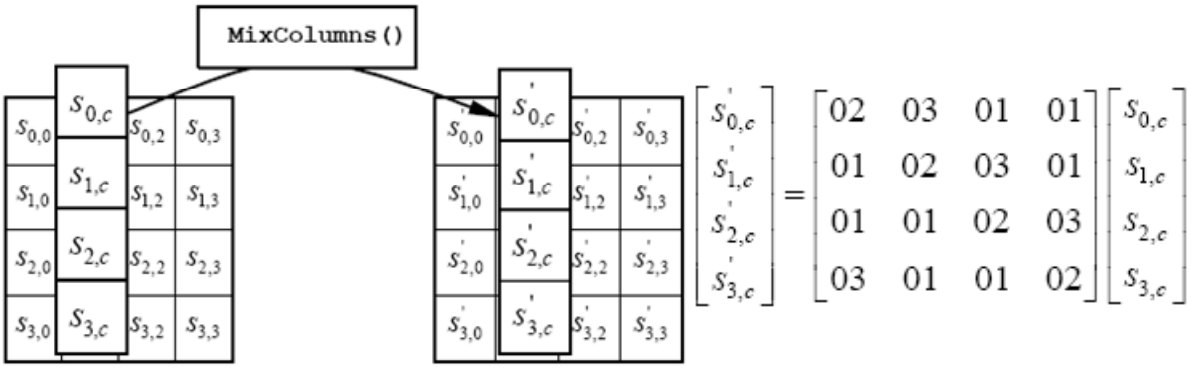


Figure 1-13 MixColumns multiplies each column of the State array by a constant matrix

The *MixColumns* transformation multiplies each column of the State array by a constant matrix (Figure 1-13 – taken from [20]). Considering each column as a polynomial over GF(2^8), such transformation amounts to a multiplication modulo $x^4 + 1$ with $a(x) = \text{x03}\ x^3 + \text{x01}\ x^2 + \text{x01}\ x + \text{x02}$. The objective of MixColumns is to introduce diffusion through columns.

The *AddRoundKey* transformation is a simple bitwise XOR operation between each column of the State array and the corresponding column $W_{n,c}$ in the Round Key matrix (provided by the key schedule) with *n* indicating the number of the round (Figure 1-14 – taken from [20]):

$$[S'_{0,c}, S'_{1,c}, S'_{2,c}, S'_{3,c}] = [S_{0,c}, S_{1,c}, S_{2,c}, S_{3,c}] \oplus W_{n,c} \qquad \text{for } 0 \le c \le 3$$

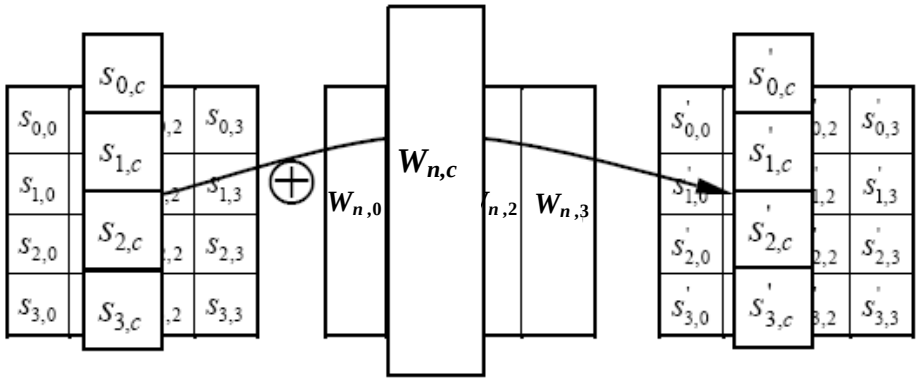


Figure 1-14 AddRoundKey: Xor operation between each column of the State array and the corresponding column in the round key matrix

The key schedule:

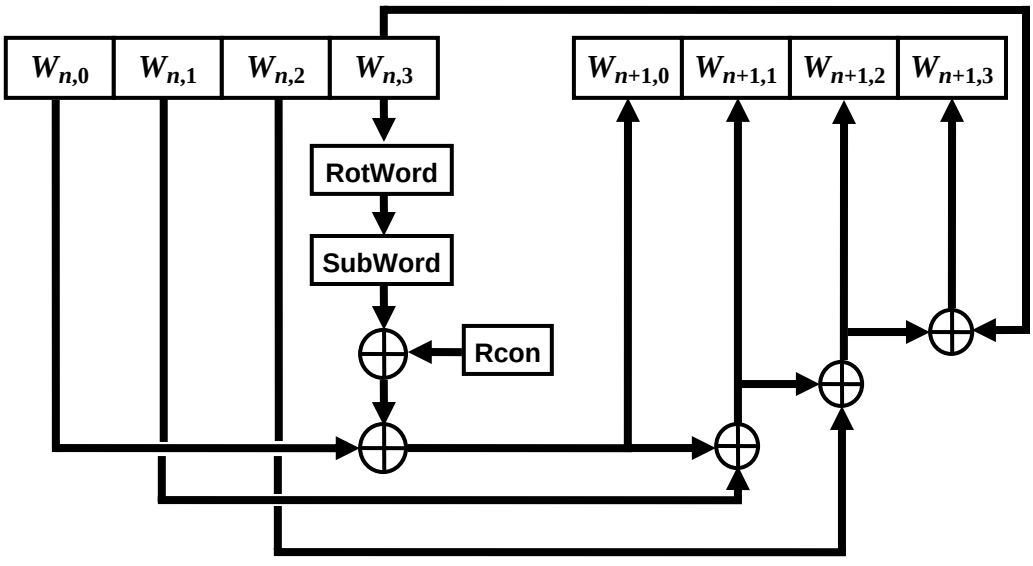


Figure 1-15 Computation of a round key

The key schedule provides the Round key required in each round of the data path; such a process is called the *key expansion*. The secret key is said to be expanded in *Nr* (number of rounds) round keys. For the AES-128 *Nr* is equal to 10. The architecture of the key schedule

is depicted in Figure 1-15. Round key 0 is the secret key and is used for the first XOR operation with the plaintext before the first round of the data path.

RotWord and *SubWord* are two transformations used in the key schedule. *RotWord* takes a word ($a_0\ a_1\ a_2\ a_3$) as input, cyclically permutes the bytes composing it, and outputs the word ($a_1\ a_2\ a_3\ a_0$). *SubWord* takes a word as input and applies the S-box to each of the four bytes. *Rcon* [1 to 10] is a constant array of ten words [20].

A Round key (n) is composed of four words $W_{n,i}$, with n as the suffix of the corresponding Round key and i as the position of the word in the Round Key.

To compute Round key (n+1) ($W_{n+1,0}\ W_{n+1,1}\ W_{n+1,2}\ W_{n+1,3}$), four operations are chained (Figure 1.15). The last word $W_{n,4}$ of the current Round key (n) ($W_{n,0}\ W_{n,1}\ W_{n,2}\ W_{n,3}$) is first processed by *RotWord*. Then the resulting word is applied to SubWord and a XOR operation is performed between the output of *SubWord* and *Rcon* [n+1]. Finally, the first word of Round Key (n+1), $W_{n+1,0,}$ is obtained by a XOR operation between the first word of Round key (n), $W_{n,0}$, and the result of the last operation. The other words of Round key (n+1), $W_{n+1,1}\ W_{n+1,2}\ W_{n+1,3}$, are computed with XOR operations between $W_{n+1,i-1}$, and $W_{n,i}$.

The decryption:

All the transformations described above are invertible. Hence, the decryption is obtained by applying the inverse functions *InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey* in reverse order. *AddRoundKey* remains the same as it involves only XOR operations. The round keys are generated in the same way, however the key schedule takes as input the decryption key that is the last Round key (Round key(10)) of the key schedule in the encryption process.

AES is fast both in software and in hardware. For example, the AES implementation on ASIC presented in [23] shows that the encryption of one 128-bit block is done in 11 clock cycles at 330 MHz.

1-3.1.3.5. Advantages and Drawbacks

Block ciphers are usually slower than stream ciphers but are more popular. The main reason is the fact that they are simple to understand and to implement. Moreover block ciphers are well known and studied since they are not used only to ensure data confidentiality. As it will be presented in section 1-4, block ciphers are a building block for other

cryptographic algorithms like the ones dedicated to data integrity and authentication security services.

1-3.1.3.6. Modes of Operation

Block ciphers offer a solution to encrypt a given *n*-bit plaintext block into an *n*-bit ciphertext block. Modes of operation are the recommended ways to use block ciphers to encrypt a message longer than the cipher block size *n*.

In this sub section, the principles of five block cipher modes of operation are exposed: ECB (Electronic Code Book), CBC (Cipher Block Chaining), CTR (Counter), OFB (Output FeedBack) and CFB (Cipher FeedBack). For detailed description of such modes – particularly OFB and CFB – refer to [24].

The NIST has recently started a standardization process for Authenticated Encryption modes with the objective to provide both data confidentiality and authentication. Such modes are described and discussed in a dedicated section in chapter 3 (section 3-33).

In the following, encryption modes are described considering a block cipher processing *n*-bit block. The secret key *K* is considered to be the same for encryption and decryption. The encryption and decryption operations using *K* are respectively noted E_k and D_k. Moreover, the length of the message to encrypt is supposed to be a multiple of the ciphered block size *n*. When it is not the case, message padding is required; however this issue is not exposed here, for details refer to [16][24].

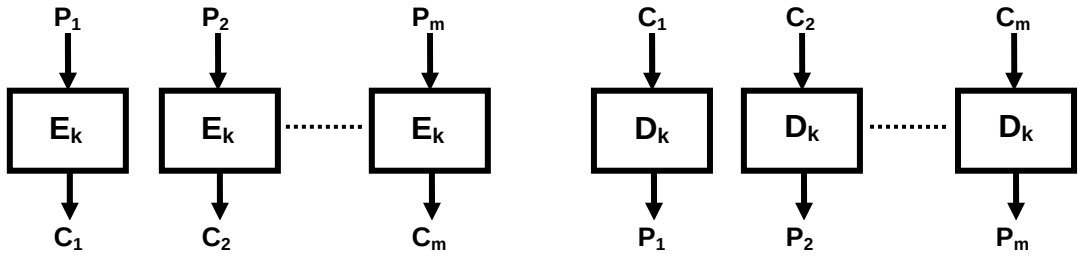
a. ECB - Electronic Code Book

ECB mode is the direct application of block cipher principle: plaintext blocks P_1, P_2, P_3, …, P_m composing a message *M*, are encrypted one at a time and independently from one another (Figure 1.16a). The resulting ciphertext is C_1, C_2, C_3, …, C_m with

$$C_i = E_k(P_i) \qquad \forall\, i \text{ such as } 1 \le i \le m$$

The decryption follows the same principle (Figure 1-16b), thus:

$$P_i = D_k(C_i) \qquad \forall\, i \text{ such as } 1 \le i \le m$$



(a) Encryption **(b) Decryption**

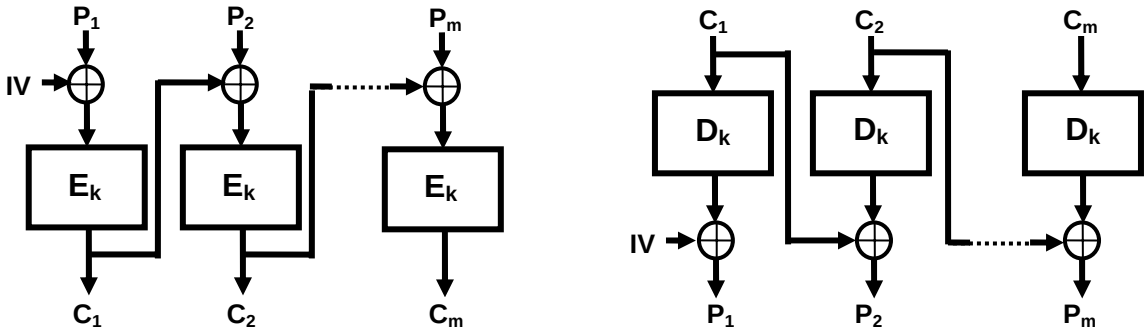
Figure 1-16 The ECB (Electronic Code Book) mode

The fact that the encryption of a same plaintext block twice results in the same ciphertext block is the main drawback of ECB mode from a security point of view. A message containing repetitive patterns must not be encrypted with such a mode because an adversary can deduce when the same information occurs twice or is transmitted twice. A way to improve the security of the ECB mode is to include random bit in each plaintext block before encryption [16].

With the ECB mode, encryption and decryption processes are fully parallelizable.

Concerning error propagation, the effect of ECB mode is limited. If one bit is erroneous in an encrypted message, only the decryption of the corresponding ciphered block is impacted.

b. CBC - Cipher Block Chaining



(a) Encryption **(b) Decryption**

Figure 1-17 The CBC (Cipher Block Chaining) mode

In CBC mode, each plaintext block P_i of a message M (P_1, P_2, P_3, …, P_m) is combined with the previous ciphered block C_{i-1} by a XOR operation prior to encryption (Figure 1-17a), thus:

$$C_i = E_k(P_i \oplus C_{i-1}) \qquad \forall\, i \quad \text{such as } 2 \le i \le m$$

An initialization vector (*IV*) is required to encrypt the first plaintext block P_1:

$$C_1 = E_k(P_1 \oplus IV) \quad \text{for} \quad i = 1$$

The decryption process is therefore (Figure 1-17b):

$$P_i = \begin{cases} D_k(C_i) \oplus IV & \text{for} \quad i = 1 \\ D_k(C_i) \oplus C_{i-1} & \forall\, i \quad \text{such as} \quad 2 \le i \le m \end{cases}$$

The main advantage of CBC compared to ECB is that a plaintext block encrypted twice gives two different ciphertext blocks as the encryption depends on the previous ciphertext block. However, a complete message encrypted twice in CBC mode with a given key results in the same ciphertext if the same *IV* is used. A way to overcome such a drawback is to change the *IV* for each encrypted message. Ideally, this *IV* should be secret. Therefore it could be randomly generated for each message to encrypt, encrypted in ECB mode and sent with the message.

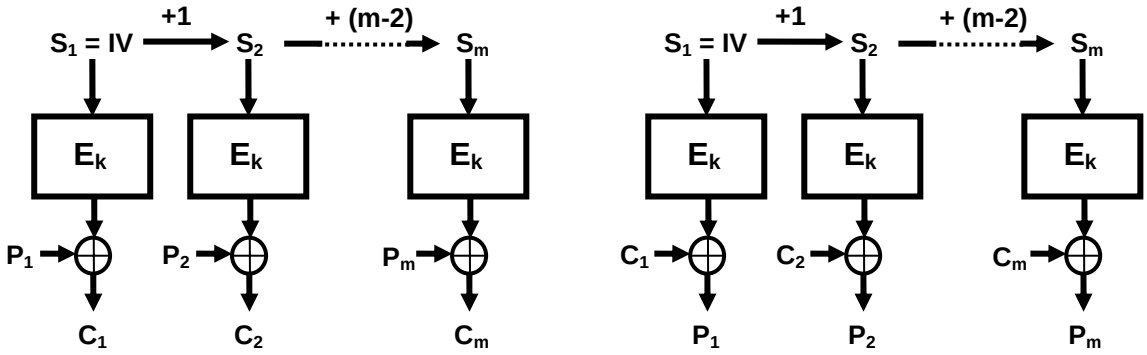
CBC encryption is performed in series (Figure 1-17a) since such a process requires the previous ciphertext block to start. On decryption, CBC is parallelizable because all required information could be available: the current and the previous ciphertext blocks and the secret key.

The CBC mode has a limited effect on error propagation. If one bit is erroneous after transmission of the encrypted message, the decryption of the corresponding ciphertext block results in a wrong plaintext block and one bit of the following plaintext block will be affected – because of the XOR operation with the erroneous bit.

c. CTR – Counter

CTR is a block cipher mode which is used as a stream cipher (like OFB and CFB). CTR is a synchronous stream cipher (Figure 1-4): the output algorithm is the block cipher and the internal state *S* of the key stream generator is a counter initialized with an *IV* (the seed) and incremented for each new plaintext block. Considering a message *M* composed of *m* plaintext block P_i with ($1 \le i \le m$), the CTR encryption process (Figure 1-18a) works as follows:

$$C_i = E_k(S_i) \oplus P_i \quad \text{with} \quad S_i = \begin{cases} IV & \text{for} \quad i = 1 \\ S_{i-1} + 1 & \forall\, i \quad \text{such as} \quad 2 \le i \le m \end{cases}$$



(a) Encryption **(b) Decryption**

Figure 1-18 The CTR (Counter) mode

As a stream cipher, for decryption (Figure 1-18b) the same key stream - as the one used for encryption - must be generated to perform the XOR operation with the ciphertext (C_1, C_2, C_3, …, C_m), therefore:

$$P_i = E_k(S_i) \oplus C_i \quad \text{with} \quad S_i = \begin{cases} IV & \text{for} \quad i = 1 \\ S_{i-1} + 1 & \forall\, i \quad \text{such as} \quad 2 \le i \le m \end{cases}$$

Like the CBC mode, in order to avoid the encryption of the same message resulting in the same ciphertext, the *IV* must be different and ideally secret for each message to encrypt with the same secret key *K*. Moreover, the key stream generated for each plaintext block depends only on the counter value *S* for a given *K*. Hence, the same value of *S* must never be used twice; otherwise the same key stream sequence is generated for two different plaintext blocks, leading to the weakness described in 3.1.2.4. The *IV* must be carefully selected on each encryption to circumvent such a security hole.

As CTR is a synchronous stream cipher, there is no error propagation (see section 1-3.1.2.3).

The CTR mode is fully parallelizable on encryption and on decryption as the key stream generation only depends on the counter value *S*.

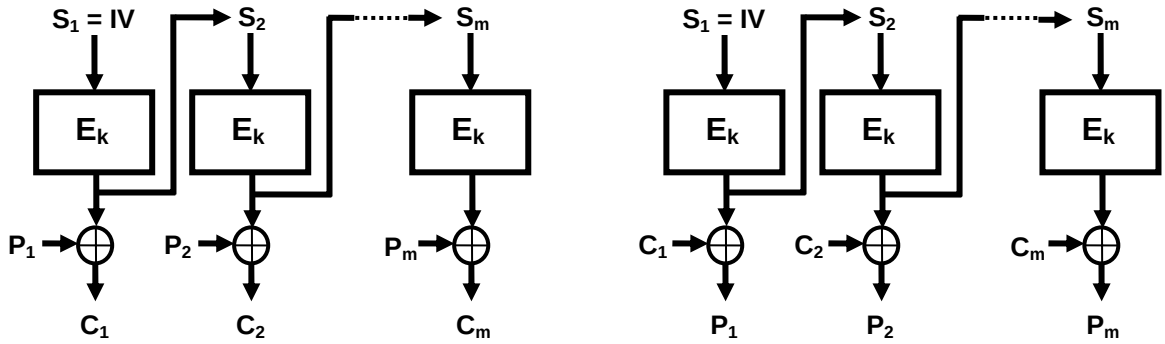
d. OFB - Output FeedBack

The OFB mode is a synchronous stream cipher, like CTR (Figure 1-4). The output algorithm is the block cipher and the internal state *S* of the key stream generator is defined by the result of the block encryption of the previous internal state. Such a mode requires an *IV* (the seed) to initialize the internal state. Considering a message *M* composed of *m* plaintext block P_i with ($1 \le i \le m$), the OFB encryption process (Figure 1-19a) works as follows:

$$C_i = E_k(S_i) \oplus P_i \quad \text{with} \quad S_i = \begin{cases} IV & \text{for} \quad i = 1 \\ E_k(S_{i-1}) & \forall\, i \quad \text{such as} \quad 2 \le i \le m \end{cases}$$

For decryption (Figure 1-19b), the same key stream - as the one used for encryption - must be used to perform the XOR operation with the ciphertext, therefore:

$$P_i = E_k(S_i) \oplus C_i \quad \text{with} \quad S_i = \begin{cases} IV & \text{for} \quad i = 1 \\ E_k(S_{i-1}) & \forall\, i \quad \text{such as} \quad 2 \le i \le m \end{cases}$$



(a) Encryption **(b) Decryption**

Figure 1-19 The OFB (Output FeedBack) mode

A message encrypted twice in OFB mode results in the same ciphertext. Hence, as for the CBC and CTR modes, the *IV* must be different and ideally secret for each message encrypted with a given key.

OFB is a synchronous stream cipher therefore there is no error propagation (see section 1-3.1.2.3).

The OFB mode is not parallelizable on encryption and on decryption. Indeed, a given internal state of the key stream generator is defined only at the end of the block encryption of the previous internal state. Therefore key stream sequences are produced in series.

e. CFB - Cipher FeedBack

The CFB mode is a self synchronous stream cipher (Figure 1-5). The output algorithm is the block cipher and the internal state *S* is defined by the previous ciphertext C_{i-1}. Such a mode requires an *IV* (the seed) to initialize *S*. Considering a message *M* composed of *m* plaintext block P_i with ($1 \leq i \leq m$), the CFB encryption process (Figure 1-20a) works as follows:

$$C_i = E_k(S_i) \oplus P_i \quad \text{with} \quad S_i = \begin{cases} IV & \text{for} \quad i = 1 \\ C_{i-1} & \forall\, i \quad \text{such as} \quad 2 \leq i \leq m \end{cases}$$

For decryption (Figure 1-20b), the same key stream - as the one used for encryption - must be used to perform the XOR operation with the ciphertext, therefore:

$$P_i = E_k(S_i) \oplus C_i \quad \text{with} \quad S_i = \begin{cases} IV & \text{for} \quad i = 1 \\ C_{i-1} & \forall\, i \quad \text{such as} \quad 2 \leq i \leq m \end{cases}$$

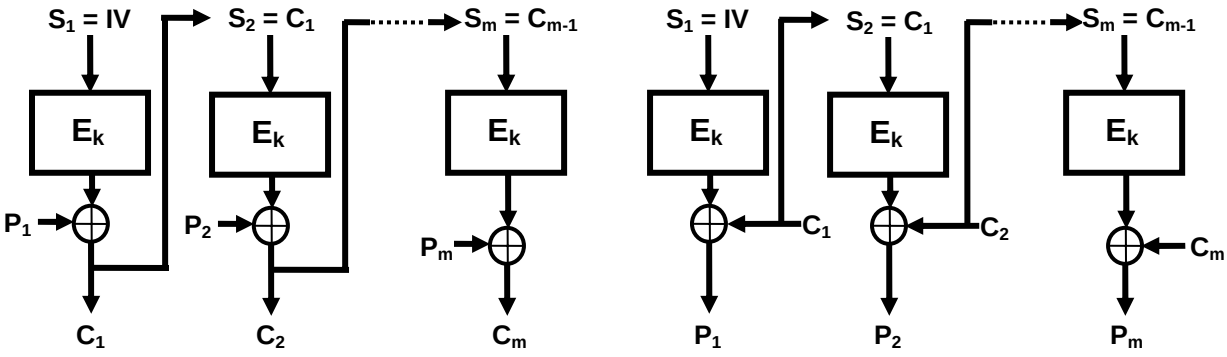


Figure 1-20 The CFB (Cipher FeedBack) mode

Concerning the choice of the *IV*, CFB implementation must follow the same recommendations as for OFB implementation.

CFB is a self synchronous stream cipher therefore error propagation is limited (see section 1-3.1.2.3).

On encryption, a given internal state of the key stream generator is defined only at the end of the previous plaintext encryption, therefore key stream sequences and plaintext encryption

are done sequentially. On decryption, CFB mode is parallelizable since this operation only requires available information – the ciphered message and the secret key- to start the computation of the key stream sequences.

The last three presented modes (CTR, OFB and CFB) may be implemented in order to generate key stream sequences shorter than *n* (the size of block processed by the underlying cipher block). Such a task is achieved by selecting the *l* leftmost bits of each key stream sequence [24] where *l* is the desired size of the key stream. Moreover, note that when modes of encryption are used as stream cipher (CTR, OFB and CFB) only the block encryption process is involved. In hardware, this allows to save silicon area.

1-3.2. Public-Key Encryption

1-3.2.1. Principle

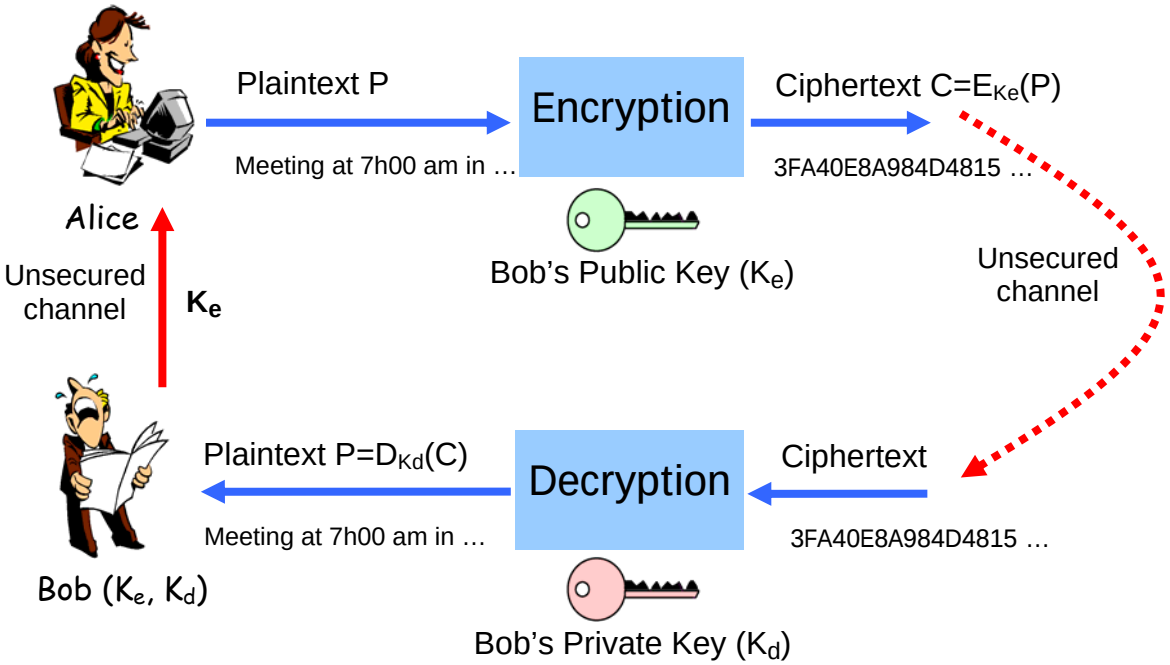


Figure 1-21 Public-key cipher principle

Contrary to symmetric-cipher, entities using cryptosystems based on public-key cipher to communicate in a confidential way do not need to agree on a secret. Such schemes are also named asymmetric-cipher because they use two different keys for encryption and for decryption: the encryption key K_e, called the public key since it can be transmitted over an unsecured channel and it does not require to be kept secret, and the decryption key K_d,

designated as the private key since only the intended recipient of an encrypted message must know it.

Take up again the example of Alice and Bob where Alice wants to send a confidential message to Bob (Figure 1-21). Bob must first compute a pair of public / private keys (K_e, K_d) and transmit the public-key K_e to Alice over an unsecured channel[4]. Then Alice encrypts her message with K_e and sends it to Bob. Finally, Bob uses K_d to decrypt the message.

The notion of public-key cryptosystems was first introduced in [25] by professors Wilfred Diffie and Martin Hellman. Such cryptosystems are based on the difficulty to solve complex mathematical problems and particularly on trapdoor one-way functions. A function f(x) = y is designated as one-way if for every x it is easy to compute f(x), while it is computationally difficult to retrieve x from y. Whereas a function f(x) = y is designated as one-way with trapdoor, if for every x it is easy to compute f(x), and that, knowing a secret – the trapdoor –, it is easy to retrieve x from y.

An example of such complex mathematical problems is the factorization of large numbers: it is easy to compute the multiplication of two large numbers but the inverse function, the factorization, is computationally very difficult to resolve. The most famous public-key algorithm, RSA [26] (for Rivest Shamir Adleman, the names of the inventor) is based on such a problem. Next section describes how RSA works.

1-3.2.2. Example: RSA

As all cryptosystems, RSA is composed of three algorithms: one for the key generation, one for encryption and one for decryption.

Key generation algorithm;

The key generation algorithm of RSA consists in computing three parameters which compose the encryption and the decryption keys K_e and K_d.

The first parameter n, called the modulus, is computed by choosing two large prime numbers, p and q, and by multiplying them together: $n = pq$.

Then, the second parameter e, called the encryption exponent, is selected so that e and (q-1)(p-1) are prime together - meaning gcd(e, (q-1)(p-1)) = 1 (gcd: greatest common divisor) – and e < (q-1)(p-1).

[4] In fact this channel must be an authenticated channel otherwise an attack described in 3.2.3 remains feasible.

Finally, the extended Euclidean algorithm [16] is used to compute the third parameter d, the decryption exponent, as follows:

$$ed \equiv 1 \bmod ((q\text{-}1)(p\text{-}1)) \text{ which is equivalent to } \exists\, k \,/\, ed + k(p\text{-}1)(q\text{-}1) = 1.$$

It results that the public key K_e consisting of e and n, and the private key K_d is equal to d.

Once the two keys are generated, the two prime numbers p and q are no longer used, therefore they must be destroyed.

Encryption:

Consider a message m represented as an integer comprised between 1 and n-1. The resulting ciphertext c of the m encryption using $K_e\,(e,n)$ is obtained with:

$$c = m^e \bmod n$$

Decryption:

The recipient of c uses $K_d\,(d)$ to decrypt it as follows:

$$m = c^d \bmod n$$

1-3.2.3. Advantages and Drawbacks

The main advantage of public-key ciphers compared to secret-key ciphers is the fact that there is no need to share a secret prior to establishing the communication. The encryption key may be publicly known but its authentication is required, otherwise a man-in-the-middle attack is feasible. Such an attack involves (Figure 1-22) an adversary who monitors an unsecured channel, intercepts a message M and possibly some meta data X useful for security purposes, analyses them, and exchanges them by chosen data (M'; X'). Here, the adversary may catch Bob's public key and replace it by its own. Alice encrypts her message with the fake public key and sends the message over the unsecured channel. The adversary retrieves the encrypted message, and is able to decrypt it with his private-key. Such an attack challenges the entity authentication and is circumvented by using public key certificates and a Public Key Infrastructure (PKI). A PKI is an infrastructure which supports the distribution of authenticated public keys. However, entity authentication is not the topic of this thesis; hence in the following public keys are considered authenticated - for information on PKI refer to [24].

In addition, public-key ciphers are very slow because they are based on complex mathematic concepts. For example, in hardware an RSA decryption takes up to 260 000 cycles at 100MHz to encrypt a 1024-bit plaintext block [27].

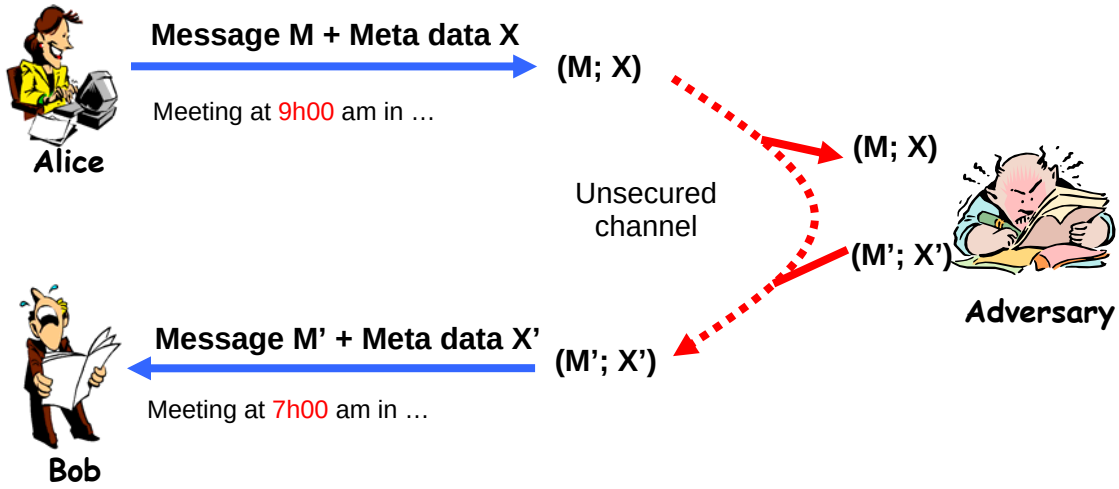


Figure 1-22 Man-in-the-middle attack principle

1-3.3. Security of Encryption Techniques

The attacks conducted on an encryption scheme might have two objectives: *key recovery* and *message recovery*.

The objective of the first attack is to recover the secret-key or the private-key used respectively in a symmetric cipher or in an asymmetric cipher. The most well-known practical attack for key recovery is the brute force attack. It consists in trying all possible keys until finding out the one allowing to retrieve the plaintext from any ciphertext. Therefore, the longer the key is, the better for the encryption algorithm in term of robustness.

The goal of message recovery attacks is to draw some conclusions on plaintexts by observing ciphertexts. In such attacks, it is supposed that the adversary knows some pair of corresponding plaintext / ciphertext, and the public key in the case of asymmetric encryption. A cryptosystem is immune against message recovery if it is semantically secure, meaning an eavesdropper is not able to determine if a given ciphertext is the encryption of one plaintext or another.

Previous adversary objectives (message recovery and key recovery) rely practically on passive attacks which could be classified by considering the kind of information the adversary knows. The following definitions have been taken from [16]:

1. A *ciphertext-only attack* is one where the adversary (or cryptanalyst) tries to deduce the decryption key or plaintext by only observing ciphertext. Any encryption scheme vulnerable to this type of attack is considered to be completely insecure.
2. A *known-plaintext attack* is one where the adversary has a quantity of plaintext and corresponding ciphertext. This type of attack is typically only marginally more difficult to mount.
3. A *chosen-plaintext attack* is one where the adversary chooses plaintext and is then given corresponding ciphertext. Subsequently, the adversary uses any information deduced in order to recover plaintext corresponding to previously unseen ciphertext.
4. An *adaptive chosen-plaintext attack* is a chosen-plaintext attack wherein the choice of plaintext may depend on the ciphertext received from previous requests.
5. A *chosen-ciphertext attack* is one where the adversary selects the ciphertext and is then given the corresponding plaintext. One way to mount such an attack is for the adversary to gain access to the equipment used for decryption (but not the decryption key, which may be securely embedded in the equipment). The objective is then to be able, without access to such equipment, to deduce the plaintext from (different) ciphertext.
6. An *adaptive chosen-ciphertext attack* is a chosen-ciphertext attack where the choice of ciphertext may depend on the plaintext received from previous requests.

The robustness of an encryption scheme is evaluated by respecting the first and the second Kerckhoffs principles which state that an algorithm must be unbreakable in practice[5] if not theoretically unbreakable and that it must be publicly known. This allows the research community to perform cryptanalysis and to check their accordance with the first Kerckhoffs principle. The sole cryptosystem proved secured is the One Time Pad.

The encryption is mainly a countermeasure against passive attacks. However, active attacks driven during transmission – e.g. the man-in-the-middle attack as depicted in Figure

[5] A cryptosystem is said broken if an adversary succeeds in a key recovery or a message attack in a time slot smaller than the one for which the secret must be kept.

1.22 – which allows message deletion, data injection or replay must be also prevented. One objective of active attacks might be to constrain the search space – for instance on a brute force attack – when the ultimate adversary goal is message or key recovery.

Countermeasures against active attacks are provided by data authentication or integrity checking techniques. Those techniques are presented in the next section.

1-4. Data Integrity Checking Techniques

This section will first present the principle of integrity checking processes of a message transmitted over an unsecured channel. Then, the cryptographic functions, called hash functions, used in such processes will be defined. However, this study focuses on mechanisms useful in our domain, i.e. store-and-retrieve application, therefore refer to [16] for a detailed definition of hash functions. Finally, integrity checking schemes and their respective objectives will be described.

1-4.1. Integrity Checking Process Principle

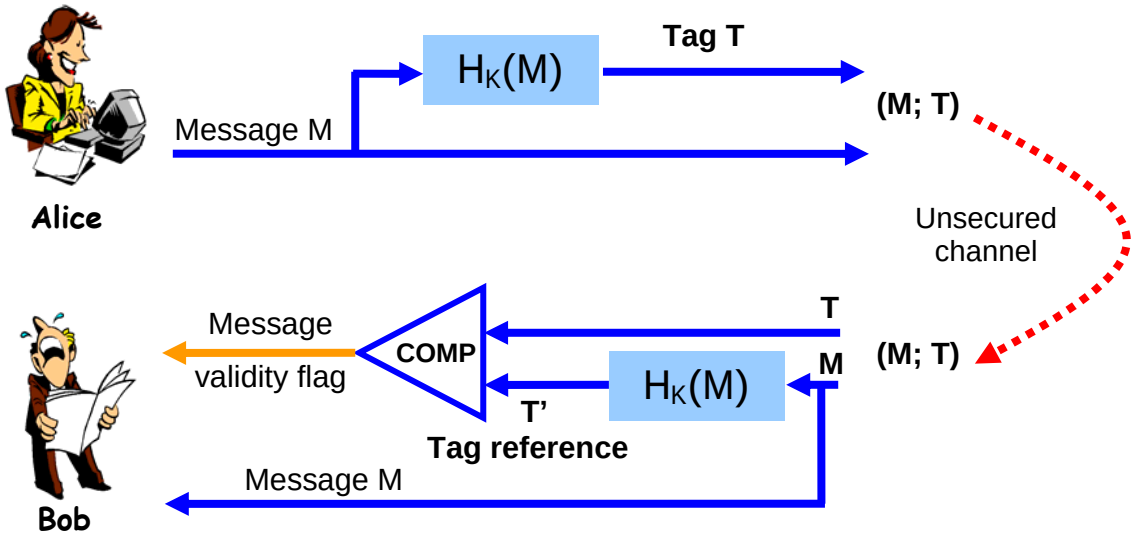


Figure 1-23 Integrity checking process principle

In order to present this principle, the example of Alice and Bob is taken up again (Figure 1-23). This time Alice sends a message *M* to Bob and needs the formal assurance that it will not be corrupted during transmission. Thus, Alice first computes a value *T*, called the tag, over the

message using a specific black-box keyed-function H_K. Then Alice sends T along with M. The role of the tag is to be a compact representative image of the message content and of its origin. Bob receives the pair (M; T), computes a tag T', called the tag reference, over M using the same black-box keyed-function H_K. If T' is equal to T, Bob has the certainty that M has not been tampered with during transmission.

Classic attacks led on integrity checking schemes are called *forgery attacks*. A forgery attack is successful when an adversary is able to find out the correct tag for a message of his choice.

Note that the purpose of the principle presented above is only to give an overview of how integrity checking works; however, the implemented schemes can be slightly different depending on the underlying cryptographic functions involved in the tag computation. Those schemes are described in the following.

The integrity checking mechanisms use cryptographic algorithms called *hash functions*. Most of the time, such functions are not used only to provide data integrity but are essential from a security point of view. Thus, the following section defines hash functions.

1-4.2. Hash Functions

A hash function h has at least two properties: compression and ease of computation. This means that given an input x of an arbitrary finite length, it is easy to compute the output $h(x)$, called hash or digest, which has a fixed bit length n.

The minimum objective of hash functions is to give a compact representative image of a message. Such an image must be ideally unique to identify the message. However the direct implication of the compression property is the fact that there are more possible inputs than outputs. That is why a hash function h may have the three following properties:

- Preimage resistance: given y, an output of h, it is computationally infeasible to retrieve the corresponding input x such as $h(x) = y$.
- Second preimage resistance: given x, an input of h, and y the corresponding output - meaning $h(x) = y$ - it is computationally infeasible to find a second input x' such as $h(x') = y$.
- Collision resistance: it is computationally infeasible to find two inputs x and x' which have the same image through h, meaning $h(x) = h(x')$.

The collision resistant property implies the second preimage resistance. Ideally, a hash function should have the three previous properties.

Hash functions are divided into two families: unkeyed hash functions and keyed hash functions.

1-4.3. Unkeyed Hash Functions a.k.a. Modification Detection Codes (MDC)

1-4.3.1. Principle

Unkeyed hash functions have a single input, the message to hash, and produce an *n*-bit digest. They are also called Modification Detection Codes (MDC) algorithms as their goal is only to provide a compact representative image of the input message.

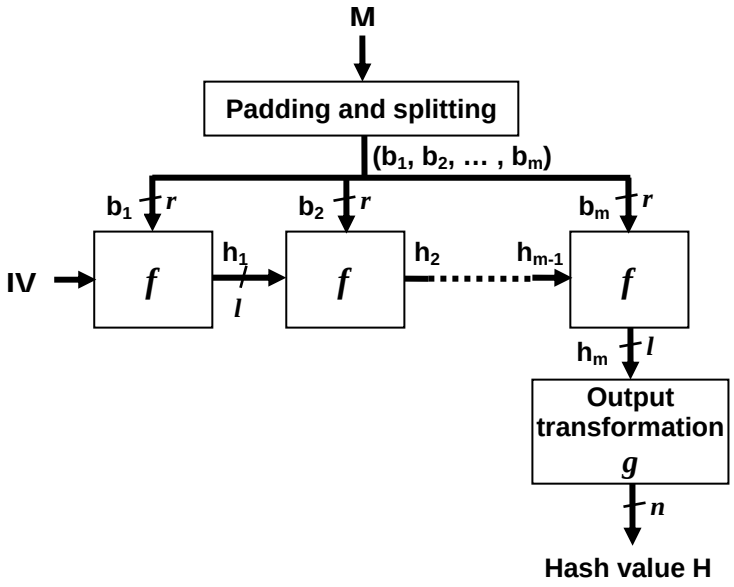


Figure 1-24 Model of iterative hash functions

Most of unkeyed hash functions are designed as iterative processes as depicted in Figure 1-24. Their input is an arbitrary length message M which is first divided into m fixed-length r-bits block b_i with $1 \le i \le m$. If the length of M is not a multiple of r, a padding method is used to attain such size. Those methods are not dealt with here; for more information, refer to [24]. Then, each block is processed in series by a compression function f. Hence, f is solicited m times. f takes as inputs its previous result and a block b_i; an IV is required for the first hash iteration. The output of the function h_i (with $1 \le i \le m$) – the intermediate hash value - is an l-bit block. The final hash-value H is obtained by performing an output transformation g on the last iteration result h_m. Typically, such a transformation consists of a truncation of h_m to generate a digest (H) with a predefined size n such as $n \le l$.

1-4.3.2. Example: SHA-1

SHA-1 is one the most widely used hash functions. For example it is employed in protocols such as SSL(Secure Sockets Layer), TLS(Transport Layer Security) and IPSec (Internet Protocol Security). It was designed by the NSA (National Security Agency) and standardized by the NIST [28].

SHA-1 takes as input a message M smaller than 2^{64} bits and processes 512-bit blocks. It provides a 160-bit digest. IVs and intermediate hash values are 160-bit long too. SHA-1 is divided into two parts: the pre-processing and the digest computation.

The pre-processing:

In SHA-1, the input message M is first padded if it is not a multiple of 512-bit and then split into m blocks b_i of such a size (with $1 \leq i \leq m$). The IV or h_0 – specified in [28] - initializes five 32-bit variables A, B, C, D, E (Figure 1-25) which are used to store intermediate hash value h_i (with $1 \leq i \leq m$). After this step the 512-bit blocks b_i are processed one at a time to obtain the digest.

Before describing such a process, several parameters and functions used by SHA-1 must be defined.

Parameters and functions:

W is an array of eighty 32-bit words and W_t is a word in W (with $0 \leq t \leq 79$).

K is a constant array of 32-bit words and K_t is a constant word in K (with $0 \leq t \leq 79$) such as:

$$K_t = \begin{cases} 0\text{x}5a827999, & \text{when } 0 \leq t \leq 19 \\ 0\text{x}6ed9eba1, & \text{when } 20 \leq t \leq 39 \\ 0\text{x}8f1bbcdc, & \text{when } 40 \leq t \leq 59 \\ 0\text{x}ca62c1d6, & \text{when } 60 \leq t \leq 79 \end{cases}$$

$RotL^n(x)$ is a circular shift rotation of the word x by n positions to the left.

$F_t(x, y, z)$[6] is a function such as:

$$F_t(x,y,z) = \begin{cases} (x \wedge y) \oplus (\neg x \wedge y), & \text{when } 0 \leq t \leq 19 \\ x \oplus y \oplus z, & \text{when } 20 \leq t \leq 39 \\ (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & \text{when } 40 \leq t \leq 59 \\ x \oplus y \oplus z, & \text{when } 60 \leq t \leq 79 \end{cases}$$

[6] $\wedge$ is the bitwise AND operation, $\oplus$ is the bitwise XOR operation and $\neg$ is the bitwise complement operation

Digest computation:

In order to compute the digest, the four following steps are applied to each 512-bit block b_i one at a time:

First W is set such as:

$$W_t = \begin{cases} b_{i,t} & \text{for } 0 \le t \le 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-16}) & \text{for } 16 \le t \le 79 \end{cases}$$

Where $b_{i,t}$ represents a t^{th} 32-bit word in b_i.

Then the five variables A, B, C, D and E are initialized by the five words composing the previous intermediate hash value h_{i-1} ($h_{i-1,0}$, $h_{i-1,1}$, $h_{i-1,2}$, $h_{i-1,3}$, $h_{i-1,4}$) except for the first block b_0 for which such a step is done during pre-processing (initialization with the $IV = h_0$).

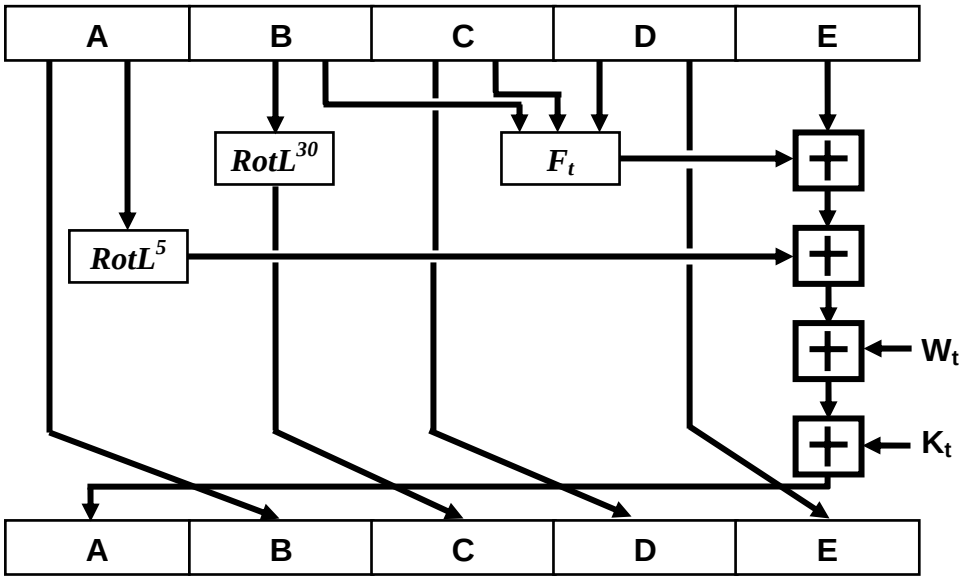


Figure 1-25 The compression function (*f*) of SHA-1

The third step is depicted in Figure 1-25. It consists in applying the following operations on the five variables[7]:

$$T = ROTL^5(A) + F_t(B,C,D) + E + K_t + W_t$$
$$E = D$$
$$D = C$$
$$C = ROTL^{30}(B)$$
$$B = A$$
$$A = T$$

[7] + used to compute the intermediate variable T is the addition modulo 2^{32}

Finally, the intermediate hash value h_i ($h_{i,0}$, $h_{i,1}$, $h_{i,2}$, $h_{i,3}$, $h_{i,4}$) is calculated with:

$$h_{i,0} = A + h_{i-1,0}$$
$$h_{i,1} = B + h_{i-1,1}$$
$$h_{i,2} = C + h_{i-1,2}$$
$$h_{i,3} = D + h_{i-1,3}$$
$$h_{i,4} = E + h_{i-1,4}$$

After repeating such a process for the m blocks composing the message M, the resulting hash is equal to h_m, meaning that the output transformation for SHA-1 is the identity.

1-4.3.3. Message Authentication Schemes Based on MDC

As previously defined, data integrity is a service which addresses the unauthorized alteration of data by an adversary during transmission or storage. Additionally to such a notion, it is required to check the origin of the data to be sure that a message really comes from the claimed sender. MDCs are not sufficient to provide such an assurance since they are publicly known as all cryptographic functions (2nd Kerckhoffs principle) and they do not enroll a secret. For example, suppose that Alice computes a tag T of a message M using an unkeyed hash function H - $H(M) = T$ - and sends it to Bob along with M. When Bob receives the pair (M; T) it computes the tag reference T' using $H - H(M) = T$'. If T is equal to T', the only certainty that he has is the fact that T is really the hash result of M using H! Indeed, a malicious third party could have intercepted the pair (M; T) during transmission (Man-in-the-middle: Figure 1.22) and replaced it by a chosen pair (M'; H(M')) without Bob being able to detect it.

The service which identifies the source of a message is called data origin authentication and it will be referred to as message authentication. Data origin authentication implicitly provides data integrity (Figure 1-30). Indeed, if a message is modified during transmission this means that the source has changed.

Mechanisms used in addition to MDCs to ensure message authentication are asymmetric signature, symmetric encryption or secret addition (HMAC [90]).

1-4.3.3.1. MDC and Asymmetric Signature

Asymmetric signatures, also called public-key decryption are based on the same cryptographic concept than the one used in public-key encryption. The difference comes from the fact that the encryption is done with the private key and the decryption with the public key instead of the reverse. In this way, the private key holder is the only one able to encrypt a message that everybody can decrypt. Here the encryption objective is not to provide confidentiality; such a principle combined with a MDC is used to provide message authentication.

Alice still wants to send a message to Bob with the assurance that it will not be modified during transmission and Bob needs to be sure that the information contained in the received message is correct and that Alice is really the sender (Figure 1-26). Thus Alice computes a hash (H) of the message M with a MDC and signs H by encrypting it with her private key K_e to obtain a tag T (also called signature). She sends the pair (M; T) to Bob over an unsecured channel. Bob receives (M; T), decrypts T using the Alice's public key K_d and hashes the message M with the same MDC than the one used by Alice. He then compares the resulting reference hash H' with H. The comparison of the two values informs Bob on the validity of M.

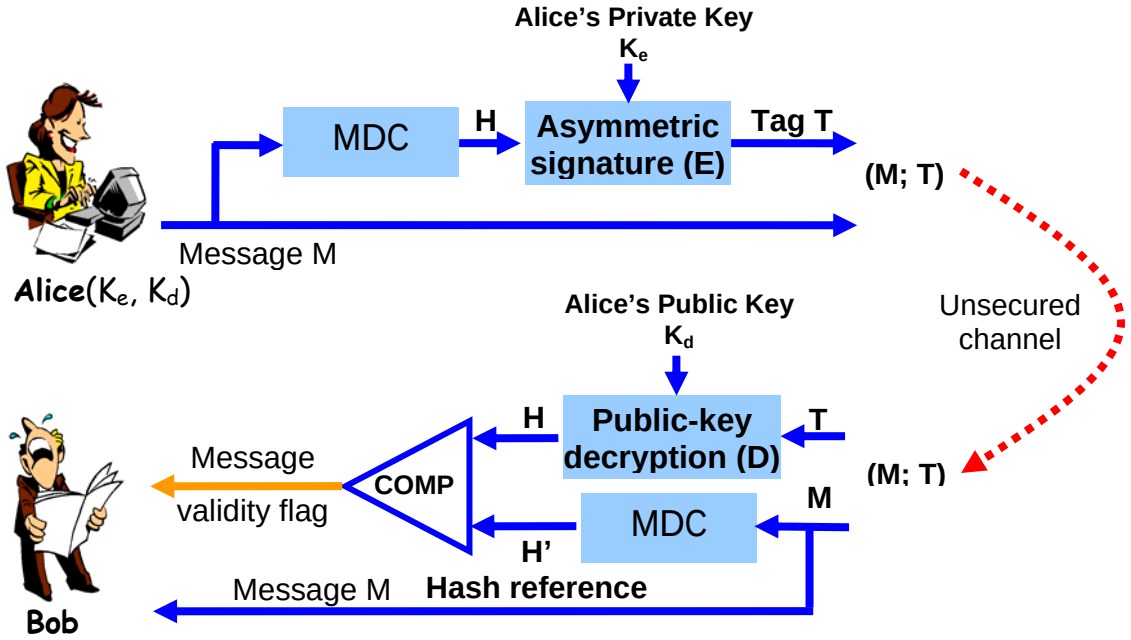


Figure 1-26 Message authentication using a MDC and asymmetric signature

MDC used in such a scheme must at least have the two properties: preimage resistance and second preimage resistance. Take up again the example of Alice and Bob. Everybody knows

the public key K_d and the tag T since it is transmitted over an unsecured channel. Therefore it is easy to recover H by decrypting T with K_d. Now, suppose that an adversary is able to find a second message M' resulting in the same H when processed by the same MDC than the one used by Alice. Thus, he can replace M by M' without Bob being able to detect it.

1-4.3.3.2. MDC and Symmetric Encryption

Another way to provide message authentication is to use symmetric encryption in addition to MDC. This time Alice and Bob must agree on a secret encryption key K_s (Figure 1-27). Then Alice computes a digest H of the message M with a MDC and encrypts M and H with a symmetric algorithm and the secret key K_s. Bob receives and decrypts the ciphered message to recover M and H. He uses the same MDC than the one used by Alice to compute a hash H' of M. If H does not match the hash reference H' this means that the message was tampered with during transmission.

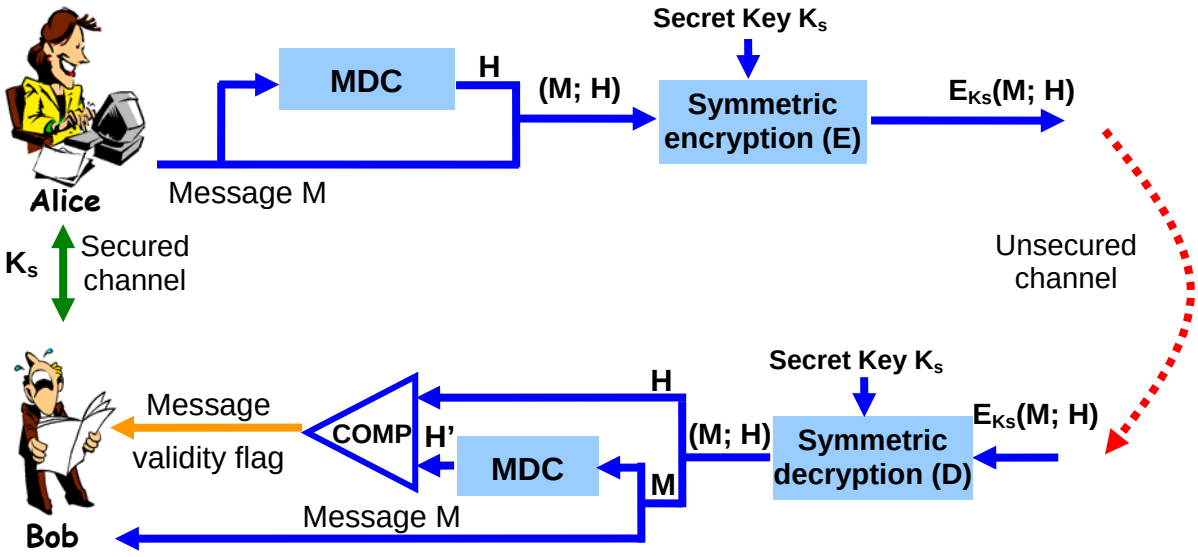


Figure 1-27 Message authentication using a MDC and symmetric encryption

In this case, if the underlying symmetric algorithm employed is secure, it is not required that the MDC should have the three properties: preimage, second preimage, and collision resistance. Indeed, an adversary who has neither the input of the MDC nor its output can not mount a forgery attack.

On the other hand, if confidentiality is not required, only the hash value may be encrypted but the MDC must at least be collision resistant. Indeed, suppose that it is not the case and that an adversary who monitors the communication channel (Man-in-the-middle: Figure 1-22)

retrieves the plaintext and the encrypted hash. Hence, he is able to first recover the digest by applying to the plaintext the same MDC than the one used by the sender and then to find a second message which gives the same digest. Since symmetric encryption is deterministic – meaning there is only one ciphertext for a given plaintext – he could send his message with the encrypted hash without the possibility for the intended recipient to be able to detect it.

1-4.4. Keyed Hash Functions and MAC Algorithms

1-4.4.1. Principle

Keyed hash functions are dedicated to message authentication and thus are also called Message Authentication Code (MAC) algorithms. In the following the hash generated by such algorithms is referred to as the MAC or the MAC-value. They take a secret key as input in addition to the message to hash. They have a further property which is computation-resistance, i.e. they are indistinguishable from random. This means that given an unspecified number of message-MAC pair (m_i, MAC (m_i)), it is impossible for an attacker to compute any message-MAC pairs (m, MAC (m)) for any new input $m \neq m_i$. In other words, the output of the MAC algorithm must seem random from an adversary point of view and this is mainly achieved by using the secret key. A hash function with such a property is necessarily preimage resistant, second preimage resistant and collision resistant since it is required to hold the secret key to be able to compute a MAC-value.

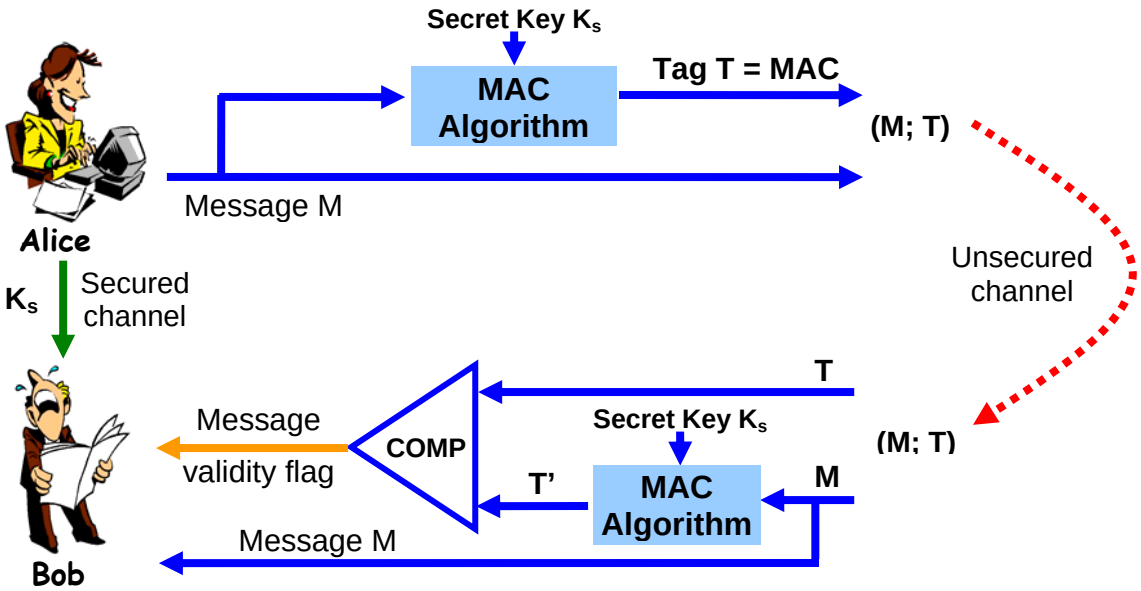


Figure 1-28 Message authentication using MAC algorithms

In order to send an authenticated message M to Bob, Alice must first securely communicate to him a secret key K_s (Figure 1-28). Then Alice computes a tag T by applying a MAC algorithm on the message M and by using the secret key. Finally, she transmits T along with M to Bob. After reception, Bob computes a MAC-value T' (tag reference) from M using K_s. If T' matches T, the message is authenticated.

Note that the passive attacks described in section 1-3.3 also apply to MAC algorithms. In this case the objective of the adversary is either to forge a message or the tag.

MAC algorithms may be designed by using the general iterative model of hash function depicted in Figure 1-24. The secret key is thus an input of the output function g or it is involved in all iterations of the compression function f.

MAC algorithms of our main interest are based on block ciphers and the most widely used one is the CBC-MAC.

1-4.4.2. Example: CBC-MAC

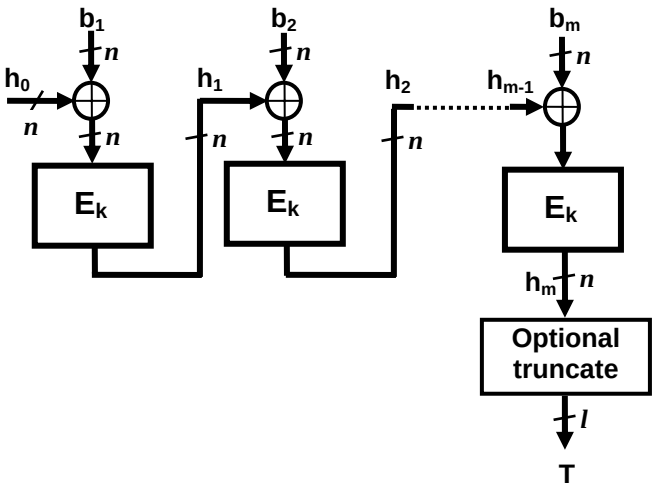


Figure 1-29 General model for CBC-MAC

The principle of CBC-MAC is presented in the following in the case where the length of the message to authenticate is a multiple of n, the block size of the underlying block cipher. For detailed description of the different standardized CBC-MAC refer to [24].

The general model for CBC-MAC is depicted in Figure 1-29. The MAC is computed by chaining block encryption, therefore the message M to authenticate is first split into m n-bit blocks b_i with (with $1 \leq i \leq m$). An intermediate MAC h_i is obtained by the encryption of the current processed block b_i Xored with the previous intermediate MAC h_{i-1} - $h_i = E_k(b_i \oplus h_{i-1})$ - except for the first block b_1 which is directly encrypted. The final MAC (or tag) is produced by truncating h_m: the l leftmost bits are selected such as $l \leq m$.

If $l = n$ a simple forgery attack may be mounted. Consider a message M_1 composed of m blocks (b_1, b_2, … , b_m) and its corresponding MAC T produced by the previous exposed CBC-MAC scheme. In the light of the XOR operator, the following message M_2 has the same MAC T:

$$\mathrm{B}_2 = b_1, b_2, ..., b_m, T \oplus b_1, b_2, ..., b_m$$

Indeed, $h_{m+1} = E_k(T \oplus b_1 \oplus T) = E_k(b_1) = h_1$ and thus the same MAC computation is resumed. However, such an attack is not applicable in the case where all the messages to be authenticated have the same length m. The message forgery would be rejected at the input because it does not have the requested length.

1-4.5. The Birthday Attacks

The birthday attack is an algorithm-independent attack i.e. such an attack considers the algorithm as a block-box. It is an application of the birthday paradox [16]. Practically, suppose a function h which randomly generates output values from a set of N elements. There is a high probability that the same element is encountered twice after approximately $\sqrt{N}$ computations. This means that for a hash function producing an m-bit digest, a collision is found after roughly $2^{m/2}$ hash value generation. For example, SHA-1 is said to offer 80-bit of security against such an attack since it produces 160-bit digest.

Resistance against the birthday attack is a security criterion to evaluate robustness of hash functions.

All previous techniques (MDC and asymmetric signature, MDC and symmetric cipher and MAC) provide message authentication. However such a notion does not include preventions against replay attacks. Such attacks are performed in two steps. First, at a time t on an unsecured channel an adversary retrieves a message M - possibly encrypted – and the corresponding tag T. Then, at a time $t + n$ of its choice, he could send the pair (M; T) again or use it in a Man-in-the-middle attack. Thus, the intended recipient will not be able to detect that this message M is a fake since after computation of the reference tag T' on M, T will match T'. The service requiring prevention against replay attacks is referred to as transaction authentication.

1-4.6. Transaction Authentication

Transaction authentication denotes message authentication augmented to additionally provide uniqueness and timeliness guarantees on data (Figure 1-30), thus preventing undetectable message replay (definition taken from [16]).

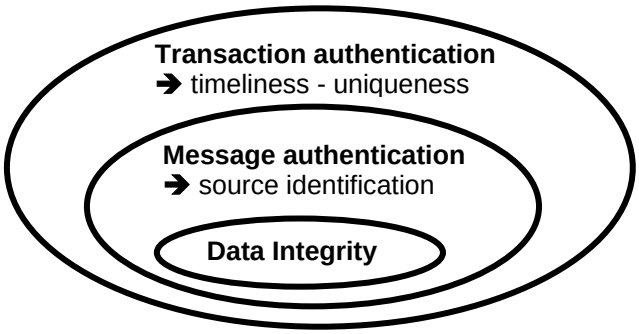


Figure 1-30 Taxonomy of security services required to prevent active attacks

Uniqueness and timeliness are ensured by using time-variant parameters as inputs with the data in message authentication schemes. Random numbers, timestamps and nonce are examples of time-variant parameters. A nonce is a number used once for a given purpose. It could be generated by a counter. Take up again the example of Alice and Bob depicted in Figure 1-28. In order to thwart replay attacks, they choose to use a nonce N. Hence, before starting to communicate, they must agree on a secret key K_s and on a random number to initialize N. Then each time Alice sends a message M to Bob, she increments the nonce and computes a tag T with a MAC algorithm which takes as inputs K_s and M concatenated with N. Finally, she transmits T along with M to Bob. After reception, Bob updates the nonce and computes a reference MAC-value T' from the secret key K_s and from M concatenated with N. If T' matches the one appended to M by Alice, the transaction is authenticated.

1-5. Conclusion

In this chapter, we defined the cryptographic terms useful in this thesis. Moreover we presented the encryption techniques guaranteeing data confidentiality and we described mechanisms allowing to check data integrity (and data origin authentication). However, we do not intend to give an exhaustive survey, but an overview of the existing solutions and a description of schemes used in the following chapters.

Chapter 2: Security Concerns

The objectives of attacks led on embedded systems are to retrieve information – possibly private – or to take control of it. One of the weakest points of such systems is the bus between the System on Chip (SoC) and the off-chip memory which contains sensitive data (end users private data, software code…). Those data are usually exchanged in clear over the bus during software loading and execution. Therefore an adversary may probe this bus to read and to retrieve private data or software code (data confidentiality concern). Another possible attack relies on code injection and on data tampering (data integrity concern).

Thus, our goal is to provide a private and authenticated tamper resistant environment to software execution. This means that an adversary must be unable to understand the data retrieved on the bus or directly in memory, and that we must detect all kinds of data and code tampering (tamper detection).

This chapter is organized as follows. Section 2-1 shows the relevancy of protecting data transferred on a processor-memory bus through an application example: software copy protection. Section 2-2 presents our threat model by specifying the adversaries and the attacks taken into account. This threat model is illustrated with an example of attack. Section 2-3 highlights the specific issues of the implementation of cryptographic functions in a computing device. Thus, we first give an overview of the memory hierarchy in a computing system, then we present the basic principles of bus encryption and of memory integrity verification and finally, we describe the operations which are a source of overhead when such mechanisms are implemented.

2-1. Software Copy Protection

The emergence of on-demand software downloading services brings in the foreground the problem of intellectual property (IP) protection. Software distributed over unsecured channel might be retrieved by a malicious eavesdropper and illegally spread. Therefore, the IP confidentiality must be ensured all along the distribution chain and also during execution. Such a goal is achieved by end-to-end encryption: from the source - the software provider - to the host - the computing system - and during execution.

As an example, in this section we consider a set-top box operator who proposes on-demand services or who needs to update the customer's unit. Those services or products must be used only by the intended user and this user must not be able to copy it and to give it away. The scheme depicted in Figure 2-1 is used to ensure confidential distribution of the intellectual property sold by the set-top box operator:

(1) A set-top box processor is equipped by the chip manufacturer with a private key D_c of a public/private key pair E_c/D_c and with a certificate C_e for E_c. C_e particularly contains the processor public key E_c and its signature produced by the chip manufacturer with his private key D_m.

(2) The chip manufacturer public key E_m is sent via an authenticated channel (PKI: Public-Key Infrastructure [24]) or delivered with the set-top box units to the set-top box operator.

(3) When the customer asks for a service or a new feature (e.g. an update of the media player) the certificate C_e is sent with the request.

(4) The set-top box operator authenticates E_c – asymmetric decryption of the signature contained in C_e using E_m (Chapter 1 – Figure 1.25).

(5) The software code S provided by the operator is symmetrically encrypted with the secret key *K*. *K* is then asymmetrically ciphered with the public key E_c before being sent to the customer's set-top box.

(6) The customer retrieves *K* by asymmetric decryption with its private key D_c. Then *K* serves to symmetrically decrypt S.

(7) S is installed in the memory[8].

[8] Note that the user privacy is not ensured with this scheme since the operator could collect information on the user activities or consummation habits. Zhang and al. [29] proposes a simple scheme also based on a PKI infrastructure to solve this issue.

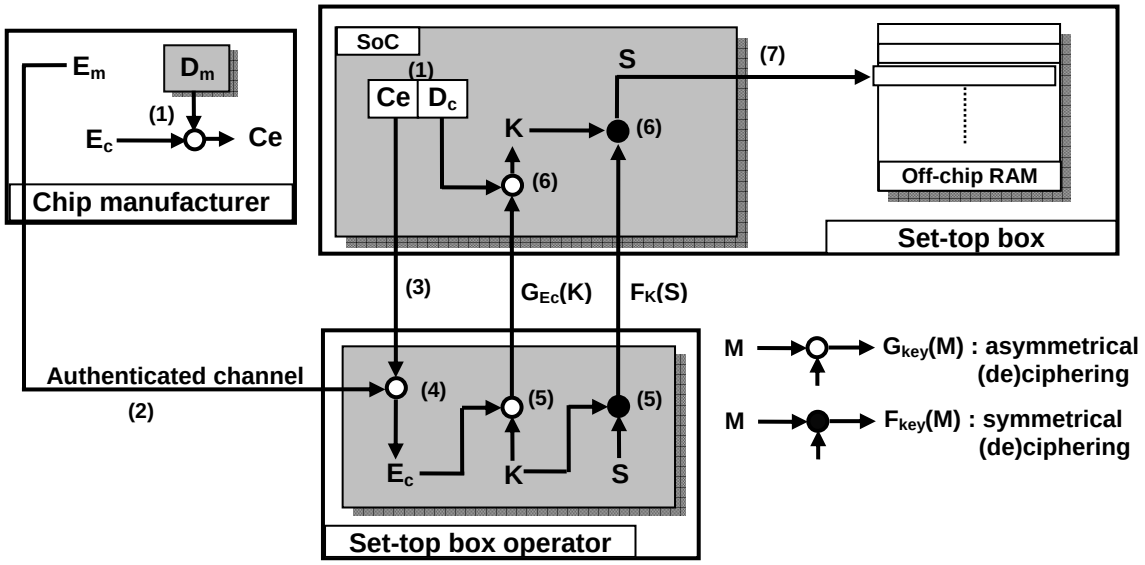


Figure 2-1 Secure software download for software copy protection

Thus, if S is recorded on the communication channel between the set-top box operator and the customer's unit it cannot be illegally distributed: each set-top box is equipped with a different private key D_c therefore the step (6) of the previous scheme would fail.

S could be installed already encrypted in the external memory to extend the protection to the processor-memory communication. However, the hardware engine implemented to decrypt S must also take into account the encryption of RW data processed at run-time. Therefore, this engine must be optimized by considering the contextual specific constraint and propose an affordable trade-off between security strength and computation performance. Moreover, as highlighted by the attack described in section 2-2.3, the integrity of the memory content must also be verified to avoid code or data corruption which could lead to sensitive information leakage.

2-2. The Threat Model

In this section, we first classify adversaries by evaluating their ability to conduct attacks to define the security level targeted in this thesis. Then we describe the attacks considered. Finally, an example of attack is detailed to illustrate our threat model.

2-2.1. Security Level and Adversaries Classification

IBM proposed a taxonomy [30] of adversaries and attacks in order to classify the security level achieved by each of their product: "Adversaries were grouped into three classes, in ascending order, depending on their expected abilities and attack strengths:

Class I (clever outsiders): They are often very intelligent but may have insufficient knowledge of the system. They may have access to only moderately sophisticated equipment. They often try to take advantage of an existing weakness in the system, rather than try to create one.

Class II (knowledgeable insiders): They have substantial specialized technical education and experience. They have varying degrees of understanding of parts of the system but potential access to most of it. They often have access to highly sophisticated tools and instruments for analysis.

Class III (funded organizations): they are able to assemble teams of specialists with related and complementary skills backed by great funding resources. They are capable of in-depth analysis of the system, designing sophisticated attacks, and using the most sophisticated analysis tools (very expensive). They may use Class II adversaries as part of the attack team."

In this work, we consider adversaries for whom the cost of the attack should not exceed the price of the protected entity or the expected amount of profits. Thus, in the following, our study will focus on countermeasures to thwart attacks and adversaries classified in class II.

2-2.2. Considered Attacks

The device to protect is supposedly exposed to a hostile environment where physical but non-invasive attacks are feasible (typically attacks carried out by adversaries of Class II).

As a consequence, the main hypothesis in our threat model is the fact that the System on Chip (SoC) is considered as tamper resistant to physical attacks. A proof of such an assumption is the Xbox cracking example. In [6] the hacker, Andrew "bunnie" Huang, exposes his approach to attack the gaming console. The targeted ASIC was manufactured in a 0,13µ process with several metal layers and to shape the chip would have required a lot of facilities. Thus, he considered this choice as too expensive and time-consuming and he decided to probe buses, which seemed a more reasonable approach. 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.

Moreover, software attacks are not considered; the operating system (OS) or at least a part of the OS, called the secure kernel, is also trusted.

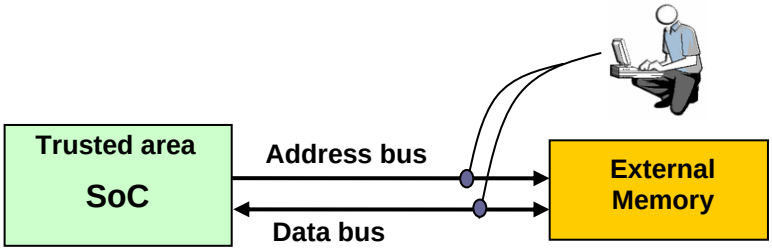


Figure 2-2 Board level attack based on bus probing and on data injection

In this work, we focus mainly on board level attacks involving processor-memory bus probing (Figure 2-2) or memory tampering. Such attacks allow to perform readings of the memory content or to inject data on the bus or directly into memory. We are particularly concerned by the "Man in the middle" attacks. The corresponding protocol implementing this attack is divided into two parts:

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

➢ Then the adversary may insert chosen texts – called "fake" in the following - on the processor-memory bus and thus challenge data integrity. The objective of the attacker could be to take control of the system by injecting malicious codes or to constrain the search space in case of a message recovery attack. The choice of the fake by the attacker leads us to define three kinds of feasible active attacks even if data are encrypted:

- *Spoofing attacks*: the adversary exchanges a memory block transmitted on the bus with a random fake one. The attacker mainly alters program behavior but cannot foresee the results of his attack if the data are encrypted.

- *Splicing or relocation attacks:* the attacker swaps a memory block transmitted on the bus with another one previously recorded in the external memory. 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*: the protocol is nearly the same as for the splicing attack one; however the fake memory block is recorded at a specific address location and inserted later on at the same address (current data value replaced by an older one). Such an attack may be viewed as a *temporal permutation* of memory blocks at a specific address location.

In order to perform those kinds of active attacks, the adversary can interfere in the protocol of communication between the SoC and the memory to handle the data, address and control lines. In this way he can insert data directly into RAM memory or switch between his own RAM and the device RAM at run-time (as implemented in the attack described in section 2-2.3).

We must provide message authentication to thwart spoofing attacks while for splicing and replay we must ensure transaction authentication (section 1-4.6).

For clarity, in the following, the terms integrity checking and authentication, integrity and authenticity are used interchangeably. Moreover we refer to the described attacks – spoofing, splicing and replay – rather than to the security services – message and transaction authentication - to define the goal of the designed countermeasures.

The next sub-section describes an attack on a commercial device to illustrate how a man in the middle attack led on the processor-memory bus can challenge the confidentiality of the off-chip memory content.

2-2.3. Attack conducted on a Commercial Device: The DS5002FP

Markus G. Kuhn performed a man in the middle attack on the Microcontroller DS5002FP [31] in 1997. Such a microcontroller was the most widely used commercial bus-encryption

processor at that time. This attack allowed the student to access the ciphered data stored by the DS5002FP in the external memory and to retrieve their clear version.

The confidentiality of the data stored off-chip by the DS5002FP relies on an 8-bit block encryption / decryption function. The address bus is also encrypted with a 17-bit block encryption function. The secret key used is a 64-bit key randomly generated and stored on-chip. The software is uploaded in clear through the serial port, then encrypted and stored in the external SRAM. The data encryption function depends on the secret key and on the data address.

The principle of this attack is to inject guessed ciphered instructions on the processor-memory bus and to figure out their corresponding clear version by observing CPU reactions. Such instructions are used to construct small encrypted programs which help the attacker to learn more on the system. Finally, a program designed from all the information deduced allows to output the whole memory content in clear form on the parallel port.

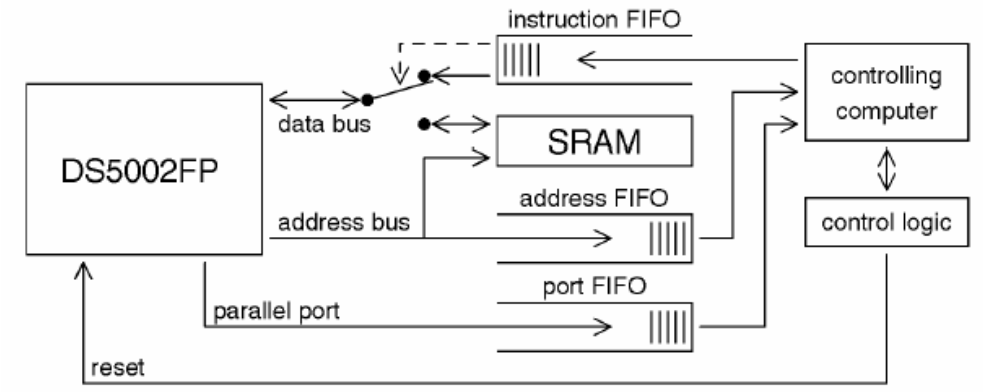


Figure 2-3 Principle of the DS5002FP CPU attack with a read-out device connected to the bus (taken from [31])

In order to perform the attack a computer and a read-out device (developed by Markus G. Kuhn [31]) are used (Figure 2-3). The read-out device is connected to most of the pins of the DS5002FP microcontroller and to the SRAM memory. It allows to reset the CPU and to record in FIFO memory the CPU (Central Processing Unit) reaction on the address bus and on the four 8-bit parallel ports. Moreover, such a device is used to swap 8-bit encrypted data on the bus: instructions fetched by the CPU are replaced by the guessed ciphered instructions stored in a FIFO memory. Chip-enable and read/write signals are routed through the control logic of the read-out device to enable switching between SRAM and the instruction FIFO.

Therefore those signals are linked either to the SRAM or to the instruction FIFO depending on whether the attacker wishes to inject chosen data or not.

The cost of this attack is low, it has been evaluated at 300$. Moreover, it can be done in only four hours. With this attack Kuhn shows practically that implementing short block encryption drastically limits the search space of a message recovery attack. In the DS5002FP the 8-bit encryption function (address-dependent) induces a mapping plaintext/ciphertext with only 256 entries at each memory location. Moreover, this attack highlights the fact that providing encryption only might not be sufficient to guarantee data confidentiality. M.G. Kuhn did not attack the encryption scheme directly; he constructed his own encrypted program gradually by inserting guessed instruction on the bus to finally dump the memory content through the parallel port after it had been decrypted by the encryption / decryption unit itself. A strong data integrity checking mechanism could have prevented such an attack.

2-3. System on Chip Context

In order to clearly highlight the challenge of protecting data transferred between a SoC and its memory and the consequence, in term of performance, of the choice of the underlying cryptographic functions, it is important to understand how those data are processed and accessed. In this section we do not describe data processing in computer architecture [32, 33] but we present the computing characteristics which influence the design of the hardware engines for security and their computation performance. Then we give the basic implementation principles adopted for them. Finally we present the source of performance degradation generated by those engines.

2-3.1. Memory Accesses

The off-chip memory is the main memory which contains the applications and data executed and processed by the CPU. However, the processor also uses an on-chip memory, called cache memory, closer, smaller and faster than the main memory. Its goal is to reduce the average time of memory accesses and, as a result to enhance global computing performance. Such a goal is reached by exploiting the *locality* principles:

- *Temporal locality*: If data is requested by the processor, there is a great probability that it will be used again shortly after.
- *Spatial locality*: If data is requested by the processor, there is a great probability that the data stored around in adjacent addresses are also requested shortly afterwards.

Thus, the cache memory is divided into blocks (or cache lines) and when an instruction or data is loaded from the main memory, it is simultaneously copied into the cache memory (temporal locality) with the surrounded data belonging to the same cache block (spatial locality). The architect-designer chooses the cache line length carefully to optimize computation performance by considering the underlying processor architecture and the applications running on it. Since the cache memory is smaller than the off-chip memory a replacement policy must also be defined (refer to [32] for more details).

In this work, Load / Store RISC (Reduced Instruction Set Computer) architecture is used as processor, meaning that on a read cache miss – the data fetched by the processor is not present in the cache memory - the CPU loads a payload of a cache line size. However, such a behavior is not relevant for all read operations and loading certain data in the cache may be very inefficient, for instance when those data are known to be needed briefly. Another case is for data belonging to large structures like arrays. Such structures may be read entirely in the cache and thus result in cache pollution and in eviction of useful data from the cache. In order to solve this issue, those data are tagged as non-cacheable i.e. they are never stored in the cache memory and are directly fetched from the off-chip memory.

On write operations, two different write policies may be implemented to achieve coherency between the cache and the off-chip memory: write-back and write-through. On write-through cache policy, the modifications of a data in the cache are instantaneously copied into the off-chip memory whereas on write-back a cache line is tagged in the cache when it is changed and the external memory is updated only when the block is evicted from cache. The latter write policy avoids successive off-chip write operations. However on each read cache miss, it requires to chain one write operation - to update the external memory with the evicted cache block if tagged as modified - with one read operation - to answer to the cache miss request. Moreover, for both cache policies, when the data to write is not in the cache, it is directly written in the off-chip memory[9].

[9] Such a behavior is called write-no-allocate. The policy which consists in loading the matching cache block before performing the write is named write-allocate. However we do not consider this latter case in this thesis since the processor used in the following (ARM9E [89]) does not implement it.

2-3.2. Basic Principles for the Hardware Mechanisms for Data Security

2-3.2.1. Hardware Mechanisms for Data Security Localization

Hardware mechanisms for security are designed between the cache memory and the memory controller on the trusted area (Figure 2.4).

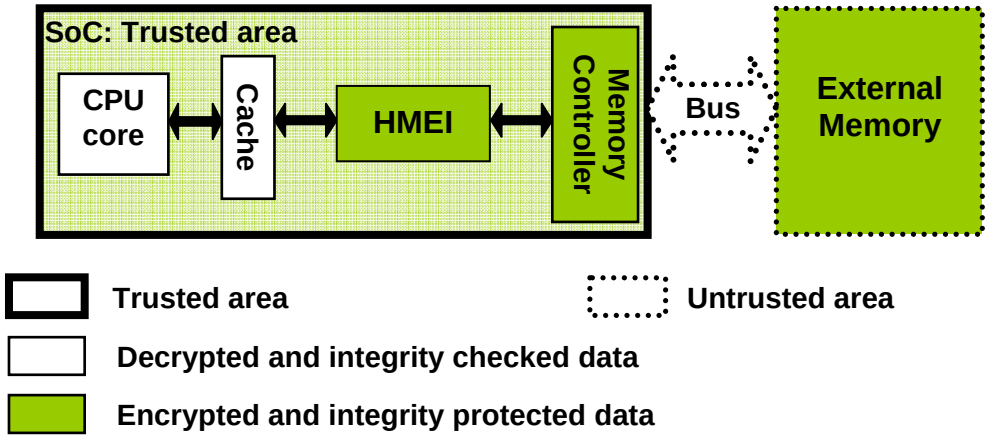


Figure 2-4 Localization on the SoC of the hardware mechanisms providing data confidentiality and authentication services

This choice is motivated by the following reasons:

1. *Performance*: such localization allows to store deciphered and integrity checked data in cache memory. Hence, only off-chip memory accesses are impacted by the additional latencies introduced by the underlying cryptographic functions.
2. *Security*: Secret values, like encryption keys, enrolled in the cryptographic computations are stored on the trusted area and thus, are considered as inaccessible and secret from adversaries' point of view.
3. *Compatibility*: the design of the security engines is fully independent of the type of the memory to protect.

2-3.2.2. Bus Encryption Principle

Guaranteeing confidentiality of external memory content consists in preventing any useful information leakages from those off-chip memories. Hence, the basic requirement is that data monitored by an eavesdropper on the processor-memory bus or retrieved in memory must be

unintelligible. This task is achieved by performing bus encryption. The principle –introduced by Best [8, 9, 10] is obvious; data are encrypted on write operations and decrypted on read operations. In this way data transiting on the bus and stored in the off-chip memory are encrypted, making them incomprehensible from an adversary point of view. However, targeted memories are Random Access Memory (RAM), meaning that memory accesses could be of any length and start from any address; hence we must define a granularity of encryption i.e. the size of the atomic block – called in the following *chunk* – processed by the encryption engine on external memory accesses. Such a size is one of the parameters which fix the trade-off between performance and security. Chunks that are too short lead to weak encryption as highlighted by the Markus Kuhn attack – where a chunk of data is an 8-bit block – while too long ones may decrease computation performance e.g. by polluting the memory bandwidth on small memory accesses (see section 2-3.3.2). Several cryptographic choices for the encryption engine influence the definition of the chunk size:

- The ciphered block size of the implemented underlying block cipher – to be used as the output algorithm in a stream cipher or directly for block encryption – sets its minimum length.
- The encryption mode; for instance when CBC mode is used, a chunk is of a CBC chain length.

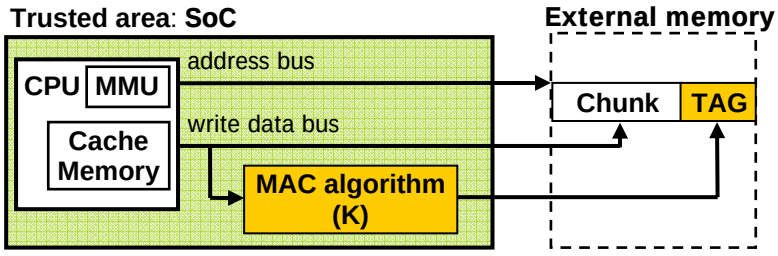
As memories are organized in cache blocks, in the literature a chunk is usually of this size. However, as highlighted in section 2-3.3.2, this is not necessarily the most efficient choice.

2-3.2.3. Principle of Memory (Content) Integrity Verification

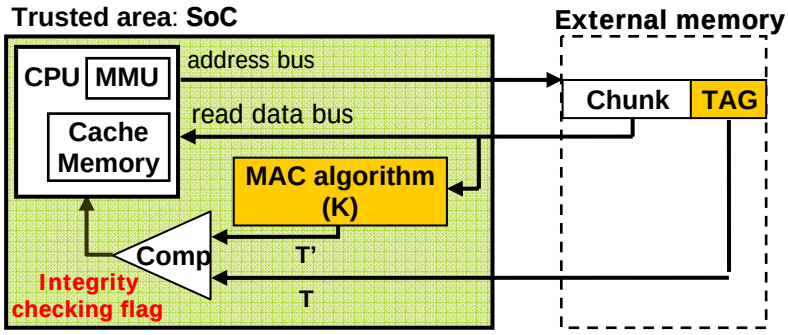
The integrity of the memory content is ensured by checking that read data has not been tampered with during external storage or transmission over the bus. Like for encryption, a granularity of integrity checking must be defined. In this section the chunk is the atomic block loaded on-chip to be authenticated on read operations.

To fulfill the integrity checking objective, a value is appended to each chunk stored in the external memory. This value called tag (*T*) is usually computed on-chip with a MAC (Message Authentication Code) algorithm on *write operations* (Figure 2-5a). Such algorithms based on hash functions or on symmetric block encryption accept as inputs the chunk and a secret key. Theoretically they create a unique chunk/tag pair and the generated tag must give a

compact representative image of the chunk and of its source i.e. the processor. Moreover, only the SoC is capable to compute this tag as the secret key is stored on-chip. On *read operations* (Figure 2-5b), the integrity of the loaded chunk is checked by verifying the tag. To achieve this task, a tag reference *T'* is computed on the chunk and compared with the tag *T* retrieved from the off-chip memory. If the tag matching process fails, an integrity checking flag informs the CPU which in turn adopts an adequate behavior (for instance a HALT instruction to stop processor execution).



(a) Write operation: Tag computation



(b) Read operation: Tag verification

Figure 2-5 Integrity checking principle of external memory

However, an efficient memory integrity checking engine must also verify that the memory works as a valid memory, meaning that a value read by the processor at a given address is the most recent value stored at this address (replay attacks). As explained in Chapter 3 (section 3-2), the general principle presented above must be tuned to be compliant with the previous requirement.

Note that for integrity checking the minimum chunk size is defined by the length of the atomic block processed by the underlying MAC or hash functions. Nevertheless, when data confidentiality and integrity is ensured the size of the chunk must be decided by also considering the constraint imposed by the encryption scheme.

2-3.3. Run-Time Performance Degradation Considerations

2-3.3.1. Data Properties

During software execution, two kinds of data are processed: Read Only (RO) data and Read Write (RW) data.

Instructions or RO data are only written once in the off-chip memory at loading (from a Non Volatile Memory - NVM - or from a network connection). As a consequence, latencies introduced by the decryption and the integrity checking processes must be optimized on read operations.

RW data are more difficult to secure because they are dynamically modified during software execution: several data values can be stored at the same address in the off-chip memory. Therefore RW data - contrarily to RO data - are vulnerable to replay attacks.

2-3.3.2. Sources of Time Performance Degradation

The run-time performance slowdown induced by the implementation of cryptographic algorithms, has mainly two sources:

- the latencies introduced by the underlying functions involved in encryption and in tag computation,
- the memory bandwidth pollution generated by the loading of meta-data such as tags.

However, the processing of data by the security engines generates specific operations which represent another source of degradation. Whatever the size of the chunk and by considering that data integrity and confidentiality are ensured, a performance overhead must be expected on:

- *Read operations of data smaller than a chunk*: such operations occur mainly for non-cacheable data[10]. It is required to i) load the whole matching chunk from external memory with its tag ii) decipher it and check its integrity iii) and 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.

[10] And only for non-cacheable data when the chunk is smaller or equal in size to the cache block length.

- *Write operations of data smaller than a chunk* require to perform the following steps: i) load the matching chunk with its tag from the off-chip memory ii) decipher it and check its integrity iii) modify the corresponding sequence in the chunk iv) re-cipher it and re-compute its tag v) write it back into memory with its new tag. This chain of operations is called in the rest of the thesis a *Read Modify Write (RMW).* A RMW operation is always required in the following cases:
  (1) when the write-through policy is implemented
  (2) for non-cacheable data
  (3) on write miss – the data to modify is not in the cache - with the write-back cache policy.

  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 operations, ideally the chunk size should be defined as small as possible without affecting security.

Note that those operations only concern RW data. This is one of the reasons – with the specific countermeasure required against replay (Chapter 3 and 6) - why the overhead of the hardware mechanisms designed for data security mainly emanates from the RW data processing.

2-4. Conclusion

In this chapter we specified the targeted security level we want to achieve by defining the kind of adversary we take into account. Moreover we presented our threat model where all components (buses and off-chip memories) are considered as untrustworthy except the SoC. This threat model is illustrated with an example of attack which shows that guaranteeing data confidentiality often requires also checking their integrity. Then, we described the principle of bus encryption and of memory integrity verification. Finally, we introduced the notion of chunk (atomic block loaded on read operations to be decrypted or/and integrity checked) essential to fix the trade-off between security and run-time performance.

Chapter 3: Related Works

This chapter surveys the hardware cryptographic engines dedicated to protect the off-chip memories of computing systems. It is organized as follows. Section 3-1 presents the existing bus encryption engines. In section 3-2, the mechanisms for memory integrity verification proposed in the literature are discussed with a specific emphasis on the schemes handling the replay attack issue. Finally, section 3-3 describes the technique providing both data confidentiality and authentication and the related schemes implemented in SoC.

3-1. Hardware Engine for Bus Encryption

The principle of bus encryption was first introduced by Best in 1979 [8, 9, 10]. Best fixed the basic principles presented in Chapter 2 (section 2-3). Accordingly, a cipher unit is implemented on-chip with a secret cipher key located in an on-chip register. The block cipher chosen by Best is based on basic cryptographic functions such as mono and poly-alphabetic substitutions and byte transpositions.

Today's hardware engines for bus encryption still use Best's principles but propose improved security by implementing standardized encryption algorithms. Such engines use symmetric rather than asymmetric cryptography for performance reasons; asymmetric encryption or decryption on each external memory access would add a prohibitive latency. Bus encryption engines are divided into two families. The first one is called *direct encryption*

since the encryption and the decryption processes start only after receiving data; they use block cipher in ECB or CBC modes. The second family, One Time Pad (OTP) engines, is based on the CTR mode of encryption which enables to parallelize the keystream generation with memory access latencies.

3-1.1. Direct Encryption

Mainly two academic works proposed a detailed description of engines based on direct encryption: Gilmont and al. [34, 35, 36] and a first version of the AEGIS (Architectural EnGine for Information Security) processor [12, 13, 37, 38] developed at the MIT (Massachussetts Institute of Technology) by Suh and al.

Gilmont and al. implements a direct encryption/decryption scheme based on the DES algorithm. The software code which is installed has already been ciphered. The encrypted application and the secret key *K* are transmitted by the software provider using an asymmetric mechanism as the one exposed in Chapter 2 (section 2-1). The encryption mode implemented seems to be ECB since a pipelined DES is used. The chunk size is of 64 bits and the encryption is salted with the virtual address. Therefore for RO data, this ensures that a same plaintext encrypted twice yields two different ciphertexts. However it is not true for RW data: a same value encrypted twice and stored at the same address results in the same ciphertext. This could lead to information leakage: for instance an adversary can deduce when a loop counter returns to a given value. Concerning performance cost, Gilmont and al. evaluates the overhead implied by decryption on an ARM7 processor core to 1% in the best case and 12% in the worst case. However it is not clear in [36] if they consider the encryption cost.

A first version of the AEGIS [12, 37] processor implements AES direct encryption and uses two keys per application, K_{static} for RO data and $K_{dynamic}$ for RW data. The program received is already encrypted under K_{static}. $K_{dynamic}$ is generated on-chip with a random number generator [12, 37]. The granularity of encryption (chunk) is aligned on a L2 cache block basis (512-bit). The cache block is broken into 128-bit sub-blocks B[1], B[2], B[3] and B[4] and encrypted in CBC mode as depicted in Figure 3-1 (taken from [37] - || is the concatenation operator). The initialization vector *IV* required for the CBC mode consists of the address chunk and of a 32-bit value *RV*; the rest of the *IV* is padded with zeroes to be 128-

bit. For RO data *RV* is set to zero and for RW data it is randomly generated on each write operation to avoid the ciphering of the same plaintext twice leading to the same ciphertext. *RV* is stored in the off-chip memory. On a read cache miss, *RV* and the chunk are loaded, if *RV* is zero, the cryptographic engine uses K_{static} for decryption otherwise it uses $K_{dynamic}$. Such a decryption (CBC) can be done in parallel.

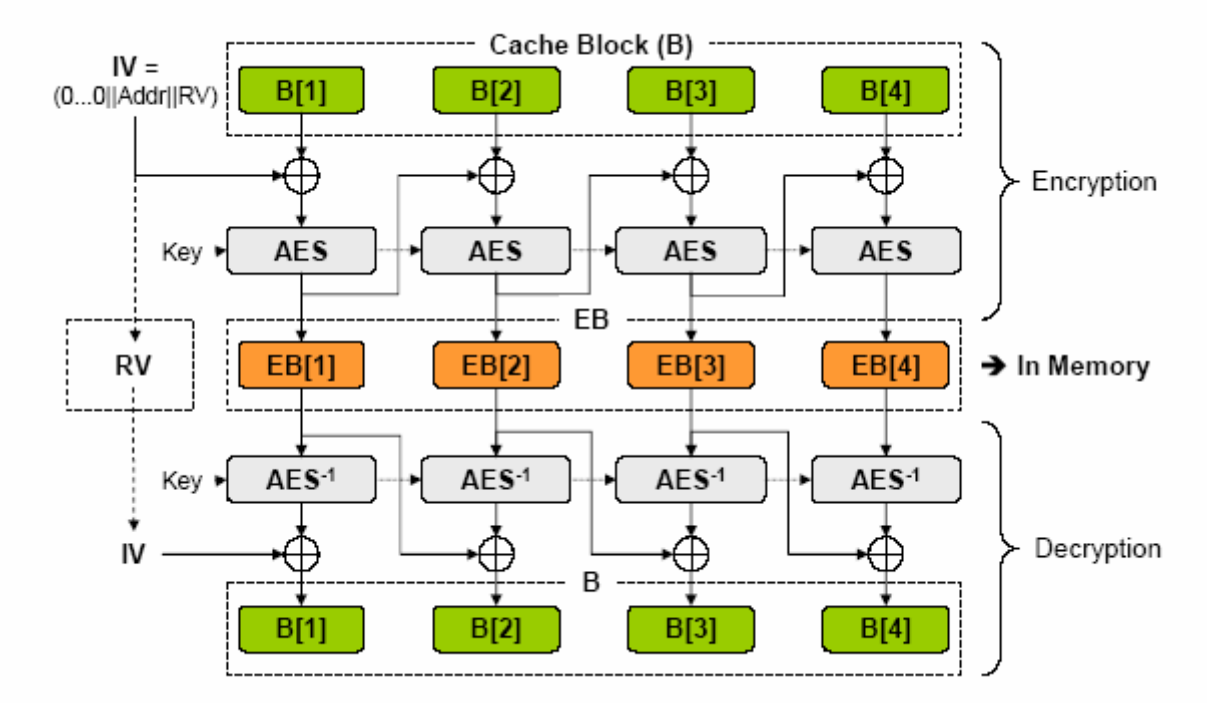


Figure 3-1 Direct encryption scheme (AES-CBC) proposed in the first version of the AEGIS processor

Suh and al. give a performance evaluation of the proposed scheme on a high-end processor (1GHz) with two levels of cache (L1 I-cache = 64KB, L1 D-cache = 64KB, unified L2 caches = 1MB), a 64-bit bus (200 MHz) and considering 80 cycles of memory access latency and 40 cycles for the AES latency. The performance slowdown reaches 25% when the whole external memory is encrypted and the memory overhead implied by *RV* storage is of 6.25%. In order to allow parallel decryption four AES cores are required [12].

This encryption engine is secure regarding data confidentiality. However, an adversary has access to *RV*; hence the only information leakage comes from the possibility for an attacker to know when a cache block has the same value at different times if the same random vector (*RV*) is used. To circumvent this possible security hole, Suh and al. propose to replace *RV* [12] by a counter incremented on each write operation and to re-encrypt (with a new key) all the read write memory section when the counter reaches its limit.

The main drawback of such a scheme is on write operations because CBC encryption is done sequentially, drastically increasing the memory access latency. Moreover on write smaller than 512 bits, a RMW operation is required; hence at least two AES blocks must be loaded to be able to modify the matching data and to re-encrypt it. In the worst case, when the write affects B[1], the whole chunk must be loaded and re-encrypted because with CBC encryption a given ciphertext influences the encryption of the following ones in the chain.

A third work, called XOM (eXecute Only Memory) [11, 39, 40], implements a direct encryption engine based on the 3-DES [51] block encryption algorithm. [41] proposes a performance evaluation with the following parameters for the simulation framework: 32KB separated L1 instruction and data cache and 256KB unified L2, 100 cycles of memory access latency and 50 cycles for the encryption latency. The simulation results show a performance slowdown of 20.8% on average and of 41.81% in the worst case scenario.

Concerning commercial devices, Dallas Semiconductors updated their family of secure microcontrollers – DS5250[42] – by removing the address bus encryption and by implementing a DES (and 3-DES) algorithm since the attack of Markus Kuhn (Chapter 2, section 2-1.3). Moreover, several patents (VLSI Technology [43], General Instrument [44]) surveyed in one of our conference papers [45] propose encryption engines based on similar schemes as the one presented above.

3-1.2. One Time Pad (OTP)

The implementation of One-Time Pad engines described in the literature are based on the CTR block encryption mode (synchronous state cipher – Chapter 1). The main advantage of such a scheme for memory encryption is that a preprocessing step corresponding to the keystream (or pad) generation can be parallelized with the memory access latency if the counter value is known by the processor. Hence, ideally, the latency overhead is only introduced by the XOR operation between the ciphered data and the pad on read operations. The XOR operation can be done in one cycle. However some key points must be respected to securely implement an OTP engine in our application domain. First the engine must be able to produce on decryption the same pad as the one used on encryption for any chunk by considering that the memory is randomly accessed. As a consequence, it must be able to

retrieve the counter position corresponding to any chunk. Then, as highlighted in Chapter 1, for the schemes based on stream ciphers to be secured a key stream sequence must never be used twice.

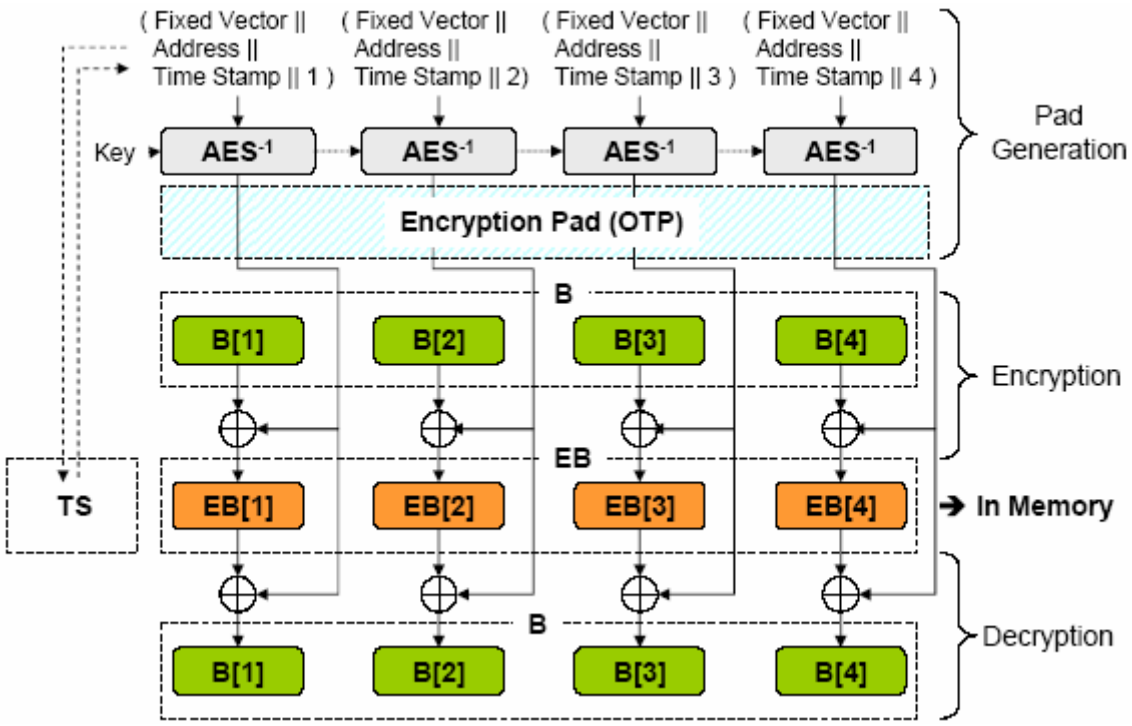


Figure 3-2 One-Time Pad encryption scheme (AES-CTR) proposed in the last version of the AEGIS processor

Mainly two academic works propose the implementation of an OTP for memory encryption. The first one is incorporated in the most recent version of the AEGIS processor [13, 37]. As for direct encryption, the chunk is of a cache block size. Figure 3-2 (taken from [37]) depicts an example of implementation for a 512-bit cache block (divided into 128-bit sub-blocks: B[1], B[2], B[3], B[4]). The pad used to encrypt or to decrypt a sub-block in a given chunk is generated by the AES decryption on (*V* || *Add* || *TScounter* || *i*) which represents the counter value. || is the concatenation operator. *V* is a fixed vector randomly chosen that makes the counter 128-bit. *Add* is the chunk address. *i* is the position of a 128-bit sub-block in the chunk. *TScounter* is generated on-chip by a 32-bit global counter on each write operation (⇔ encryption) of a RW data and is stored in the off-chip memory. *TScounter* is equal to zero for RO data. On read operation, for RO data the keystream generation can be fully parallelized with the memory access latency since all values composing the counter are known on-chip. However for RW data the *TScounter* value must first be loaded from the

external memory to compute the pad and hence to decrypt the requested data. Moreover, to avoid the generation of the same pad twice when the global counter reaches its limit, Suh and al proposes to re-encrypt the whole protected memory with a new key. Performance evaluation of this OTP scheme alone has been done for high-end processors [13, 37] using a 256 bytes on-chip buffer for time stamps and with the same architectural parameters as the one chosen for their direct encryption scheme. It is shown that the performance degradation remains under 18% in the worst case scenario. However, it is not indicated if this evaluation considers the re-encryption overhead. They estimate the re-encryption period between 35 minutes and 5.35 hours on a 1 GHz processor and that one re-encryption takes less than 300 million cycles. The memory overhead induced by the off-chip storage of *TScounter* is of 6.25% of the encrypted memory space dedicated to RW data.

A similar work is proposed by Yang and al. [41, 46]. The main difference comes from the fact that they use one sequence number (16-bit) per chunk rather than a global timer (*TScounter* of 32-bit) for RW data. As a consequence, it is required to load its value not only for decryption as in the previous scheme but also for encryption in order to increment it. Despite this disadvantage, the performance evaluation presented in [41] is better since they use a large on-chip cache (64 KB) to store those sequence numbers. Another key point explored by Yang and al. to improve performance is the exploitation of the "write buffer storage waiting time" to hide encryption latency. In most processors, buffers are used to store data to write until the bus is free. Hence, on encryption the pad generation is launched as soon as the data to write enter the write buffer, profiting from the storage waiting time in this buffer. The performance degradation is estimated to 6.7% in the worst case scenario and to 1.28% on average on a high-end processor with two levels of cache (L1 I-cache = 64KB, L1 D-cache = 64KB, unified L2 caches = 256KB with 1024-bit L2-cache block), and considering 100 cycles of memory access latency and 50 cycles for the AES latency. Those figures do not consider re-encryption overhead when a sequence number reaches its limit. However the re-encryption may occur less frequently than for Suh's scheme since a sequence number should be incremented slower than a global timer. The off-chip memory overhead, for the architecture parameters chosen, is about 1.5% of the amount of the encrypted RW (16-bit sequence number for each chunk). This low overhead is achieved by the use of large chunk size: 1024-bit cache block. However, to maintain such performance, a large on-chip cache (64

KB) dedicated to the sequence numbers is required whereas the AEGIS processor only uses a small buffer (256 B).

Concerning security, Suh describes in [37] two potential security holes of the OTP implementation proposed by Yang and al. Both security holes are due to the generation of a same pad to encrypt (respectively decrypt) two different plaintexts (respectively ciphertexts). First, the fact that the sequence numbers are stored off-chip allows an adversary to choose the value used to generate a pad. Hence the attackers could replay the same sequence number on each data read by forcing the same encryption pad to be used on write back. This attack is not feasible in AEGIS since the *TScounter* value is generated on-chip, therefore the adversary cannot tamper with it. A way to solve this security hole for the Yang OTP is to check the integrity of sequence numbers. The second security weakness comes from the generation of the seed used to produce the pad via its AES encryption. This seed is created by the addition of the chunk's (or cache block) virtual address *VA* and the sub-block sequence number *SN*. As a consequence, the same seed is generated for two chunks with $VA_1 = a$ and $VA_2 = a + x$ if the relative sequence numbers are equal to $SN_1 = b + x$ and $SN_2 = b$, resulting in the same pad after the seed encryption. A simple way to solve this security hole is to concatenate the virtual address with the sequence number instead of adding both [37].

Both implementations require a RMW operation on write of data smaller than the chunk size. However, the OTP schemes allow to reduce the corresponding performance overhead. Indeed, in Suh's OTP implementation, the use of an on-chip global timer enables the generation of the pad for the chunk to write during the read memory access latency. Therefore the re-encryption step of the RMW operation results simply in a XOR operation (1 cycle). This also applies to the Yang scheme if the matching sequence number is in the dedicated cache.

The Suh and al. encryption scheme is the most secure implementation of OTP but Yang and al. propose an interesting trade-off between performance and security.

3-1.3. Summary

Table 3-1 sums up the characteristics of each family of memory encryption engines. The preprocessing feature of OTP schemes makes them more efficient than direct encryption

engines; however, their main drawbacks are the sensitive management of the counter and the re-encryption requirement of the whole memory when such a counter reaches its limit.

Table 3-1 Summary of the existing memory encryption engines

	Direct Encryption (Gilmont, AEGIS 1st version, XOM, DS5250)	**One Time Pad – Counter Mode** (AEGIS 2nd version, Yang)
Security	+ + +	+ + +
Performance (latencies)	-	+
Off-chip Memory Consumption	Low	Low
Silicon Area Usage	- (Block Encryption and Decryption core)	+ (Block Encryption or Decryption only core)
Comments	• Straightforward implementation (+)	• Sensitive management of the counter (-) • Require periodic memory re-encryption (-)

3-2. Memory Integrity Verification Engines

The integrity checking objectives in our application context is to detect any code or data injection or corruption on the processor-memory bus and thus to forbid the execution of intentionally altered memory content.

Gilmont and al. [36] claims to protect the integrity of the data stored off-chip with encryption at run-time. They base their theory on the fact that the encryption is salted with the virtual address; therefore an adversary who wants to modify an encrypted data or to swap memory blocks in the memory has little chance to induce a valid behavior. While this assumption may be considered for static code since the key is different for each application loaded in memory, it is not true for dynamic data because a replay attack could easily be conducted with a predictable behavior. For example, a loop counter can be replayed by an adversary to freeze the code loop execution or to control the number of loop iterations wanted. Moreover, Markus Kuhn's attack demonstrates that using the address in the encryption does not provide a sufficient countermeasure against a ciphered instruction searched attack. An efficient integrity checking engine must come with a tamper detection

mechanism to avoid the feasibility of such kinds of attacks and to prevent the processing of altered data.

In the following, we present a study of the schemes proposed in the state of the art to check data integrity. We first discuss the MAC based solutions and we show their limitations. Then we present hash tree schemes which allow to handle the specific issue of replay attacks.

3-2.1. Integrity Checking Engines Based on MAC algorithms

In the XOM project, the principle of memory integrity verification presented in Chapter 2 (section 2-3.2.3) is implemented. Therefore, the tag computed on the chunk thwarts spoofing attacks since a modification of the data will be detected by the tag matching process. The MAC algorithm used is not specified but to prevent splicing attacks, the address is enrolled in the tag computation (Addressed-MAC).

Many embedded systems are running operating systems (OS) which are based on preemptive multitasking: the OS periodically interrupts tasks, stores the context (on-chip registers, stack pointer, frame pointer…) off-chip and restores it later. A possible attack consists in replaying a previously recorded context. The countermeasure proposed by XOM to avoid such a replay attack is to use a "register key". The value of this register is modified on each interruption and is given as an additional input of the MAC algorithm. However as stated in [40, 47, 48] this countermeasure does not prevent other RW data from being replayed in the external memory. For instance, a loop counter can be replayed in the external memory to extend its duration. The integrity engine based on a addressed-MAC scheme implementation cannot detect it since the information used for the tag computation – the replayed data and its address - remains the same at a given address.

A solution might be to enroll secret random values in the tag computation. Such a solution has never been explored in our application domain and will be deeply studied in Chapter 4, 5 and 6.

In [49] Merkle proposes a technique, called hash trees, which allows to solve the issue of replay attacks. This technique, thoroughly implemented in our application domain in [13, 37, 47], is presented in the next sub-section.

3-2.2. Hash Trees

A simple solution to thwart replay attacks would be to store the tags or the hashes of each chunk on-chip. In this way tags or hashes are inaccessible from adversaries and any corruptions of the loaded data are always detected by the tag (or hash) verification process. To limit the on-chip memory overhead two options are possible: i) generate smaller tags ii) attach a tag to a chunk composed of multiple cache blocks (⇔ extend the chunk size). While the latter might induce a great loss of performance by increasing the memory bandwidth usage, the former might dramatically reduce the robustness of the integrity checking engine since more chunks would correspond to the same tag. However, even with a small tag or hash, the on-chip memory overhead would be unaffordable.

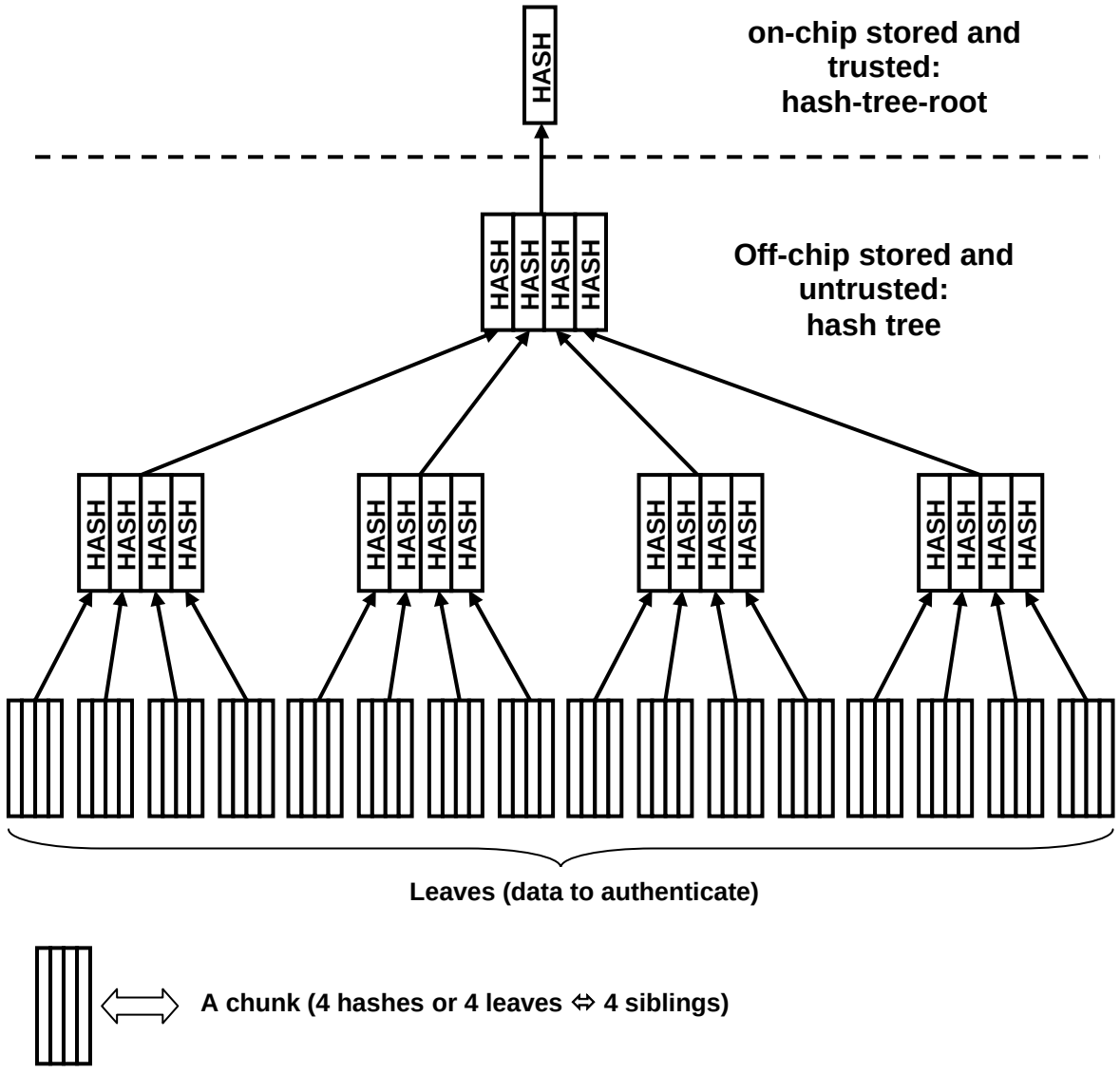


Figure 3-3 A balanced 4-ary hash tree

Merkle or hash trees solve the latter issue by securely storing chunk hashes off-chip. The principle is to keep a trusted value (hash tree root) on-chip which reflects the current external memory state. This value is updated on each write operation and checked on each read operation through a tree[11] of hashes computed over the memory to authenticate (Figure 3-3). Each element of the tree has the size *h* (in bit) of a hash. The leaves are the *h*-bit data blocks composing the memory. Each internal node is the collision-resistant hash of its children chunks (containing *m h*-bit blocks or hashes). The elements issued from the same parent in the tree are called siblings. It results in an *m*-ary hash tree.

The integrity checking process on each *read operation* works as follows:

i) Read the matching chunk - composed of the fetched data and its sibling in the tree - and its parent hash in the untrusted memory
ii) hash the chunk
iii) compare it with its parent hash.

The same steps are repeated with the parent hash and its parent until the root of the tree is reached. Since the latter is stored on-chip, it cannot be tampered with by an adversary; hence it is trusted. If one of the hash verification fails, this means that a data or a hash has been corrupted in memory; the process stops and the integrity checking engine raises an exception.

On *write operations*, the tree must be updated, thus the following steps are required:

i) authenticate the matching chunk (with the process described for read operations)
ii) perform the write in the loaded chunk
iii) hash the chunk
iv) update the parent in the tree with the resulting hash

Similarly to read operations those steps are repeated with the parent hash all along the path from the leave to modify to the root of the tree. The authentication of each chunk loaded on write operations is mandatory otherwise an adversary could corrupt the sibling of the fetched data during this process without detection.

[11] Refer to Chapter 6 section 1 for the definition of a tree structure.

The overhead in memory consumption to store an *m*-ary hash tree is constant and equal to $1/(m-1)$. The number of hash verifications on a read operation is of $\log_m(N)$ with N the number of *h*-bit block composing the memory to protect.

On read operations, the hash computations are parallelizable during the integrity checking process and the degradation mainly emanates from the memory bandwidth pollution generated by the huge amount of meta-data (hashes belonging to the same branch in the tree) to load. Moreover, on write operations the update of the tree is a sequential process. All those issues make the direct implementation of the hash tree principle clearly unaffordable for a computing system. In [47] Gassend and al. show that the corresponding performance slowdown can reach a factor 10. To decrease this overhead they propose to exploit the principle of locality of cache memory and to apply it to hash values (Cached hash tree). When a chunk of hashes is loaded on-chip it is stored in the cache memory after being checked. The cached hashes are trusted since stored on-chip; as a consequence the integrity checking process on read operations stops when a hash in the tree is fetched from cache. Such a solution also improves performance on write operations if the intermediate hashes in the tree branch to update are cached. However, this update process still requires serializing hash computations.

Suh integrates cached hash tree principle in the AEGIS processor. A first implementation of such a tree for embedded systems (25MHz processor, 12.5MHz SDRAM memory, 64-bit bus width, 32KB I/D cache, 512-bit cache blocks) is proposed in [12, 37]. The scheme protects 4MB of memory against replay and uses an additional hash cache of 16KB. The performance slowdown is evaluated to 18.9% when the data cache miss rate is of 12.5%. Concerning high-end processors, the performance slowdown is evaluated to 20-30% on average and to 50% in the worst case scenario (256KB L2 cache used also to cache hashes) in the same simulation framework as the one considered for their direct encryption scheme. However, the performance overhead of hash trees greatly depends on several parameters: cache block and chunk size, amount of memory to protect, hash length, processor-memory bus width and data cache size. This point is further detailed in chapter 6.

Another scheme based on hash trees called Log Hash Tree is proposed in [13]. However, the integrity checking is done at regular intervals, thus giving the adversary potentially enough time to perform an attack.

3-2.3. Summary

Among the existing solutions to check memory integrity (Table 3-2), only hash trees provide a countermeasure against all active attacks presented in Chapter 2 (spoofing, splicing, replay). However such a solution suffers from some drawbacks. First, the update of hash trees on write operations is not a parallelizable process. Then, despite the cached hash mechanisms, the memory bandwidth pollution generated by the loading of several hashes during data authentication or the tree update remains high. Finally, large hash values are required to resist collision, implying a non-negligible off-chip memory usage.

Table 3-2 Summary of solutions achieving memory integrity checking

	Addressed-MAC (XOM)	**Cached Hash Trees** (AEGIS)
Security	- (no countermeasure against replay	+ + +
Performance – Memory bandwidth consumption	+	-
Off-chip Memory consumption	Low	High

3-3. Memory Encryption and Authentication: Techniques and Related Works

In this section we present the three existing techniques providing data confidentiality and integrity and the related engines implemented to protect the off-chip memory content. First we present the principle of generic composition consisting in pairing an encryption mode with an integrity checking technique. Then we present the concept of Added Redundancy Explicit Authentication (AREA). Finally, the authenticated encryption mode candidates proposed to the NIST standardization process are described.

3-3.1. The Conventional Way: Generic Composition Schemes

3-3.1.1. Principle

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 (or a hash) algorithm. The three possible schemes defined in [50] are depicted in Figure 3-1:

- *Encrypt-then-MAC* (Figure 3-1a) encrypts the plaintext to get a ciphertext *C*, and then appends to *C* a tag *T* computed with a MAC algorithm over *C*.
- *MAC-then-Encrypt* (Figure 3-1b) calculates a tag over the plaintext *P*, appends the resulting tag to *P* and then encrypts them together.
- *Encrypt-and-MAC* (Figure 3-1c) 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 read or on 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.

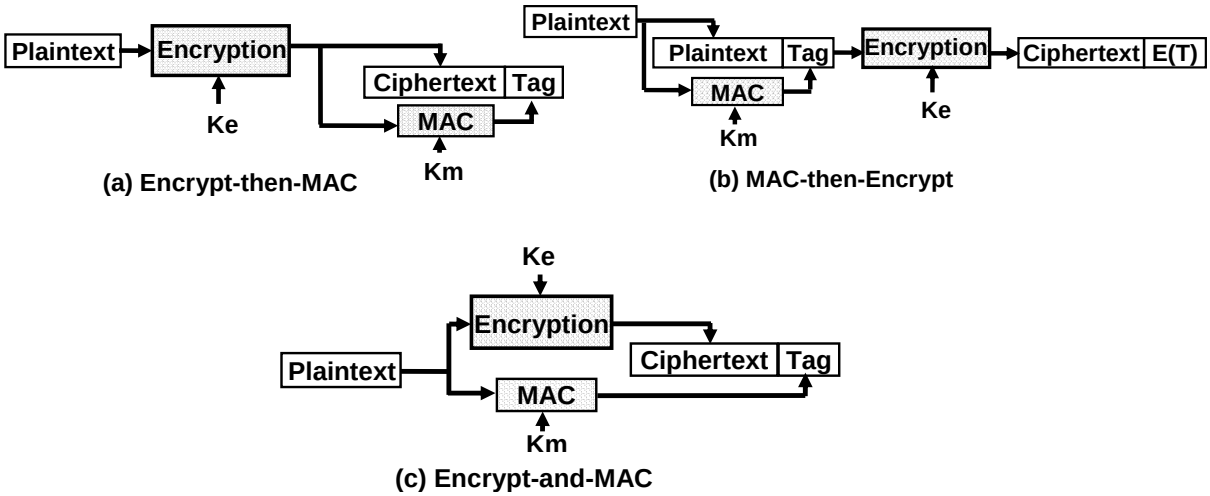


Figure 3-4 The conventional way to provide data confidentiality and integrity: The generic composition schemes

In [50], 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.

3-3.1.2. Off-Chip Memory Protection Engines Based on Generic Composition

The works presented in this section aim to provide trusted processor architecture with different objectives. However, they all need to protect the processor-memory transaction to thwart non-invasive physical attacks. Thus, they all implement a generic composition scheme by considering the same security perimeter: the System on Chip (SoC).

3-3.1.2.1. AEGIS

The AEGIS processor objective is to offer a trusted computing platform for a large set of applications with a special focus on distributed computation. They consider software as well as physical attacks. Concerning the latter issue, Suh and al. implement a generic composition scheme constructed with the OTP engine presented in section 3-1.2 and a MAC algorithm for RO data and a cached hash tree for RW data (as presented in section 3-2.2) in the Encrypt-then-MAC fashion. For an embedded processor (50MHz) with 32KB of data cache, 16KB of hash cache and 4KB of time stamp cache, the performance slowdown is low and evaluated at 25% for a data cache miss rate of 12.5%. However the memory region protected against replay by the hash tree is limited to 16MB. By reducing the data cache to 4KB and the hash cache to 2KB the degradation can reach 73%. Such an engine requires 3 AES "encryption only" cores and 5 SHA-1 hardware copies. The off-chip memory overhead is of 33% of the amount of memory immune against replay – hash tree storage - 25% of the RO memory section – MAC (tag) storage - and of 6.25% of the RW memory region – time stamp storage. For a high end processor (1GHz, 200MHz and 64-bit memory bus, 256B on-chip buffer dedicated to time stamps) and in the case where all data and instructions are protected, the performance slowdown is evaluated at 40% on average (256KB unified L2 cache) and at 60% in the worst case scenario.

3-3.1.2.2. SP – Secret Protected

The SP (Secret Protected) architecture (formerly VSCoP – Virtual Secure Coprocessing) focuses on the protection of critical information such as cryptographic keys contained in mobile devices and of sensitive computations involving the management of those keys. 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. The integrity of the TCM code is ensured with a CBC-MAC (AES) scheme while for data, both confidentiality and integrity are provided by pairing AES-CBC encryption and the AES-CBC-MAC with the Encrypt-then-MAC construction. The chunk size in SP is of a cache line (512-bit). To thwart replay attacks they propose to implement a hash-tree scheme similar to the one of AEGIS.

A particularity of SP is that the targeted device does not contain permanent secret (i.e. private key). A secure I/O device is designed to record a pass-phrase entered by the user. This pass-phrase is hashed to be used as the master key enrolled in the hardware cryptographic computations. In this way the sensitive data are directly associated to the user rather than the device.

The SP architecture fulfills its objective efficiently. Considering 100 cycles of initial off-chip memory latency, separated L1 cache with 64KB of I-cache and 64KB of D-cache, and a 2 MB unified L2 cache, the performance slowdown is kept under 1%. The overhead is negligible because the amount of data to protect is low compared to the unsecured ones, inducing an increase of the off-chip memory latency only on a small number of memory accesses. Those results do not consider the implementation of hash tree but for the same reasons the overhead should remain of the same order.

3-3.1.2.3. XOM

XOM [11, 39, 40] 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. To thwart physical attacks, an Encrypt-then-MAC construction of a direct encryption mode paired with an addressed-MAC is proposed. However, as highlighted in Chapter 2, XOM failed in preventing replay attacks. Moreover, the architecture of the off-chip memory protection engine and its performance evaluation are not detailed.

3-3.1.2.4. Summary

The most secure engines providing both data encryption and integrity checking are those implementing hash trees because they offer a countermeasure against replay (Table 3-3). However, only the SP architecture obtains negligible performance overhead by focusing on the protection of critical information such as keys and code required to manage them.

Table 3-3 Summary of the memory protection engine (encryption and integrity checking) based on generic composition

	Encrypt-then-MAC construction		
	XOM	AEGIS	SP
Security	- (MAC - no protection against replay)	+ + + (Merkle Trees)	+ + + (Merkle Trees)
Performance	N/A	-	+
Off-chip Memory Consumption	N/A	high	low
Comments	N/A	Performance evaluation done on complete applications	Only information related to secret keys are protected

In the following, the underlying block cipher processes n-bit blocks under k-bit keys. E_K and D_K are respectively the encryption and the decryption functions under the key K. The message to encrypt M is divided into m n-bit plaintext blocks P_i with $(1 \leq i \leq m)$. Similarly the ciphered version of M is divided into m n-bit ciphertext blocks C_i with $(1 \leq i \leq m)$.

3-3.2. AREA: Added Redundancy Explicit Authentication

The principle of the AREA 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 M to encrypt. Similarly, a cipher mode has infinite error propagation on decryption if a plaintext P_i can be expressed as a function of all previous ciphertext blocks C_i to C_1 in the encrypted message $E_k(M)$. For instance, CBC has infinite error propagation on encryption since a given ciphertext block can be written as a function of all previous plaintext blocks, but has limited error propagation on decryption since a given plaintext 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 appended at the end of the plaintext message before encryption. In this way, the result of the encryption of P_{m+1}, C_{m+1} will depend on all plaintext blocks P_i composing the message to

authenticate. If the error propagation of the underlying cipher mode is not infinite, distinct messages leading to the same C_{m+1} can be easily found. P_{m+1} and C_{m+1} are sent along with the encrypted message. 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 (infinite error propagation). The recipient can detect a malicious modification by comparing the decryption of C_{m+1} with P_{m+1}.

An example of cipher mode usable for an AREA scheme is the PCFB (Propagating Cipher FeedBack) mode which is described in the following section.

AREA schemes seem really efficient since only one pass over the data is required to provide both data confidentiality and authentication on encryption and decryption (i.e. on write and read operations in our application domain). However, the infinite error propagation is usually achieved by chaining encryption (e.g. CBC, PCBC[12], and PCFB) or decryption (e.g. PCBC, PCFB) operations, making parallelization impracticable.

3-3.3. Authenticated Encryption Modes

As shown in [50] and [53], 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.

This section describes those modes while keeping our objectives in mind. For instance, some of the presented AE modes, also called AEAD modes - where AEAD stands for Authenticated-Encryption with Associated Data - allow authenticating some data without having to encrypt them. However, the objective of this work is to provide data authenticity-and-confidentiality, therefore this point is not furthermore developed in this thesis, for more details see the corresponding reference. Moreover, one of our interests is to reach such an objective by introducing low latencies on both write and read operations. Considering the latter aspect this section is divided into three parts. First we give an overview of the AE modes including an expensive non-parallelizable computation in their construction, either for

[12] PCBC stands for Plaintext Cipher Block Chaining or for Plaintext-Ciphertext Block Chaining. It has never been standardized as a cipher mode. Several versions and definitions of PCBC have been proposed as AREA schemes. [53] presents the different existing PCBC modes and describes potential attacks.

encryption or for tag calculation. Then we describe the operations of the AE modes classified by the NIST as parallelizable. Finally, we discuss the implementation of the most relevant AE modes for protecting the off-chip memory content of computing systems.

For clarity we do not detail the mathematical concept behind those modes in the following section, we only sketch a simplified description of the proposed AE modes to allow the reader to realize the cost of a hardware implementation in a SoC.

Moreover, the presented authenticated encryption modes are only described for messages of length which is a multiple of the block size of the underlying block cipher.

3-3.3.1. Authenticated Encryption Modes with Non-Parallelizable Operations

3-3.3.1.1. CCM - Counter CBC-MAC

CCM [55] is a secure combination of two existing modes using a single key: CTR mode for encryption and CBC-MAC for authentication. CTR is highly parallelizable and pre-processing is feasible: as discussed in Chapter 2, keystream (or pad) sequences could be generated in parallel before data is available if the counter initialization is known. The main drawback concerning our application is the use of CBC-MAC which implies serialized block encryption and prevents any parallelization. Moreover, CCM follows the Encrypt-then-MAC construction and on write operations the tag computation can only start after the end of the encryption of the first plaintext block.

3-3.3.1.2. EAX - Encrypt Authenticate Translate

Similarly to CCM, EAX [56] is the generic composition of an authentication mode, OMAC (One key CBC-MAC [24, 57]), and of an encryption mode, CTR, in the Encrypt-then-MAC fashion. EAX also uses a single key. It allows to enhance the security of CCM regarding authentication particularly for messages of varying length [Dent]. However, such a point is not of our concern since in our application domain we process blocks of fixed size (usually defined by the chunk size). The same CCM drawbacks apply to EAX since OMAC is still a recursive construction.

3-3.3.1.3. PCFB - Propagating Cipher Feedback

PCFB [58] is a stream cipher which mixes OFB and CFB modes. The output algorithm is an *n*-bit block cipher and the internal state is the concatenation of *l*-bit of the previous ciphertext and of ($n - l$) bits of the previous key stream. In this way, it results in an infinite two-way error propagation. In order to provide authentication, PCFB works like an AREA scheme. Despite the fact that this mode is the only one to perform one pass over the data to provide both confidentiality and integrity, its two-way error propagating inherent feature makes it not parallelizable on both encryption - write operations - and decryption - read operations: in order to generate a keystream for the current encryption (respectively decryption), the previous ciphertext and keystream (respectively keystream) are required.

3-3.3.1.4. IACBC - Integrity Aware Cipher Block Chaining

IACBC [59] uses two keys (*K0* and *K1*) and requires a random initialization vector *r* for each message processed. It implements the CBC mode for encryption under *K1* with post-whitening (after encryption) with an *n*-bit vector S_i: $C_i = E_{K1}(P_i \oplus C_{i-1}) \oplus S_i$ with $C_0 = E_{K1}(r)$ (Figure 3-5). The whitening step consists in the addition of a key-dependent value to break the pattern of the text on which it is applied. To obtain the set of S_i, a subset of $t = \lceil \log_2(m+1) \rceil$[13] new random and independent vectors W_i is computed from the encryption of *r* under *K0*. Then, a Gray code controls the generation of the pairwise independent S_i vectors. To obtain the authentication tag *T*, the checksum (XOR summation) of the plaintext block composing *M* (checksum = ΣP_i) is first xored with the result of the last plaintext block encryption before its post-whitening step, then block encrypted and finally whitened with S_0.

The shortcoming of IACBC is the use of the CBC principle which prevents any parallelization during encryption (write operations). On decryption (read operations), IACBC is parallelizable but after two serialized block cipher invocations: one to obtain *r* (assuming that *r* is not known by the decrypting party) and one to obtain S_1. The integrity checking process is parallelizable with the ciphertext decryption since the verification can be done by comparing the *T* decryption result with the checksum obtained at the end of decryption.

[13] $\lceil X \rceil$ denotes *X* rounding up.

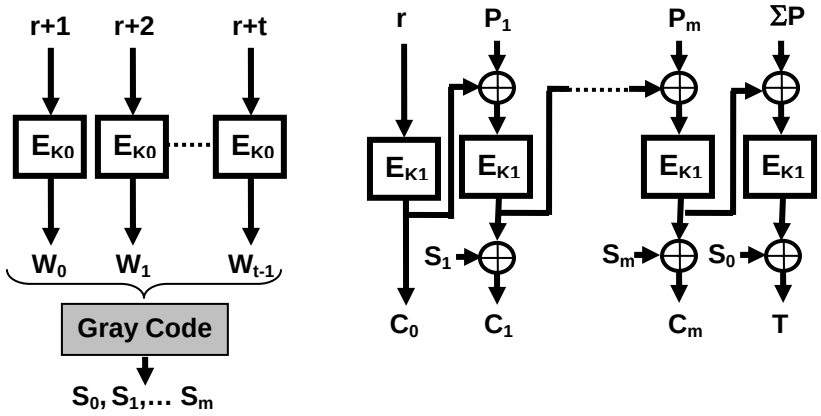


Figure 3-5 IACBC - Integrity Aware Cipher Block Chaining

3-3.3.1.5. XCBC-XOR

XCBC-XOR [60] is very similar to IACBC since it is based on CBC principle and on post-whitening. The main difference comes from the whitening step which is done with an addition in the group of natural number modulo 2^{128} and not in a binary Galois Field. Whereas the latter is simply performed with a bitwise XOR, the former requires managing a carry-bit through the adder which is less efficient in hardware. Moreover, in XCBC-XOR the whitening value is obtained by multiplying the position i of the currently processed block P_i in the message M by the initialization vector r: $S_i = i \times r$.

3-3.3.2. Parallelizable Authenticated Encryption modes

3-3.3.2.1. IAPM – Integrity Aware Parallelizable Mode

IAPM [59] was proposed by Jutla (also inventor of IACBC) as the refinement of the IACBC to make it parallelizable. The recursiveness of the CBC-like encryption was removed and replaced by a pre-whitening step on the plaintext with the corresponding pairwise independent vector S_i: $C_i = E_{K1}(P_i \oplus S_i) \oplus S_i$ with $C_0 = E_{K1}(r)$(Figure 3-6). To obtain the authentication tag T, the checksum (ΣP_i) of the plaintext block composing M is processed like a plaintext block with S_0 as whitening value.

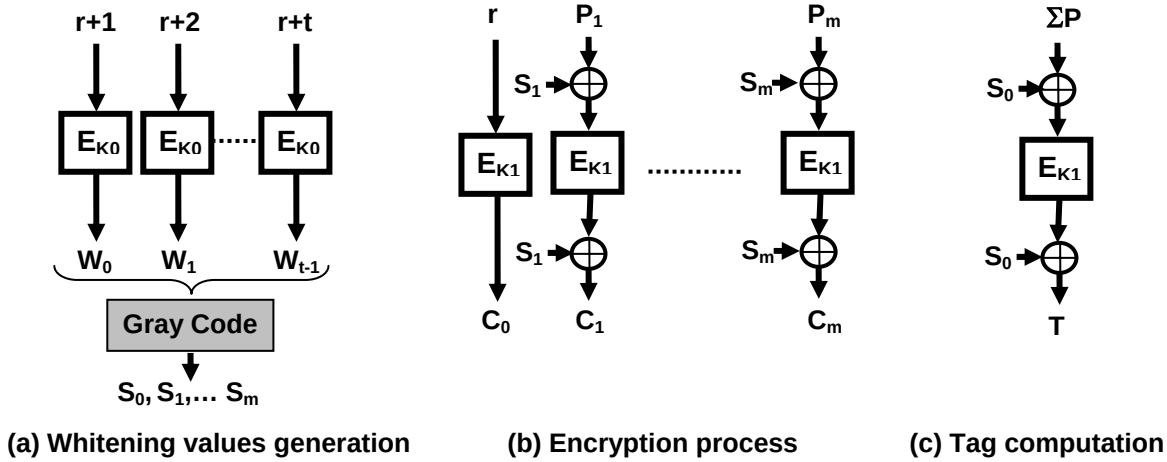


Figure 3-6 IAPM - Integrity Aware Parallelizable Mode

Similarly to IACBC, IAPM requires two serialized block cipher invocations before starting data processing: one to obtain r (assuming that r is not known by the decrypting party) and one to obtain S_1. This non-parallelizable step can be considered as negligible when long messages are processed and when the throughput is privileged to the latency.

3-3.3.2.2. XECB-XOR

Like IAPM for IACBC, XECB-XOR [60] removed the recursiveness of XCBC-XOR with a pre-whitening step. However, XECB-XOR suffers from the same drawbacks as XCBC-XOR: the underlying algebraic structure is not suited for hardware implementation.

3-3.3.2.3. OCB – Offset Code Book

OCB [62] is a refinement of IAPM, proposed by Rogaway. The encryption process (Figure 3-6b) is the same; the differences come from the whitening values and the tag generations. Moreover, OCB uses a unique encryption key K and requires an initialization vector N which only needs to be a nonce (a Number used ONCE).

Let L be the encryption of the n-bit null vector under K: $L = E_k(0^n)$ and γ_i a Gray code with i the position of the processed block in the message M. Moreover R is the encryption under K of the addition (XOR) in the binary Galois Field addition GF(2^n) of N and L: $R = E_k(N \oplus L)$.

Thus, the whitening value S_i used in the encryption of P_i is computed by:

$$S_i = (\gamma_i \otimes L) \oplus R$$

where $\otimes$ denotes the binary Galois Field GF(2^n) multiplication. The ciphered text C_i is obtained as for IAPM with:

$$C_i = E_K(P_i \oplus S_i) \oplus S_i$$

The tag results from the encryption under K of the XOR operation between the checksum of the plaintext block (ΣP_i) and the last whitening value S_m:

$$T = E_K(\sum_{i=1}^{m} P_i \oplus S_m)$$

Assuming that L is produced in advance and memorized, OCB is parallelizable on encryption and on decryption after one block cipher invocation which corresponds to the computation of R.

Rogaway proposes to truncate the tag T. If such an operation is performed the integrity checking process is no more parallelizable with the decryption since the tag cannot be simply decrypted and compared to the checksum of the decrypted block xored with the last whitening value.

Note that IAPM as OCB are not collision-resistant since the tag is computed on a XOR summation. As a consequence a bit flip in the same position in two plaintext blocks of a message will yield in the generation of the same tag for the original message as well as for the tampered one. However, it is not of our concern since in our context the adversary does not have access to the plaintext. This attack does not apply to IACBC since the CBC-like construction induces infinite error propagation until the checksum on encryption; hence a modification of one bit in the message affects the tag computation.

3-3.3.2.4. GCM – Galois Counter Mode

GCM [63] is an Encrypt-then-MAC construction which uses the CTR mode for encryption and a MAC algorithm based on the Carter-Wegman[64] design under a unique key K.

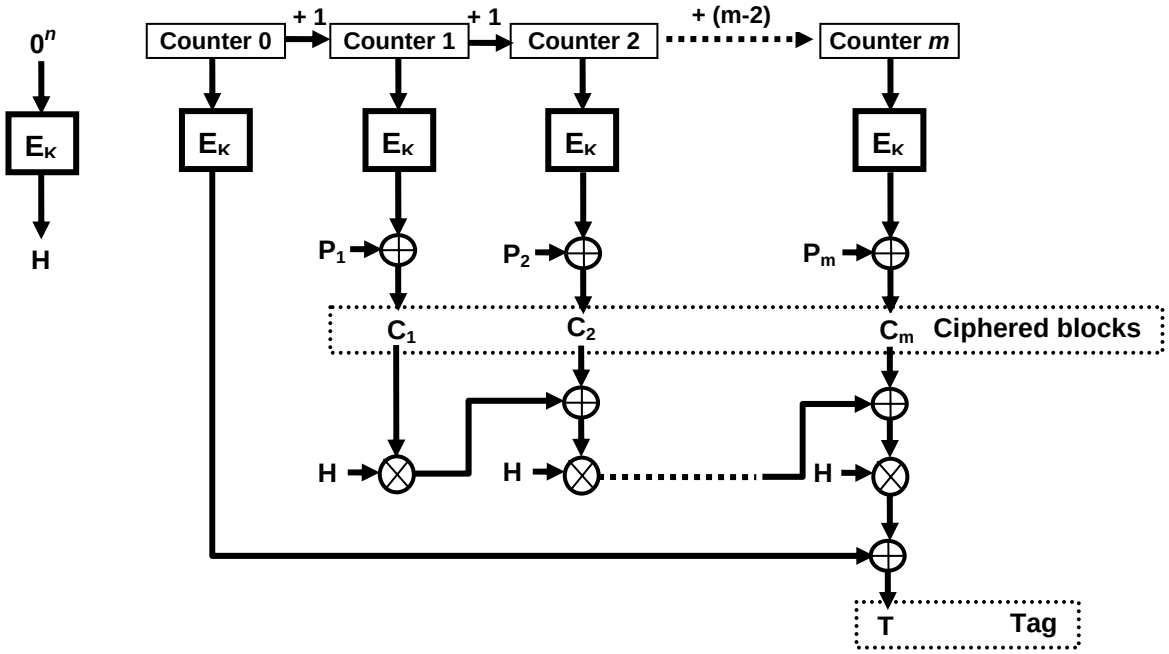


Figure 3-7 GCM -Galois Counter Mode (The message to encrypt is a multiple of the block length of the underlying block cipher)

Let H be the encryption of the n-bit null vector under K: $H = E_k(0^n)$ and GHASH(x) the function consisting in the multiplication in GF(2^{128}) of x with H (Figure 3-7). To obtain a hash of the ciphered message, GHASH(C_1) is first performed and then GHASH(GHASH(C_{i-1}) $\oplus$ C_i) is recursively computed. The final tag T results from the encryption of this hash with the CTR mode; hence T is computed with:

$$T = E_K(Counter0) \oplus GHASH(GHASH(C_{m-1}) \oplus C_m)$$

In GCM, the Encrypt-then-MAC scheme is implemented, thus encryption and tag computation are not parallelizable. Moreover, the tag computation is a recursive process. Nevertheless [65] shows that a multiplication over the binary Galois field GF(2^{128}) can be done in hardware in less than one cycle at 271 MHz, making GCM an efficient authenticated encryption mode.

3-3.3.2.5. CWC – Carter-Wegman authentication with Counter

Similarly to GCM, CWC [66] pairs a CTR encryption mode with a Carter-Wegman MAC. However, CWC suffers from many drawbacks when compared to GCM [67]. In particular, the proposed implementation of Carter-Wegman MAC computes a hash consisting in a

multiplication in a prime Galois field GF(2^{127}-1) which is less suitable in hardware than a multiplication in the binary Galois Field GF(2^{128}). In [68] they estimated the cost of the integer-based hash function (CWC) to 100K gates while in [63] the hardware resources for a binary field hash function are evaluated to 30K gates.

3-3.3.2.6. CS – Cipher State

Cipher State [69] mode proposes a totally different approach. Instead of pairing two schemes, one for encryption and one for authentication, CS retrieves information during encryption between the underlying block cipher rounds and computes an authentication code from them. This technique seems interesting but does not catch the attention of the research community since, contrarily to the other AE modes, there are no comments or proposed implementation of such a mode.

3-3.3.3. Discussion

The NIST has recommended two modes: the CCM for short tag [71] and the GCM [72] for long tag. Nevertheless CCM is discarded for our application due to its non-parallelizable tag computation which would greatly increase memory access latencies. Concerning GCM, [73] advises not to consider this mode when the tag should be truncated. Typically when authentication is done on short memory blocks it is desirable for us to have a short tag to save expense of off-chip memory.

Otherwise, IAPM and OCB is the NIST non-recommended[14] AE mode which seems to be the most relevant for our application because encryption and tag computation, and decryption and integrity checking are strongly parallelizable. However, one block cipher invocation for OCB and two for IAPM must be completed before those processes can start; such a non-parallelizable computation could increase memory access latencies. Moreover, when the tag is truncated, the integrity checking process is no more parallelizable with decryption.

The last comment on the AE modes is the fact that they all require to be tuned to prevent replay and splicing attacks when implemented to protect processor-memory communications.

We did not find SoC implementation of Authenticated Encryption modes to protect the off-chip memory content.

[14] The main reason for this rejection is the fact that they are patented.

3-4. Conclusion

In this chapter we have first presented the engines proposed in the literature to protect separately the confidentiality and the integrity of the off-chip memory content. We have highlighted that for two reasons the OTP encryption scheme (based on the CTR encryption mode) is the most efficient technique to be used in our application domain. First, a preprocessing step allows parallelizing the key stream preparation with the external memory access latency (particularly on read operations). Secondly, such schemes only need encryption-only cores which are less gate-consuming than encryption/decryption cores required by direct encryption. Concerning memory integrity verification, we have showed that hash trees are the only existing technique allowing to thwart replay attacks, but at a non-negligible cost in term of performance. Finally, in the third part of this chapter, we have described the techniques providing both data confidentiality and integrity: generic composition scheme, AREA and authenticated encryption. The generic composition scheme proposed by Suh and al. is secure but the main shortcomings are the hardware implementation of two algorithms, one for encryption and one for the integrity checking, and the non-parallelization of both processes (and the intrinsic non-parallelizability of hash trees) on write operations. Concerning the AREA technique, the requirement of two-way infinite error propagation leads to serialized computations, thus preventing latency optimization. Finally, the most efficient authenticated encryption mode seems to be GCM despite the recommendation of using long tag, because the tag computation is fast compared to classic hash or MAC functions. The CTR mode is implemented in GCM allowing preprocessing and mitigating the additional hardware cost of the multiplication over $GF(2^{128})$ – required by the GCM hash function. However, the GCM must be tuned to thwart splicing and replay attacks.

Chapter 4: PE-ICE - Parallelized Encryption and Integrity Checking Engine

The proposed *Parallelized Encryption and Integrity Checking Engine,* PE-ICE, is a dedicated solution guaranteeing the confidentiality and the authenticity of data transferred onto the processor-memory bus of a computing system.

The first objective of PE-ICE is to perform encryption and integrity checking in a parallelized way and hence to optimize latencies introduced by the underlying hardware mechanisms on read and write operations. Moreover, we achieve this task by keeping in mind a second goal which is to optimize the hardware resources usage. A third objective of PE-ICE is to decrease the memory bandwidth pollution generated by the encryption and the integrity checking processes on RMW operations.

This chapter is organized as follows. Section 4-1 gives an overview of PE-ICE. Section 4-2 describes the integrity checking process in PE-ICE and the property of block cipher on which it relies. Section 4-3 presents the encryption mode implemented and introduces the notion of *fine granularity of integrity checking*. Section 4-4 deals with issues of PE-ICE implementation in a SoC. Section 4-5 discusses the level of security offered by PE-ICE and the implementation key points to maintain it. Section 4-6 describes how to retrieve data processed by PE-ICE in memory. Section 4-7 evaluates the amount of on-chip and off-chip memory consumed by PE-ICE. Finally, section 4-8 sums up terms and PE-ICE parameters defined in this chapter and proposes a synthetic description of PE-ICE operations.

4-1. General Overview

PE-ICE may be viewed as a black box implementing a block cipher and performing one pass on the data to provide both confidentiality and authentication. On encryption (⇔ write operations; Figure 4-1a), the inputs of PE-ICE are the data to protect P_L[15] and a tag T – composing a plaintext block - and a secret key K; the only output is a ciphered block C. On decryption (i.e. read operations; Figure 4-1b), the inputs are a ciphered block C, a secret key K and a reference tag T'; the outputs are a payload P_L and an integrity checking flag which informs the processor on the validity of P_L regarding its integrity.

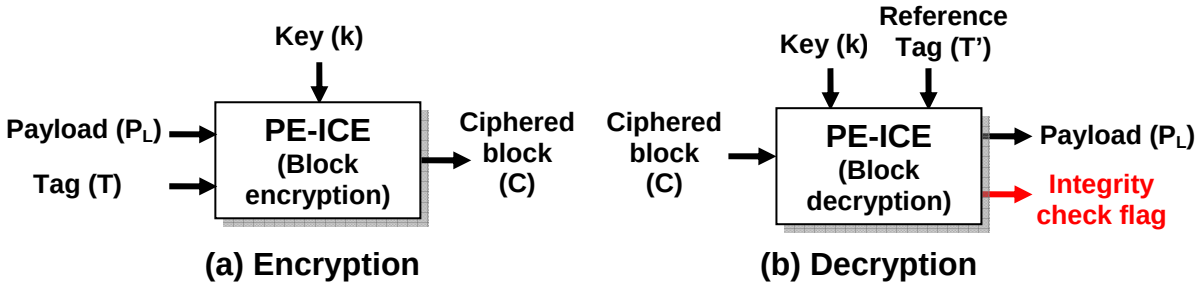


Figure 4-1 PE-ICE general overview

The block encryption provides the data confidentiality service. Data authentication is ensured in PE-ICE by adding the integrity checking capability to the underlying block cipher.

4-2. Adding the Integrity Checking Capability to Block Encryption

4-2.1. The Diffusion Property of Block Ciphers

In order to verify the integrity of data, PE-ICE relies on the diffusion property identified by Shannon [19] for block ciphers to be considered as secure. Theoretically a block cipher must be indistinguishable from a random permutation with equiprobable outputs from an adversary point of view, and the redundancy in the statistics of the plaintext has to be dissipated in the statistics of the ciphertext. Therefore once a block encryption is performed, the resulting position and value of each bit in a ciphertext block C are a function of all bits of the corresponding plaintext block P. Suppose that P is composed of two distinct data (P_L and

[15] P_L is named hereafter payload

T), after ciphering, it is impossible to distinguish the P_L ciphered part from the *T* one in *C* (Figure 4-2). Moreover if one bit is modified in *C*, after decryption there is a strong probability that the resulting *T* will be impacted. This probability depends on the size of *T*.

Suppose that *T* is *t*-bit long and that the ciphered block length is *n*-bit. The number of possible plaintext blocks with the same *T* resulting from the decryption of a tampered *C* is equal to 2^{n-t}. Hence the probability *D* that *T* remains the same after decryption is $D = \frac{2^{n-t}}{2^n} = \frac{1}{2^t}$.

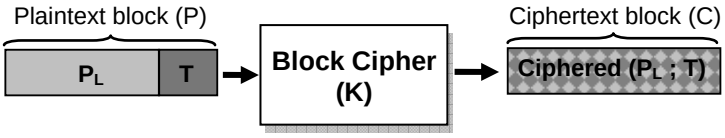


Figure 4-2 The diffusion property of block ciphers

4-2.2. PE-ICE Encryption and Integrity Checking Process

PE-ICE is localized between the last level of cache memory and the memory controller on the SoC (Figure 4-3). Data confidentiality is ensured by block encryption: on write operations data are encrypted before being stored in the external memory and on read operations they are decrypted before being stored in the cache memory. Concerning data authentication, the previous property is used as follows to add the integrity checking capability to block-encryption in PE-ICE.

On write operations (Figure 4-3a), 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. Such a tag must theoretically be a nonce, a Number used ONCE, for a given encryption key and does not need to be calculated over the data with a specific algorithm; for example it may be generated by a counter. After encryption, an indistinguishable and unique pair P_L/T is created and the resulting ciphered block *C* is written in the external memory.

On read operations (Figure 4-3b) *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 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.

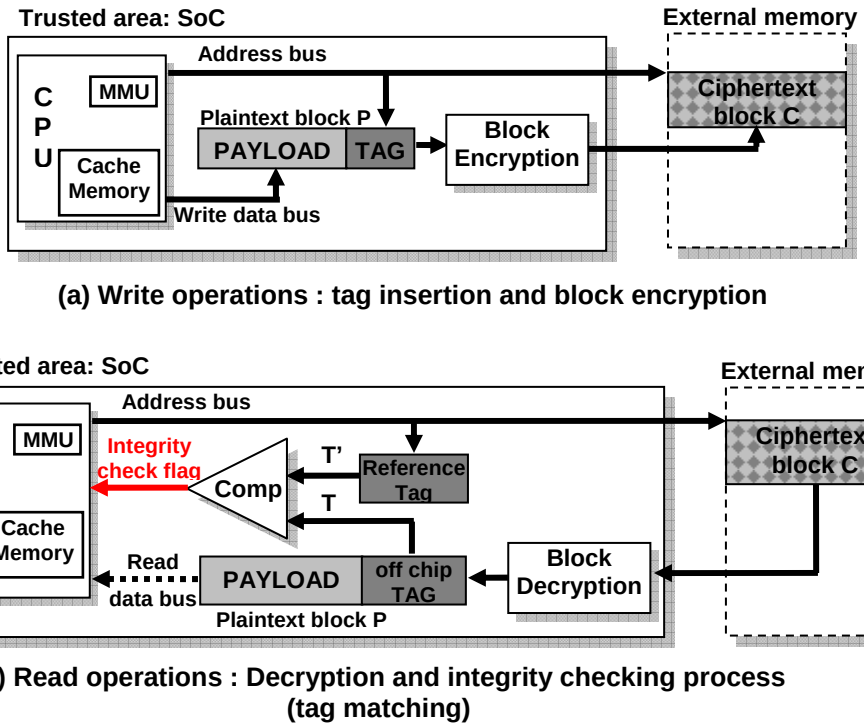


Figure 4-3 PE-ICE Encryption and Integrity checking process

To summarize, PE-ICE could be seen as an AREA (Added Redundancy Explicit Authentication – Chapter 3) scheme at the block level instead of the message level: 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.

The PE-ICE principle presented above gives a countermeasure against spoofing attacks considering that we are able to re-generate a tag reference on read operations which match the tag used on write operations. The next sub-section describes the tag generation and how its composition allows to thwart splicing and replay attacks.

4-2.3. The Tag Generation

In the processor-memory communication context, the SoC alone is achieving 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 perform the integrity checking process. 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 and RW, and depends on their respective properties.

RO data are only written once in external memory and are not modified during 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 P_L/T pair. However, this adversary must not be able to choose the reference tag *T'* or to influence its generation. Hence, PE-ICE uses the most significant bits of the ciphered block address as tag (Figure 4-4a) to respect the nonce requirement. 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 one loaded as tag *T*.

RW data are modified during software execution and are consequently sensitive to replay. Using only the address as tag is not enough to prevent such an attack because the address bits 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 is composed of a vector *RV* (*T* = *RV* - Figure 4-4b) which is changed on each write operation. In the proposed definition of PE-ICE, *RV* is a random value[16] generated on-chip. In this way, the tag is unpredictable[17] from an adversary point of view making this latter unable to know when two ciphered blocks have the same tag. However on read operations PE-ICE must be able to retrieve the correct random values – called in the following the *reference random values RV'* - to generate the reference

[16] In this thesis, we suppose that a random number generator is embedded on-chip and that it does not leak information on the produced value.

[17] Unpredictability is not mandatory, a nonce like a counter is sufficient for *RV*- a random value is not a nonce since a given value can arise twice. If a counter is implemented to generate *RV*, we must assure that the same counter value is not used twice by changing the encryption key and by re-encrypting the corresponding memory section once the counter reaches its limit. Otherwise an adversary can predict when a replay will succeed by waiting that the counter generates the same value. However, re-encryption can be very expensive when the size of *RV* is chosen small by the designer.

tag *T'* for the integrity checking process. On the other hand, the set of *RV'* must be secret and tamper-proof from an adversary point of view; otherwise he could perform a replay when he notices that two blocks are authenticated with the same *RV'* – if known – or he could choose the data to replay by replaying it with the corresponding *RV'* – if non-tamper-proof. In order to solve this issue the random values generated on write operations are stored on-chip[18] as reference random values in a dedicated memory. Thus, they are trusted since the SoC is trusted. Such a tag also protects against splicing attacks. However, instead of making this attack impossible – as it is the case for RO data with the use of the address in the tag – the security relies, as for replay, on the difficulty for an adversary to find two blocks processed by PE-ICE which the reference random values match. The probability to overcome this difficulty is the same as for replay attacks and is defined in section 4-2.1.

The size of *RV* fixes a trade-off between the strength of the countermeasure against replay and the on-chip memory overhead (*RV'* stored on-chip); that is why a second configuration of the tag is proposed for RW data where the most significant bits of the address of each ciphered block are concatenated with a *RV* (*T = RV || ADD* - Figure 4-4c). Such a configuration decreases the strength against replay but maintains a countermeasure against splicing and reduces the on-chip memory cost.

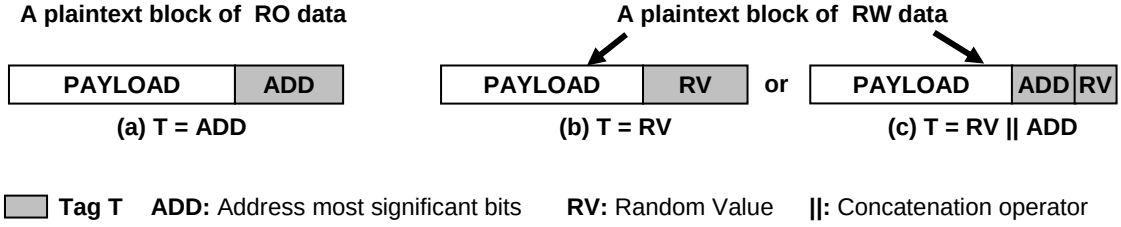


Figure 4-4 Plaintext blocks and tag composition before encryption

4-3. Encryption Mode and Chunk Definition

As highlighted in Chapter 2 the definition of the granularity of encryption and of integrity checking – the chunk size – is essential to limit the memory bandwidth pollution on RMW operations. When direct encryption is implemented the smaller size of the chunk for encryption is the ciphered block length considering that the encryption is done in ECB. Thus,

[18] This assumption is considered in Chapter 4 and 5. However Chapter 6 proposes a scheme to securely store the random value off-chip and to save on-chip memory.

PE-ICE implements ECB[19] mode. Such a mode also enables *per-block integrity checking* in PE-ICE since the tag contained in each plaintext block is generated independently for each plaintext block. This is the finest granularity of integrity checking allowed by PE-ICE; in this way, when a RMW operation is required only the matching ciphered block is loaded to be decrypted and checked. Hence, in the following, the *chunk* size is defined as the length of a block processed by the block cipher implemented in PE-ICE

4-4. Protecting the Physical Address Space vs. the Virtual Address Space

In works like XOM [39], 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 in 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 it because from its point of view the data is fetched from the correct address.

The main advantage of using the virtual address comes from the fact that the application can be loaded in memory already encrypted off-line by PE-ICE (following the principle described in Chapter 2, section 2-3.1).

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 where the virtual address is usually not available. Therefore, such an implementation could be impracticable when PE-ICE must be interfaced with a processor core for which the source code is not easily obtainable (e.g. ARM). Moreover, for physically-addressed cache memory it will be required to store the virtual address - particularly to be able to write-back dirty cache block 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 [40] or prevents [41, 46] their protection. In previous works [41, 46] this point remains an open question.

[19] In [61, 74, 75, 83, 84, 85] PE-ICE is presented by using CBC mode for RO data. The use of CBC is secure and does not induce additional performance overhead when compared to ECB mode. This is because only the decryption process is required for RO data during run-time and such a process can be done in parallel. However, it does not strengthen the encryption in comparison to ECB when implemented in PE-ICE. Indeed, the tag (nonce) generated for RO data ensures that the same plaintext block does not occur twice (see section 5.2). Therefore, for the sake of clarity, in this thesis we consider that all data are encrypted in ECB.

Nevertheless, considering our threat model (OS trusted), an adversary could 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 is the need to decrypt and re-encrypt the code when it is received already ciphered and when a MMU is implemented. Indeed, the physical address is unknown until the translation of the address when the application is loaded in the off-chip memory. Hence, the program must be first decrypted to be able to insert the physical address in plaintext blocks and then re-encrypted. However, the cost of such a computation is negligible since it occurs once at the application load-time. Else, in case of no MMU and of static mapping of programs in the memory, the programmer might use the logical/physical address to encrypt the software with PE-ICE prior to installing it in memory. The latter case does not introduce performance overhead at load-time since the encryption of the executable binary is done off-line.

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 4-5.3).

In the following when the term *address* is used alone we refer to the physical address.

4-5. Security Considerations

4-5.1. Active Attacks

The objective of active attacks is to challenge both data confidentiality and integrity. The security level of the proposed scheme against the three active attacks (spoofing, splicing and replay) described in the threat model and lead on a chunk depends on four parameters:

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

Table 4-1 Security limitations of PE-ICE regarding active attacks led on a chunk evaluated in chances to succeed

Attack		RO data	RW data	
			$t = a + r$	$t = r$
Spoofing attack		$\frac{1}{2^t}$	$\frac{1}{2^t}$	$\frac{1}{2^t}$
Splicing attack Splicing-segment size: $2^a \times b$ bytes	Inside a splicing segment	0	0	$\frac{1}{2^t}$
	Outside a splicing segment	0	$\frac{1}{2^r}$	
Replay attack		N/A	$\frac{1}{2^r}$	$\frac{1}{2^t}$

The probability for an adversary to succeed with a spoofing attack depends only on *t*. Such a probability is defined in section 4-2.1 for spoofing and is equal to $\frac{1}{2^t}$.

Concerning replay, the outputs of the random number generator implemented to produce *RV* are supposed equiprobable; therefore the probability to succeed a replay is equal to $\frac{1}{2^r}$ and is the same for a splicing attack when $t = r$ (the tag *T* is only composed of a random value *RV* – Figure 4.4c) .

When the address is used in the tag, the physical addressing space protected against splicing attacks is determined by *a* and by *b*. It is equal to $2^a \times b$ and is called in the following *splicing-segment*. However, *a* might have a size which could be insufficient to cover the whole address space[20]; hence a different key must be attributed to each splicing segment contained in the application addressing space. The key is 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 not be performed with the correct 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

[20] 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).

attack on a chunk of RO data while for RW data a splicing attack led on a chunk from a splicing segment to another has $\frac{1}{2^r}$ chance to succeed and 0 inside a splicing-segment.

Table 4-1 summarizes the security limitations of PE-ICE regarding the attacks presented in the threat model for each kind of data RW and RO.

4-5.2. Confidentiality and Passive Attacks

An interrogation concerning security might arise when considering PE-ICE: from the confidentiality point of view, is the encryption robustness affected by inserting in the plaintext blocks data potentially known by the adversary (the data address)?

Considering our threat model (SoC trusted), the adversary might only perform two passive attacks, *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 – to challenge data confidentiality. 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, inserting a tag that is a true nonce in each plaintext block removes the ECB weakness when the same plaintext block is ciphered. Indeed, theoretically the tag is a number used once and as a consequence is different for each plaintext block encrypted with a given encryption key, making each plaintext block different from the others. Therefore this ensures that the encryption of the same payload twice yields two different ciphertext blocks in ECB. However, practically this is only true for RO data. The use of a random value to thwart replay attack implies that for RW data the tag is not a true nonce since the same random value can occur twice. Hence, such a random value strengthens the robustness of ECB modes by ensuring that most of the time a same payload leads to different ciphertexts: statistically there is little probability that the same plaintext block (payload + tag) should appear twice during application execution. This probability depends on the size of *r* and on the way the payload is

generated. In such a case, the eavesdropper might only deduce that a same data has been written at a given address and at two different times[21].

4-5.3. PE-ICE 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 requirement.

Different executables 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 executables. Thus we need to avoid replay attacks of RO data from an application to another. Therefore to detect 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 foreseen for RW data and is valid for the different kinds of replay 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. The key management is detailed in a use case proposed for PE-ICE in Chapter 5.

[21] For RO data the tag is a true nonce in a given splicing segment; hence if the encryption mode used for RO data is CBC [61, 74, 75, 83, 84, 85] this tag allows to differentiate the first plaintext block of every CBC chains avoiding the use of an IV.

We consider that an application is always stored at the same address when no MMU are implemented[22]; hence the encryption keys only need to be splicing-segment-dependent and application-dependent[23]. In this case, the application might be encrypted off-line with PE-ICE and stored as such in the off-chip memory if the set of encryption keys used respects the two latter requirements.

Concerning the shared libraries, the solution proposed in section 4-4 must be carefully implemented. In order to observe 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.

4-6. Physical Address Computation

PE-ICE shifts the physical addressing by inserting tags between payloads. This shift must be transparent for the CPU, thus PE-ICE handles the translation.

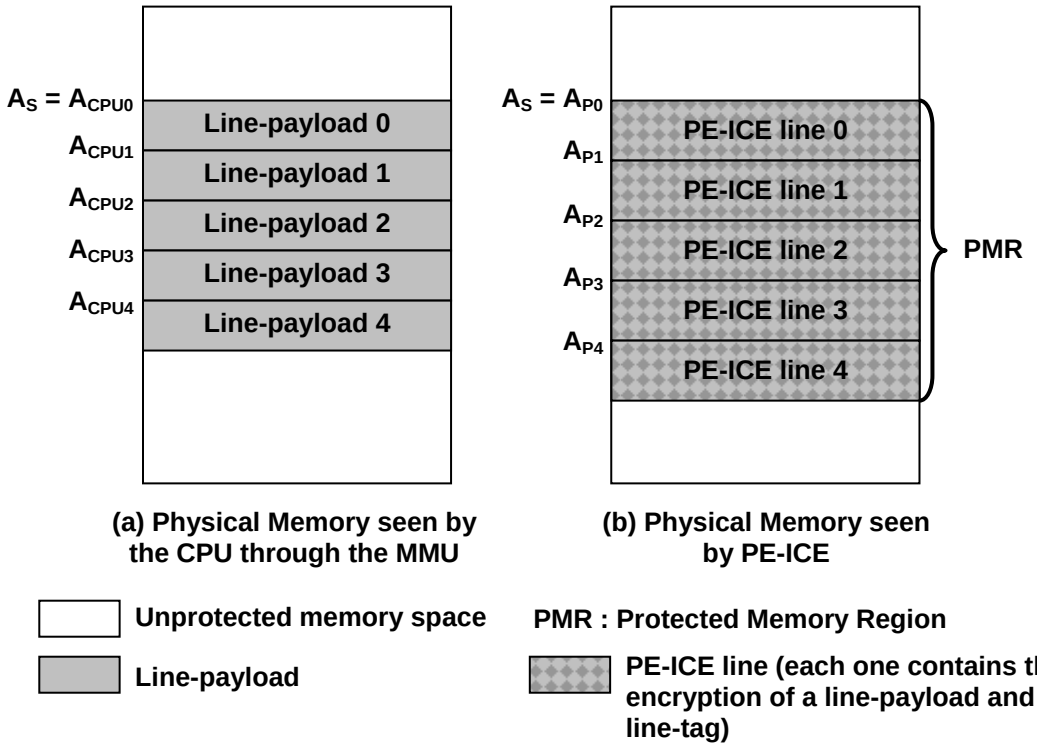


Figure 4-5 Off-chip Memory layout - Reorganization of the Protected Memory Regions (PMR) in PE-ICE line by PE-ICE and shifting of the physical address. Example depicted is a PMR containing five PE-ICE lines (⇔five line-payloads seen by the CPU)

[22] Operating Systems such as µclinux [70] provide the capability to dynamically store applications in memory.
[23] The same remark applies if the virtual address is used in the tag.

To allow a straightforward computation of addresses PE-ICE reorganizes the Protected Memory Regions (PMR) in sections called PE-ICE blocks or PE-ICE lines (Figure 4-5) in the same way as the CPU organizes the off-chip memory in cache block. A PE-ICE line contains a certain number N_c of chunks (⇔ciphered blocks). In the following, a *line-tag* and a *line-payload* denote respectively the amount of payload and tag contained in a PE-ICE line. All PE-ICE blocks have the same layout defined by the distribution of the line-tag and of the line-payload over the *Nc* chunks.

In order to retrieve an address A_P of a PE-ICE block from the address A_{CPU} provided by the CPU, PE-ICE needs to hold in a register the starting address A_S of the protected memory region (PMR). Then, the number P[24] of line-payloads between A_S and A_{CPU} in the address space seen by the CPU is determined by:

$$P = \frac{(A_{CPU} - A_S)}{l_{pb}} \qquad \text{(eq. 4.1)}$$

l_{pb} is the size of a line-payload in bytes. Hence, l_{pb} must be a power of 2 to allow a simple computation of P in hardware.

Considering l_{tb} the size of the line-tag in bytes, A_P can be computed as follows:

$$A_P = A_{CPU} + P \times l_{tb} \qquad \text{(eq. 4.2)}$$

Similarly, l_{tb} must be a power of 2.

In brief, a PE-ICE block is defined by the number *Nc* of chunks (or ciphered blocks) it contains and both the length of its line-tag and line-payload must be a power of 2.

However, when an operation requires accessing data in a line-payload, we must be able to retrieve the matching chunk in the corresponding PE-ICE block. This issue depends on how the line-tag is spread in the chunk(s) contained in a PE-ICE block and on the underlying block cipher implemented; therefore this consideration is dealt with in Chapter 5 which describes the proposed PE-ICE configurations.

In the following A_{PC} denotes the address of a chunk in memory and l_p and l_t respectively the size of a line-payload and of a line-tag expressed in bits.

[24] It can be also expressed as the position of an l_{pb}-byte block starting from A_S in the address space seen by the CPU, where l_{pb} is the size of a line-payload.

4-7. 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 line-tag size (l_t bits) and the length of the corresponding line-payload (l_p bits). R_{OF} is equal to l_t / l_p.

The on-chip memory overhead is defined by the ratio R_{ON} between the bit-length of random value used to protect a PE-ICE block against replay and the corresponding protected payload (the line-payload) size. The bit-length of random value is obtained by multiplying the number of chunks *Nc* contained in a PE-ICE block by the size *r* of a random value; hence R_{ON} is equal to $Nc \times r / l_p$.

4-8. Summary

This section sums up the different definitions introduced in this chapter and lists the useful parameters for the following chapters.

4-8.1. Definitions

Chunk: Atomic block processed by PE-ICE. Its size defines the granularity of encryption and of integrity checking. In PE-ICE the chunk size is of *b*-byte, the length of the block processed by the underlying block cipher.

Tag (*T*): AREA (and added randomness) contained in and independently generated for each chunk on write operations.

Reference tag (*T'*): tag generated by PE-ICE on read operations to perform the integrity checking process.

RV: Random value inserted in the tag dedicated to RW data on write operations.

RV': Reference Random Values retrieved by PE-ICE on read operations to generate the reference tag *T'* for RW data.

$\boldsymbol{P_L}$: Payload contained in a chunk.

PE-ICE block or line: Atomic block size of the PE-ICE memory reorganization. Such a block is defined by the number *Nc* of chunks it contains. All the PE-ICE blocks have the same pattern of tag and payload distribution over the *Nc* chunks.

Line-payload: Bit-length of payload contained in a PE-ICE line - must be a power of 2.

Line-tag: Bit-length of tag contained in a PE-ICE line - must be a power of 2.

PMR (Protected Memory Region): Physical memory region protected by PE-ICE. Such a region is organized in PE-ICE line.

Encryption key requirements for RO data:

Application-dependent: The key must be different for each application loaded in memory.

Installation-dependent: The key must be changed on each installation (⇔ loading in memory) of a given application.

Splicing-segment-dependent: The key must be different for each splicing-segment used by a given application.

4-8.2. PE-ICE Parameters

Table 4-2 summarizes the parameters used to define a PE-ICE configuration.

Table 4-2 Summary of the parameters defining PE-ICE

Parameter	Unit	Definition
b n	byte bit	Block size of the underlying block cipher and chunk length
t	bit	Size of a chunk tag
a	bit	Number of address bits contained in a chunk tag
r	bit	Size of the random value (RV) contained in a chunk tag for RW data and of its corresponding reference random value (RV')
Nc	-	Number of chunks contained in a PE-ICE block
$l_t = 2^c$	bit	Size of a line-tag
$l_p = 2^l$	bit	Size of a line-payload
A_S	-	Starting address of PMR
A_P	-	Physical address of a PE-ICE line
A_{PC}	-	Physical address of a chunk
R_{OF}	-	Ratio defining the off-chip memory overhead ($R_{OF} = l_t / l_p$)
R_{ON}	-	Ratio defining the on-chip memory overhead ($R_{ON} = Nc \times r / l_p$)

4-8.3. PE-ICE Pseudo Codes

PE-ICE processing at run-time can be summarized for operations on a chunk payload P_L of RO data by the functions `PE-ICE-read-ro` and `PE-ICE-write-ro` described in pseudo code in Figure 4-6.

- **Computation of the chunk address A_{PC}**
  `ShiftAdd`(A_{CPU})
  1. A_P ← $[(A_{CPU} - A_S) >> (l - c)] + A_{CPU}$
  2. A_{PC} ← `SpecConfig`(A_P, A_{CPU})
  3. Return A_{PC}

-- $l_t = 2^c$ and $l_p = 2^l$
-- `SpecConfig` is a specific function defined with PE-ICE configurations to retrieve A_{PC} from A_P and A_{CPU}

- **Read of a chunk of RO data**
  `PE-ICE-read-ro` (P_L, A_{CPU})
  1. A_{PC} ← `ShiftAdd`(A_{CPU})
  2. Read the *chunk* from A_{PC} in memory
  3. Decrypt (*chunk*) = P_L || T
  4. T' ← `MostSignAdd`(A_{PC})
  5. Check (T, T')*; if* $T \neq T'$ raise the integrity checking flag else return P_L

-- `MostSignAdd` is a function which selects the *a* most significant bits of the address A_{PC}.

- **Write of a chunk of RO data**
  `PE-ICE-write-ro` (P_L, A_{CPU})
  1. A_{PC} ← `ShiftAdd`(A_{CPU})
  2. T ← `MostSignAdd`(A_{PC})
  3. *chunk* = Encrypt(P_L, T)
  4. Write *chunk* at A_{PC} in memory

Figure 4-6 PE-ICE operations on a payload of RO data contained in a chunk

Concerning RW data, the PE-ICE operations on data of a size smaller or equal to the payload P_L contained in a chunk can be summarized by the functions `PE-ICE-read-rw`

and `PE-ICE-write-rw` described in pseudo code in Figure 4-7. We only considered the case where the tag is only composed of a random value ($t = r$). $|x|$ denotes the length of x.

- **Computation of the chunk address A_{PC}**
  `ShiftAdd`(A_{CPU})
  2. $A_P \leftarrow [(A_{CPU} - A_S) >> (l - c)] + A_{CPU}$
  3. $A_{PC} \leftarrow$ `SpecConfig`(A_P, A_{CPU})
  4. Return A_{PC}

-- $l_t = 2^c$ and $l_p = 2^l$
-- `SpecConfig` is a specific function defined with PE-ICE configurations to retrieve A_{PC} from A_P and A_{CPU}

- **Read of a chunk of RW data**
  `PE-ICE-read-rw` (*data*, A_{CPU})
  1. $A_{PC} \leftarrow$ `ShiftAdd`(A_{CPU})
  2. Read *chunk* from A_{PC} in memory
  3. Decrypt (*chunk*) = $P_L \,||\, T$
  4. $T' \leftarrow$ `ReadOnChipMem`(A_{PC}) (= RV')
  5. Check (T, T') ; if $T \neq T'$ raise the integrity checking flag
     else if $|data| < |P_L|$, retrieve data from P_L
     else return P_L

- **Write of a chunk of RW data**
  `PE-ICE-write-rw` (*data*, A_{CPU})

-- RMW operation
  1. if $|data| < |P_L|$, $P_L \leftarrow$ *PE-ICE-read-rw*(P_L, A_{CPU}) and write *data* in P_L

-- $|data| = |P_L|$
     else $A_{PC} \leftarrow$ `ShiftAdd`(A_{CPU})
  2. Encrypt($P_L \,||\, T$)
  3. $T \leftarrow RV$ = `RandomGen` -- Generate a new random value
  4. $RV' \leftarrow$ `WriteOnChipMem`(A_{PC}, RV) -- Update RV' in the on-chip memory with RV
  5. *chunk* $\leftarrow$ Encrypt(P_L, T)
  6. Write *chunk* at A_{PC} in memory

-- `RandomGen` is a random generation number call which allows generating RV.
-- `WriteOnChipMem` and `ReadOnChipMem` are the functions which allow to read RV' and to write RV in a dedicated on-chip memory.

Figure 4-7 PE-ICE operations on a RW data of a size smaller or equal to the payload contained in a chunk

4-9. Conclusion

In this chapter, we have explored new directions to ensure data confidentiality and authenticity. We have presented the formal definition of PE-ICE. PE-ICE is not a newly proved authenticated mode but a dedicated solution to protect processor-memory communications whose security relies on our threat model (SoC trusted and, ciphertext-only and known plaintext attacks) and on the underlying block cipher security.

We showed that PE-ICE provides data confidentiality and authentication by performing only one path on the data and by using a single block cipher algorithm. Data confidentiality is guaranteed by block encryption while data authentication relies on the diffusion property of block-encryption algorithms and by applying the principle of AREA at the block level. Added redundancy before encryption enables to thwart splicing and spoofing attacks while added randomness handles the issue of replay attacks.

Moreover, we introduced the notion of fine granularity of integrity checking which allows to decrease the memory bandwidth pollution on RMW operations. Thus, PE-ICE proposes a fine chunk size independent of the cache line length while offering a high security level regarding the threat model defined in Chapter 2.

Finally, the use of the same algorithms to ensure data confidentiality and authenticity allows to reduce the cost of hardware resources.

Chapter 5: PE-ICE Implementation

In this Chapter we describe and evaluate different PE-ICE configurations, and we discuss their implementations in a SoC. Note that we only consider 32-bit processor architecture.

This Chapter is organized as follows. Section 5-1 describes three PE-ICE configurations based on different versions of the Rijndael algorithm. Section 5-2 presents the hardware design of one PE-ICE configuration and gives the latencies introduced by all considered PE-ICE configurations. Section 5-3 evaluates PE-ICE performance at run-time. Finally in section 5-4 a comparison with a generic composition scheme is proposed.

5-1. PE-ICE Configurations

A configuration of PE-ICE depends on the *n*-bit block length of the underlying block cipher and is denoted PE-ICE-*n*. Moreover, a configuration is defined by the layout of a PE-ICE line, meaning the pattern of the line-tag and of the line-payload distribution over the chunks it contains.

In this section three PE-ICE configurations based on different versions of the Rijndael algorithm are described and solutions are proposed to handle the issue of encryption key managements.

5-1.1. PE-ICE-128

PE-ICE-128 is the proposed configuration for the AES block cipher to be used in PE-ICE. AES encryption processes 128-bit blocks (described in Chapter 1) and is the standardized version of the Rijndael algorithm [15].

5-1.1.1. Layout of a PE-ICE-128 Line

As explained in Chapter 4, in order to allow a straightforward computation of a PE-ICE line address, the line-payload and the line-tag lengths must be both a power of 2. Moreover encryption forces a PE-ICE-128 block to be a multiple of the ciphered block length. As a consequence, choosing a line-payload and a line-tag both of 64-bit or of 128-bit fulfills all requirements but induces a huge off-chip memory overhead of 100%. Thus, to reduce this overhead we define the line-payload and the line-tag length respectively to 256-bit and 128-bit. The resulting PE-ICE-128 is therefore composed of three chunks and is defined by the following parameters: l_p = 256-bit, l_t = 128-bit, and Nc = 3. The line-tag distribution among plaintext blocks (chunks) composing a PE-ICE-128 block is depicted in Figure 5-1.

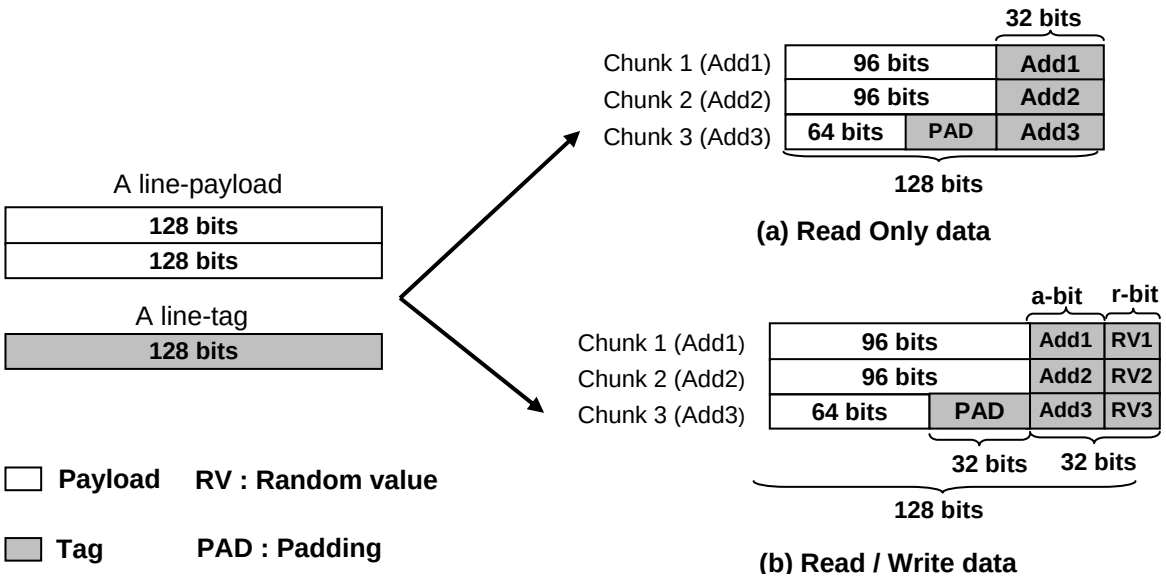


Figure 5-1 Layout of a PE-ICE-128 line before encryption

The tag of the third chunk in a PE-ICE-128 line contains 32-bit of padding (PAD section) to make it 128-bit (ciphered block size). This choice could appear as a wasting of bits which could be useful to strengthen the security of PE-ICE; for instance, these bits can be shared

among the three chunk tags of a PE-ICE-128 block as presented in [74, 75]. However, this implies that the size of P_L is no more a multiple of 32-bit which makes the corresponding chunk more difficult to address (see 1.1.4). Moreover, such a line-tag repartition may require loading two chunks - instead of one for the proposed layout - on 32-bit read and on RMW operations on 32-bit, leading to additional memory bandwidth pollution.

5-1.1.2. Security Limitations

Concerning RO data, the 32-bit address of each ciphered block is used as tag (Figure 5-1a). 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 (Figure 5-1b), the strength of the proposed countermeasure against replay and splicing attacks depends on the designer's choice of the values of r and a. For example if a = 24 and r = 8, a splicing segment is 256 MB long (2^{24} 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}.

Table 5-1 Security limitations offered by PE-ICE-128 and by PE-ICE-160 regarding active attacks led on a chunk and evaluated in chances to succeed for an adversary

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

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 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. However a scheme is proposed in Chapter 6 allowing to choose a large value for *r* without generating on-chip memory overhead.

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 5-1 summarizes the security limitations offered by PE-ICE-128.

5-1.1.3. Memory Consumption

The off-chip memory overhead of PE-ICE-128 is of 50% (l_t = 128 and l_p = 256 ➔ R_{OF} = ½) therefore the tag storage takes up to 33% of the protected memory region (PMR).

The on-chip memory overhead is of 9.4% of the off-chip memory protected against replay by considering *r* = 8 (*Nc* = 3, *r* = 8, l_p = 256 ➔ $R_{ON} = 3/32$).

5-1.1.4. Computation of a Chunk Physical Address

The address of a PE-ICE-128 block A_p is obtained by applying to A_{CPU} the equation provided in Chapter 4 (eq. 4.1):

$$A_P = A_{CPU} + \frac{(A_{CPU} - A_S)}{2} \qquad \text{(eq. 5.1)}$$

However, one of the objectives of PE-ICE is to load the minimum of data on small transfers. Hence, PE-ICE-128 must be able to retrieve the address A_{PC} of the chunk in a given PE-ICE-128 block which contains the requested data. To solve this issue, the less significant bits of A_{CPU} (A_{CPU}[4;0]) addressing the data in the payload are used with an address decoder (AD). The output of AD is an offset O_i (0, 16 bytes, 32 bytes) to add to A_P and to determine the matching chunk address.

$$A_{PC} = A_P + O_i \qquad \text{(eq. 5.2)}$$

5-1.2. PE-ICE-160

The Rijndael algorithms support any key and block size that is a multiple of 32, between 128 and 256. For PE-ICE-160, we use Rijndael processing 160-bit blocks and using a 128-bit key. This version of the algorithm is referred to as Rijn-160.

5-1.2.1. Layout of a PE-ICE-160 Line

The definition of a PE-ICE-160 line is easier with such a block cipher. Indeed, the biggest power of 2 contained in 160 is 128, hence we choose a line-payload of such a size and a line-tag of 32-bit. All requirements are fulfilled: the PE-ICE-160 line is a multiple of the ciphered block length (Nc = 1); l_p and l_t are both a power of 2. The resulting PE-ICE-160 line is depicted in Figure 5-2. Note that for PE-ICE-160, a PE-ICE-160 line, a line-payload and a line-tag are respectively equivalent to a chunk, P_L and a chunk tag (Nc = 1, $|P_L| = l_p$, $t = l_t$).

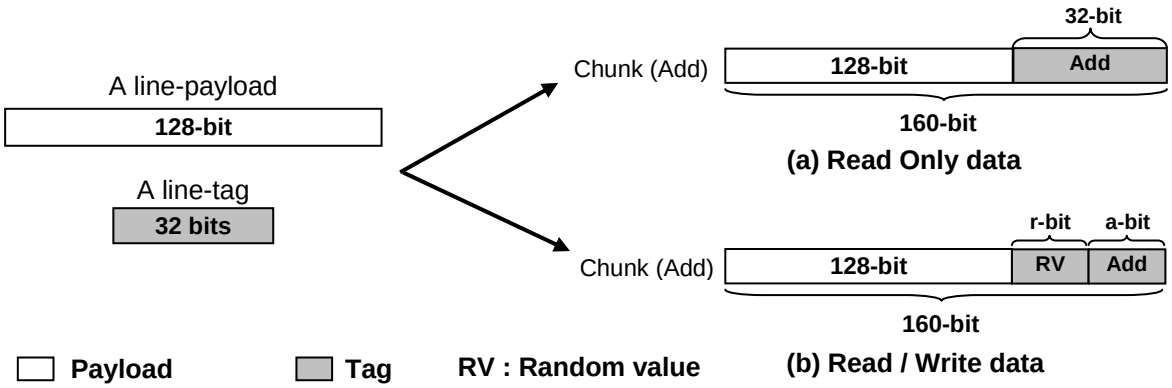


Figure 5-2 Layout of a PE-ICE-160 line before encryption

5-1.2.2. Security Limitations

PE-ICE-160 has exactly the same security limitations as PE-ICE-128 since the size of the tag dedicated to a chunk is the same (Table 5-1).

5-1.2.3. Memory Consumption

The off-chip memory overhead of PE-ICE-160 is of 25% (l_t = 32 and l_p = 128 ➔ R_{OF} = ¼) therefore the tag storage takes up to 20% of the protected memory region (PMR).

The on-chip memory overhead is 6.25% of the off-chip memory protected against replay by considering r = 8 (Nc = 1, r = 8, l_p = 128 ➔ $R_{ON} = 1/16$).

5-1.2.4. Computation of a Chunk Physical Address

An advantage of PE-ICE-160 is the straightforward computation of the address A_{PC} of a chunk. Indeed, as mentioned above a PE-ICE-160 block is equivalent to a chunk, hence A_{PC} =

A_P and is obtained by simply applying the equation (eq. 4.1) provided in Chapter 4 to the address A_{CPU} provided by the CPU:

$$A_{PC} = A_{CPU} + \frac{(A_{CPU} - A_S)}{4} \qquad \text{(eq. 5.3)}$$

5-1.3. PE-ICE-192

PE-ICE-192 is the PE-ICE configuration with the Rijndael algorithm processing 192-bit block and using 128-bit keys. Such a Rijndael version is referred to as Rijn-192.

5-1.3.1. Layout of a PE-ICE-192 Line

Similarly to PE-ICE-160 the bigger power of 2 contained in 192 is 128, hence we choose a line-payload of such a size and a line-tag of 64-bit. The resulting PE-ICE-192 line is depicted Figure 5-3. Note that for PE-ICE-192, a PE-ICE-192 line, a line-payload and a line-tag are also equivalent respectively to a chunk, P_L and a tag ($Nc = 1$, $|P_L| = l_p$, $t = l_t$).

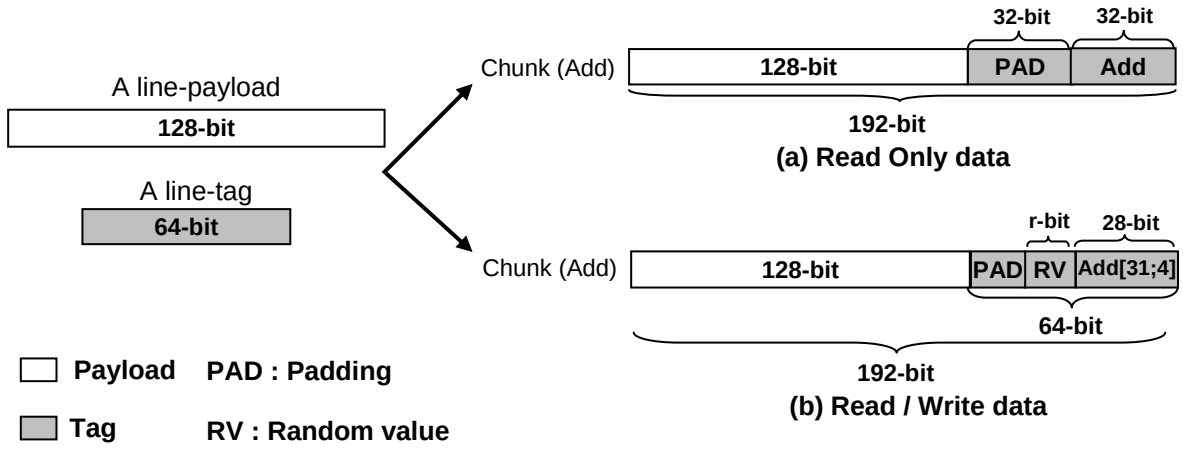


Figure 5-3 Layout of a PE-ICE-192 line before encryption

5-1.3.2. Security Limitations

Concerning RO data, a chunk tag T contains the 32-bit of the chunk address (Figure 5-3a), thus it protects the whole address space (4GB) against splicing attacks ($a = 32$) and only one encryption key dedicated to RO data is required per application during run-time.

The particularity of PE-ICE-192 is the fact that (Figure 5-3b) the size of the chunk tag ($t = 64$) prevents splicing attacks on the whole addressing space dedicated to RW data ($a = 28$)

while offering a large range of choice for the value of *r*. Indeed, *r* can be up to 36 without decreasing the strength of the countermeasure against splicing. The chance to succeed with a replay depends on the value of *r* and is equal to $\frac{1}{2^r}$. One key is also required for RW data but for all applications.

For both kinds of data the chunk tag is 64-bit long, hence an adversary has a $1/2^{64}$ chance to succeed with a spoofing attack.

Table 5-2 summarizes the security limitations of such a configuration of PE-ICE-192.

Table 5-2 Security limitations offered by PE-ICE-192 regarding active attacks led on a chunk and evaluated in chances to succeed for an adversary

Attack		RO data	RW data
			$t = a + r$
Spoofing attack		$\frac{1}{2^{64}}$	$\frac{1}{2^{64}}$
Splicing attack	Inside the source splicing segment	0	0
	Outside the source splicing segment	N/A	N/A
Replay attack		N/A	$\frac{1}{2^r}$

5-1.3.3. Memory Consumption

The off-chip memory overhead of PE-ICE-192 is 50% (l_t = 64 and l_p = 128 ➔ R_{OF} = ½) therefore the tag storage takes up to 33% of the protected memory region (PMR).

The on-chip memory overhead is 6.25% of the off-chip memory protected against replay by considering *r* = 8 (*Nc* = 1, *r* = 8, l_p = 128 ➔ $R_{ON} = 1/16$).

5-1.3.4. Computation of a Chunk Physical Address

Like PE-ICE-160, the computation of the address A_{PC} of a chunk is straightforward since a chunk is equivalent to a PE-ICE-192 line; hence $A_{PC} = A_P$:

$$A_{PC} = A_{CPU} + \frac{(A_{CPU} - A_S)}{2} \qquad \text{(eq. 5.4)}$$

At this stage PE-ICE-160 and PE-ICE-192 seem to be the most straightforward PE-ICE implementations. While the latter offers a high level of security by preventing splicing attacks for all kinds of data on the whole addressing space the former proposes an optimum trade-off between security and off-chip memory overhead with strong countermeasures against splicing and replay when $t = r$.

5-2. Hardware Design and Latencies

AMBA [76] (Advanced Microcontroller Bus Architecture) are the most common on-chip buses implemented in embedded processors. Thus, PE-ICE-128 has been designed compliant with an AMBA bus in order to be easily portable on those processors and to evaluate the latencies introduced on memory accesses.

5-2.1. The AMBA-AHB Bus

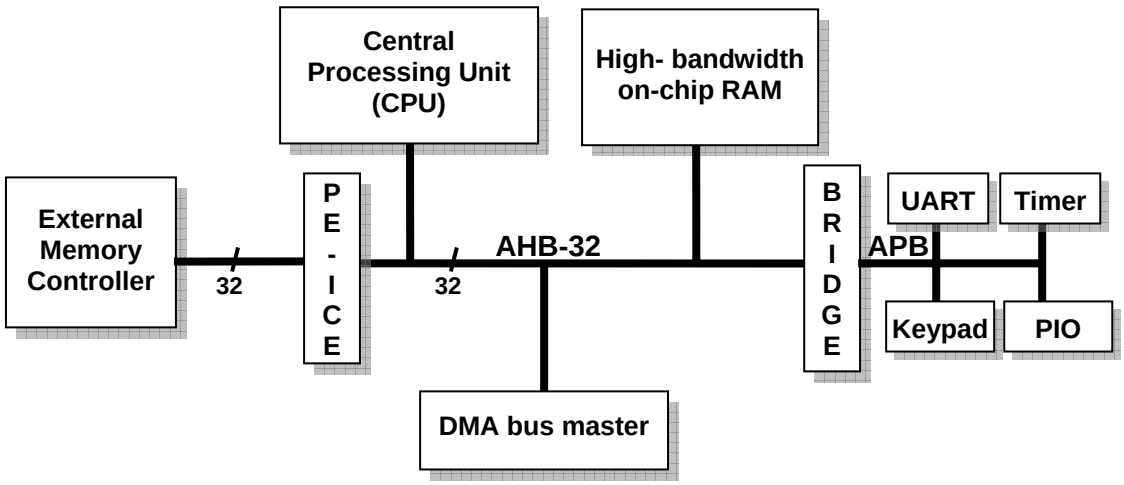


Figure 5-4 PE-ICE-128 localization on an AMBA-AHB bus

AMBA is the definition proposed by ARM of on-chip communication standards for embedded processors. Mainly two standards are described in the AMBA specification: AHB (Advanced High-perforance Bus) and APB (Advanced Peripheral Bus). APB is dedicated to low-power peripheral accesses. AHB is a bus which supports processor connections to on-chip memories and to off-chip memory interfaces. It is foreseen for high performance and

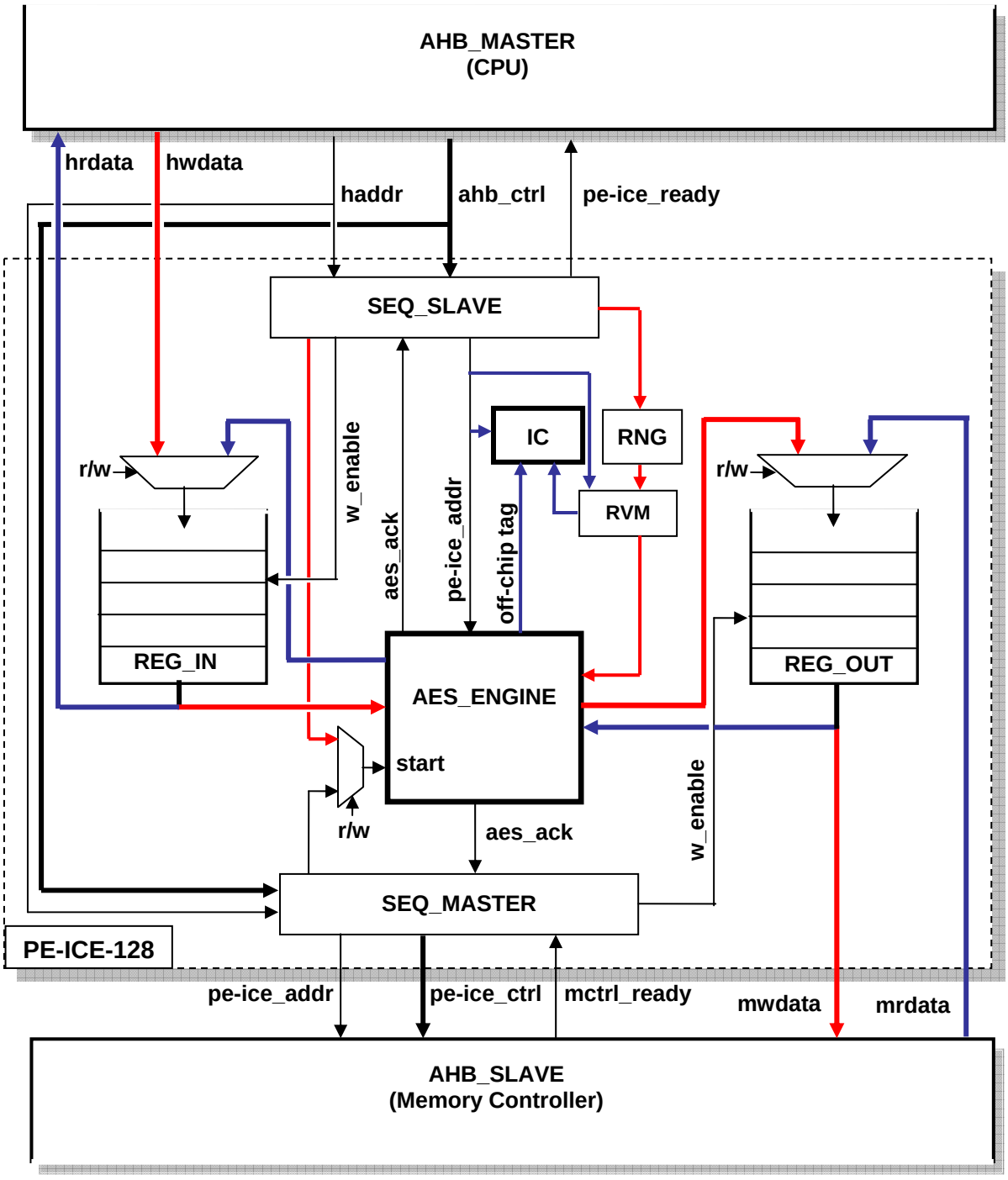
high clock frequency system modules. One of the specificities of AHB is the implementation of burst transfers.

In the present work, we are particularly interested in AMBA-AHB since data exchanged with the off-chip memory transit on this bus before leaving the SoC – write operations - or before being processed by the CPU – read operations. PE-ICE-128 is therefore designed as an interface between the off-chip memory controller and the rest of the AMBA-AHB bus (Figure 5-4). The protocol AHB is also used for communication between PE-ICE and the memory controller.

5-2.2. Design Principle

PE-ICE-128 has been designed in VHDL to accurately measure the latencies introduced on memory accesses and to be able to deduce the ones of the other PE-ICE configurations. Figure 5-5 depicts an overview of this hardware design. The description of this figure is detailed per operation:

(1) *Write operations*: The CPU transmits over the AMBA-AHB bus the controls signals – *ahb_ctrl* – which describe the requested operation (Read or Write, size of the transfer, kind of transfer: single or burst…) and the corresponding address – *haddr*. If PE-ICE-128 is free – *pe-ice_ready* – it collects this information with two sequencers – SEQ_SLAVE and SEQ_MASTER. SEQ_SLAVE stores the data to write – *hwdata* – in the register REG_IN, translates the data address and launches a random number generation if the data to write is a RW data in order to generate the tag. When enough payloads are ready in REG_IN to constitute a plaintext block with the tag, SEQ_SLAVE activates AES encryption. SEQ_MASTER also translates the data address – *pe-ice_addr*, waits for the AES_ENGINE acknowledge which indicates that the ciphered data is ready in REG_OUT, and manages the forwarding of the encrypted block to the memory controller – *pe-ice_ctrl*. Data to write in the off-chip memory are sent over the *mwdata* bus.



: Write data pass and write control signals

: Read data pass and read control signals

: Control signals used on read and write operations

IC : Integrity Checking

RNG : Random Number Generator

RVM : Random value memory

Figure 5-5 PE-ICE design principle on an AMBA-AHB bus

(2) *Read operations*: The CPU transmits over the AMBA-AHB bus the control signals and the corresponding address, respectively on *ahb_ctrl* and *haddr*. If PE-ICE-128 is free – *pe-ice_ready* - SEQ_MASTER collects this information and translates the address to obtain the off-chip memory address of the requested data: *pe-ice_addr*. It forwards the read request to the memory controller via *pe-ice_addr* and *pe-ice_ctrl*. It waits for the memory controller's positive response – *mctrl_ready* – and stores the data coming from *mrdata* in REG_OUT. When an AES block is ready in this register, it launches the AES_ENGINE for decryption. At the same time, SEQ_SLAVE also translates the address and sends it to IC (Integrity Checking). If the requested data is RW data, this address also serves to retrieve the random value needed to form the tag in RVM (Random Value Memory). IC generates the tag reference from *pe-ice_addr* and from this random value. When decryption is done, the payload part of the plaintext block is stored in REG_IN and the tag part is transmitted to IC. IC immediately compares it to the tag reference and depending on the comparison results allows or not SEQ_SLAVE to forward the data in REG_IN to the CPU on *hrdata*.

(3) *Read Modify Write operations*: These operations occur for PE-ICE-128 when the size of the data to write does not match the payload contained in one or two chunks. As presented in chapter 2, on such operations it is first required to load the matching encrypted block(s) and to decrypt it – and authenticate it – before starting the write operation. When PE-ICE-128 is configured for 256-bit cache block, such operations are 8 to 32-bit write and 128-bit write. For 8 to 32-bit write only the matching chunk is loaded from the off-chip memory whereas for 128-bit write two consecutive chunks are fetched. Therefore, as for normal write operations SEQ_SLAVE stores data from the CPU in REG_IN but at the same time SEQ_MASTER manages the read and the decryption of the matching chunks. The data to keep in the decrypted block(s) are stored in REG_IN after being checked by IC. SEQ_SLAVE starts re-encryption only after re-generation of a new tag. Finally, SEQ_MASTER writes the encrypted chunk(s) in the off-chip memory.

Two versions of PE-ICE-128 have been designed by considering the principles exposed above. The first one optimizes latencies introduced by PE-ICE-128 by considering all features of the AMBA-AHB bus described in the specification [76]. Its purpose is to evaluate PE-ICE in terms of performance. The second version is dedicated to an implementation on the LEON2 processor [77] and as a consequence takes into account the specific implementation of the

AMBA-AHB bus on this processor. For instance, only single transfers and undefined bursts are available; hence, it is not possible at the beginning of a CPU request to differentiate a 4-word write from an 8-word write or a 4-word read from an 8-word read. Thus this version of PE-ICE-128 implements all kinds of operations described above but always serializes SEQ_SLAVE and SEQ_MASTER jobs to handle this LEON specificity. On all read operations, a whole PE-ICE-128 block is loaded – three AES blocks – and on all write operations the data is first stored in REG_IN and depending on the size of the write, a read is requested or not by SEQ_MASTER to the memory controller.

Both versions of PE-ICE-128 do not include a random number generator. We use fixed numbers instead of random values. However, such a component does not influence performance since the random values could be generated in advance.

5-2.3. Latencies

In this section, we present the additional latencies introduced on off-chip memory accesses by the different PE-ICE configurations and by AES encryption/decryption. The underlying CPU considered in this study is the ARM9E for which the optimum frequency in the 0.18µm CMOS technology is around 200MHz with an AHB bus running at 100MHz [78].

5-2.3.1. PE-ICE-128 Latencies

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 in the 0.18µm CMOS technology presented in [23] shows that such a latency is valid until 330 MHz. Hence, in the following we consider two realistic cases for the ratio $R_{E/B}$ ($= \frac{F_{AES}}{F_{AHB}}$) between the AES frequency (F_{AES}) and the bus frequency (F_{AHB}): $R_{E/B} = 1$ and $R_{E/B} = 2$. When $R_{E/B} = 2$ the intrinsic latency of the AES encryption seen on the AHB bus is of 6 cycles.

Table 5-3 summarizes the additional latencies seen on the AHB bus and introduced by AES encryption in ECB mode – referred to in the following as the AES-ECB engine – and by PE-ICE-128 on all operations requested by an ARM9E processor core during run-time. Figure 5-6 and Figure 5-7 depict how those latencies occur respectively on read and write operations;

for the sake of clarity we do not show RMW operations (8 to 32-bit write and 128-bit write for PE-ICE-128). The latencies seen on the AHB when no security engines are implemented are referred to as the Base latencies[25].

Table 5-3 Additional latencies introduced by PE-ICE-128 and by the AES-ECB engine on an AMBA-AHB bus for the operations requested by an ARM9E core

Operations	AES-ECB		PE-ICE-128			
	Latencies (AHB cycles)		Latencies (AHB cycles)		Overhead vs. AES-ECB	
	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$
8 to 32-bit Write	38	28	38	28	0%	0%
8 to 32-bit Read	15	10	15	10	0%	0%
128-bit Write	15	10	39	29	160%	190%
128-bit Read	15	10	16	11	6.5%	10%
256-bit Write	15	10	18	13	20%	30%
256-bit Read	15	10	17	12	13.5%	20%
512-bit Write	15	10	22	17	46.5%	70%
512-bit Read	15	10	21	16	40%	60%

The latencies on all read operations and on 256-bit write operations of RW data have been verified by RTL simulation of the optimized version of PE-ICE-128 with the Modelsim simulator [79]. An 8 to 32-bit write generates a RMW operation on an AES block for the AES-ECB engine as for PE-ICE-128. A 128-bit write also generates a RMW operation but only for PE-ICE-128 and this is due to the fact that such a payload is contained in two chunks (Figure 5-1). The additional latencies of those two RMW operations are deduced from previous simulation results and from the read memory latencies we have considered: 9 cycles for the off-chip read latency. The choice of such latency is motivated in section 5-3 where run-time performance evaluation is detailed.

Concerning the AES-ECB encryption (respectively decryption), the additional latencies seen on the AHB bus (Figure 5-6 and 5-7), are due to the intrinsic latency of the AES plus the time to collect one 128-bit plaintext (respectively ciphertext) block on the 32-bit bus. Concerning PE-ICE-128, on write operations the overhead compared to the AES-ECB engine comes from the tag encryption and storage; a plaintext block is collected in less than 4 bus cycles (Figure 5-6), increasing the throughput of PE-ICE-128 and generating bus resource

[25] To obtain the additional latencies introduced by the security engines from Figure 5-6 and 5-7: for each operation subtract the Base latency from the ones induced by the security engines. Moreover, on write operations the tag does not appear on Figure 5-6 since it is collected at the same time as the data.

conflicts. On read operations (Figure 5-7), the inverse applies; the overhead is due to the tag which increases the amount of data to load and to decrypt, pollutes the throughput of PE-ICE-128 and creates wait cycles. 32-bit write operation does not generate additional latencies for PE-ICE-128 compared to the AES-ECB engine because this RMW is performed for both on a 128-bit block.

However, the PE-ICE integrity checking process is almost negligible compared to AES-ECB encryption alone (comprised between 0 and 47% for $R_{E/B}$ = 1) except for the 4-word write RMW operation (153% for $R_{E/B}$ = 1). Doubling the ratio $R_{E/B}$ decreases the latencies of the security mechanisms (of 29% on average for PE-ICE-128 and of 32% for AES-ECB) and slightly increases the integrity checking cost of PE-ICE-128.

5-2.3.2. PE-ICE-160 Latencies

The difference between all Rijndael versions lies in the number of rounds required to output a ciphertext block (or a plaintext block). This number of rounds *Nr* is defined in [15] 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, *Nr* is equal to 11. Hence the intrinsic latency of Rijn-160 seen on the AHB bus is of 12 cycles and of 6 cycles respectively for $R_{E/B}$ = 1 and $R_{E/B}$ = 2. Table 5-4 sums up the additional latencies introduced by PE-ICE-160 on the AHB bus (expressed in bus cycles).

Table 5-4 Additional latencies introduced by PE-ICE-160 and by the AES-ECB engine on an AMBA-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	
	$R_{E/B}$ = 1	$R_{E/B}$ = 2	$R_{E/B}$ = 1	$R_{E/B}$ = 2	$R_{E/B}$ = 1	$R_{E/B}$ = 2
8 to 32-bit Write	38	28	42	30	10.5%	7%
8 to 32-bit Read	15	10	17	11	13.5%	10%
4-word Write	15	10	17	11	13.5%	10%
4-word Read	15	10	17	11	13.5%	10%
8-word Write	15	10	18	12	20%	20%
8-word Read	15	10	18	12	20%	20%
16-word Write	15	10	20	14	33.5%	40%
16-word Read	15	10	20	14	33.5%	40%

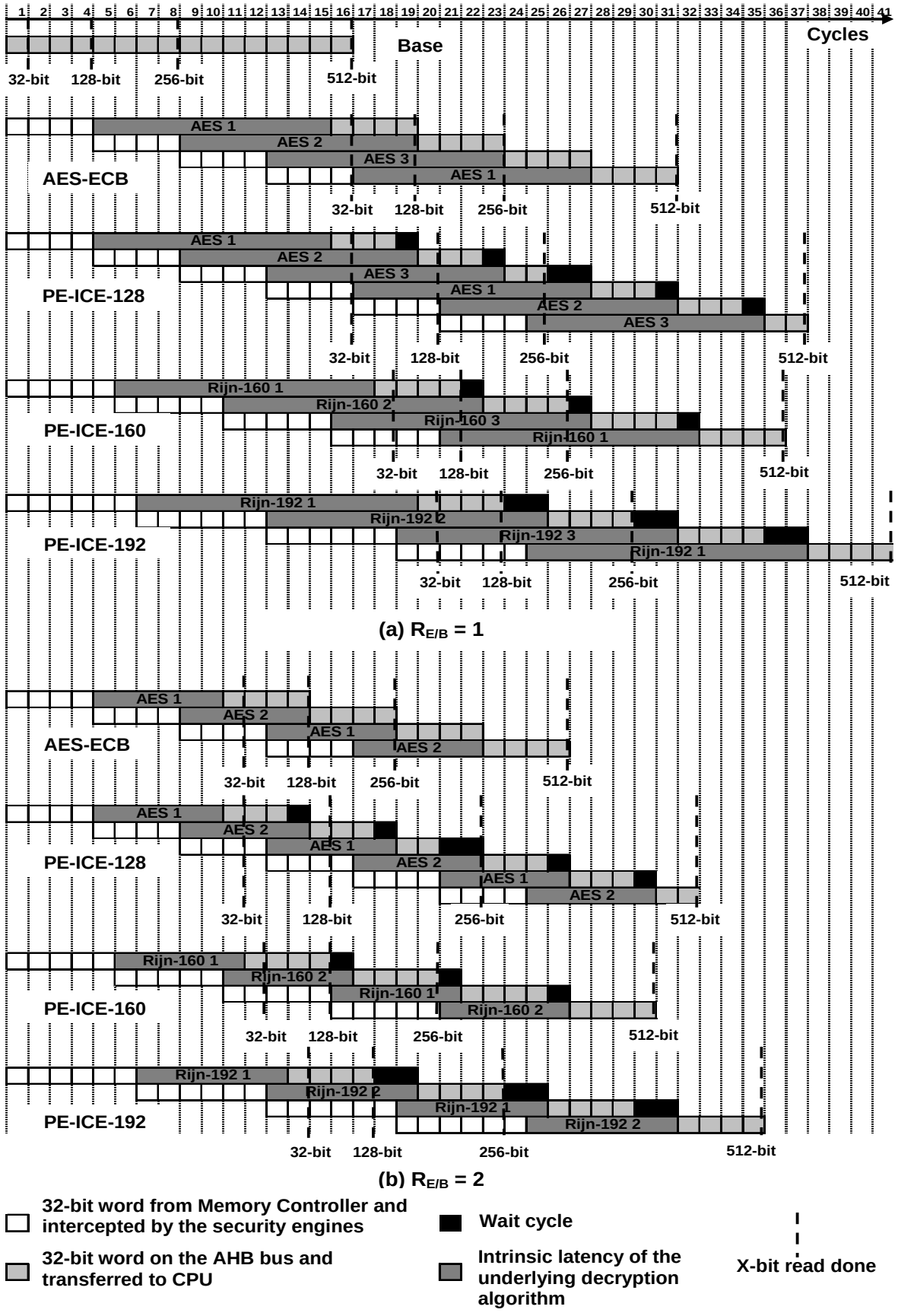


Figure 5-6 Latencies introduced on the AHB bus by the different PE-ICE configurations and by the AES-ECB engine on *read operations*

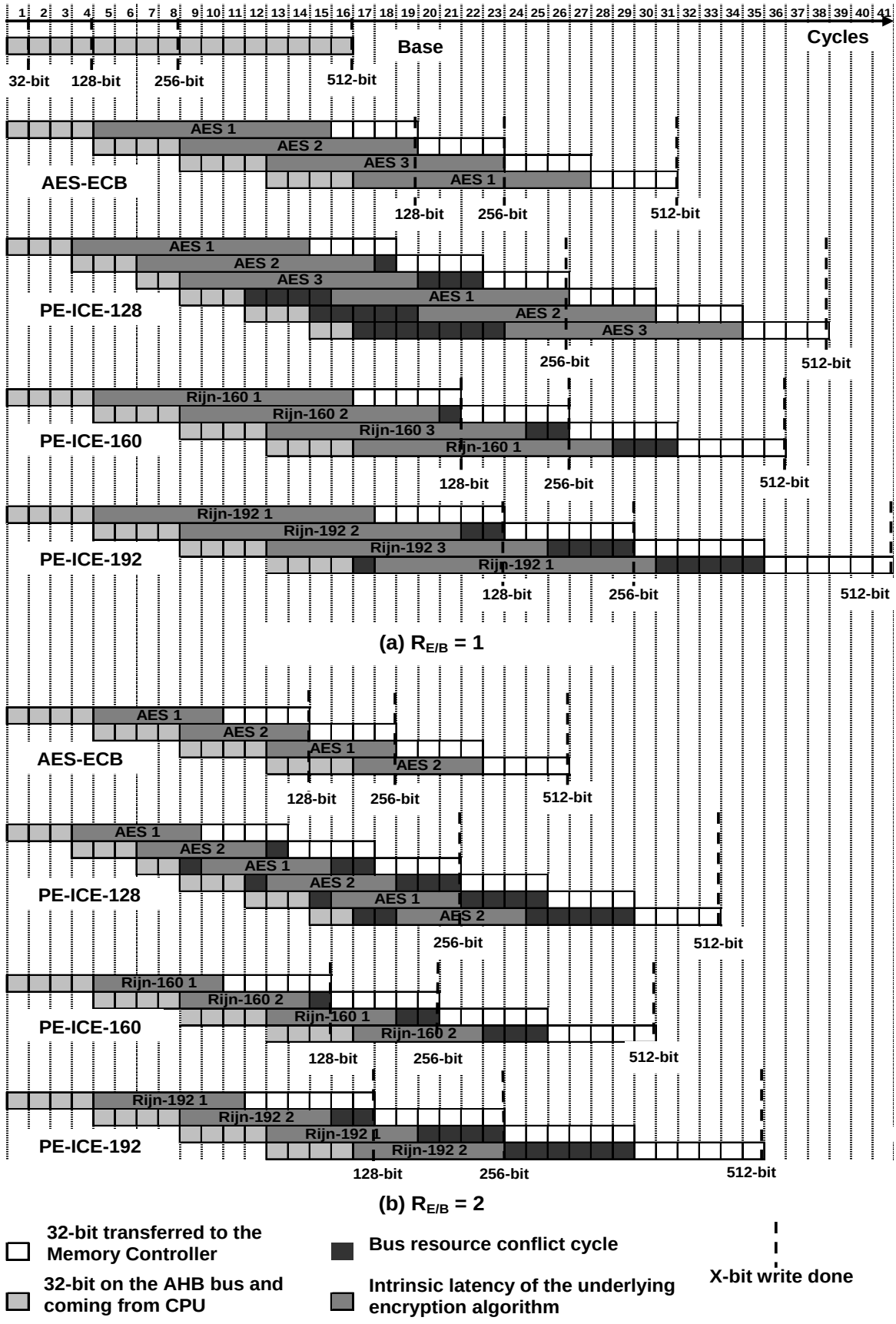


Figure 5-7 Latencies introduced on the AHB bus by the different PE-ICE configurations and by the AES-ECB engine on *write operations*

Compared to the AES-ECB engine the overhead of PE-ICE-160 is partially due to the increase of the intrinsic latency of the underlying block cipher. Moreover, on read operations (Figure 5-6) 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 (Figure 5-7).

The cost of the PE-ICE-160 integrity checking process is less important than the one of PE-ICE-128. On average it is of 19.5% for $R_{E/B} = 1$ and of 22% for $R_{E/B} = 2$ whereas for PE-ICE-128 it is respectively of 34.5% and 46%. This improvement comes from the 4-word write operation which does not generate a RMW for PE-ICE-160 since such a payload is contained in one chunk. Hence, like AES-ECB encryption, the 128-bit payload can be directly ciphered and the matching chunk overwritten in memory. A second reason for this latency reduction is the value of the ratio Payload / Tag which is higher for PE-ICE-160 than for PE-ICE-128 on most of the operations, meaning that the amount of meta-data (tag) to process is lower on average for the same amount of payload.

5-2.3.3. PE-ICE-192 Latencies

For Rijn-192, *Nr* is equal to 12. Hence the intrinsic latency of Rijn-160 seen on the AHB bus is of 13 cycles and of 7 cycles respectively for $R_{E/B} = 1$ and $R_{E/B} = 2$.

Table 5-5 sums up the additional latencies introduced by PE-ICE-160 on the AHB bus (expressed in bus cycles).

Table 5-5 Additional latencies introduced by PE-ICE-192 and by the AES-ECB engine on an AMBA-AHB bus for the operations requested by an ARM9E core

Operations	AES-ECB		PE-ICE-192			
	Latencies (AHB cycles)		Latencies (AHB cycles)		Overhead vs. AES-ECB	
	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$
8 to 32-bit Write	38	28	46	34	21%	21,5%
8 to 32-bit Read	15	10	19	13	26.5%	30%
4-word Write	15	10	19	13	26.5%	30%
4-word Read	15	10	19	13	26.5%	30%
8-word Write	15	10	21	15	40%	50%
8-word Read	15	10	21	15	40%	50%
16-word Write	15	10	25	19	66.5%	90%
16-word Read	15	10	25	19	66.5%	90%

Similarly to PE-ICE-160, the overhead of PE-ICE-192 is due to the increase of the intrinsic latency of the block cipher and to the tag which generates bus resource conflicts on write operations (Figure 5-7) and wait cycles on read operations (Figure 5-6).

Despite the fact that the 4-word write operation does not generate a RMW for PE-ICE-192, the cost of the PE-ICE-192 integrity checking process is slightly higher than the one of PE-ICE-128. On average it is of 39.5% for $R_{E/B} = 1$ and of 49% for $R_{E/B} = 2$ whereas for PE-ICE-128 it is respectively of 34.5% and 46%.

5-2.4. Silicon Area Usage

The hardware implementation of the same round is required for all versions of Rijndael. Therefore, the hardware resources needed for all PE-ICE configurations and for the AES-ECB engine to obtain the latencies listed above can be estimated in numbers of AES cores, N_{AES}, 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 for $R_{E/B} = 1$ (11 cycles) and for $R_{E/B} = 2$ (6 cycles), the AES-ECB engine must implement respectively three and two AES cores to reach the optimum throughput of 32-bit per cycle.

For the PE-ICE configurations the same reasoning can not be followed since the tag pollution makes their throughput per cycle different on read and write operations. For instance for PE-ICE-160, on write operations, 4 cycles are required to collect a plaintext block on the AHB bus and 5 cycles to output a ciphertext block to the memory controller while on read operations, 5 cycles are required to collect a ciphered text block and 4 to output a plaintext block. The maximum throughput theoretically required on write operations is higher than on read operations. However, the available bandwidth between PE-ICE and the memory controller limits the throughput on write operations: on read and on write operations a ciphered text block is processed only every 4 cycles for PE-ICE-128. Hence, considering the intrinsic latencies of the AES when $R_{E/B} = 1$ and when $R_{E/B} = 2$, to reach the optimum throughput N_{AES} must be equal respectively to three and two, as for the AES-ECB engine. By reasoning similarly for PE-ICE-160 and PE-ICE-192, we obtain the same values for N_{AES}.

For an atomic bus transfer the same key is shared by all implemented AES cores, hence only one key expander core is needed for all PE-ICE configurations and for the AES-ECB

engine. By considering the figures provided by Ocean Logic [80] an AES encryption/decryption core with 11 cycles of latency takes 24 Kgates and the corresponding key expander core 32 Kgates in the 0.18µ technology. This means that, like the AES-ECB engine, the hardware cost of all PE-ICE configurations can be approximated to 104 Kgates and 80 Kgates respectively for $R_{E/B} = 1$ and for $R_{E/B} = 2$.

5-2.5. Latency Optimization

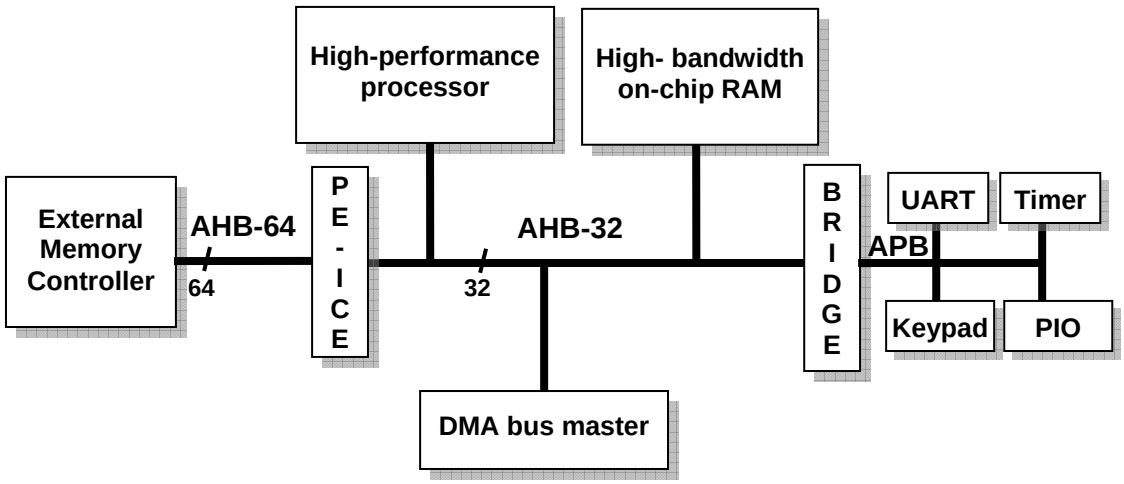


Figure 5-8 Architecture using 64-bit processor memory bus

In previous works [12, 13, 37, 41, 46, 47] the off-chip memory bandwidth is increased by using a 64-bit processor-memory bus. Hence to achieve the same objective, we propose an alternative implementation where PE-ICE is interfaced with the memory controller with a 64-bit AHB bus (AHB-64) and with the CPU with an AHB 32-bit bus (AHB-32 - Figure 5-8).

Therefore, the time to collect a ciphertext block on read operations is drastically reduced, allowing for PE-ICE configurations to hide the wait cycles introduced with a 32-bit bus. On write operations the former bus resource conflicts are removed since the bus can be released faster after encryption. Thus the overhead compared to AES encryption is null for PE-ICE-128 except for the additional RMW it generates (128-bit write); for PE-ICE-160 and PE-ICE-192 it is only due to the increase of the intrinsic latency of the underlying block cipher.

By considering the assumption of a 64-bit bus between PE-ICE and the memory controller, the latencies seen on the AHB bus are given in Table 5-6a for the AES-ECB engine and in Table 5-6b for PE-ICE-160 and PE-ICE-192.

Table 5-6 Additional latencies introduced by the PE-ICE configurations and by the AES-ECB engine on an AMBA-AHB bus for the operations requested by an ARM9E core with a bus width of 64-bit

(a) AES-ECB and PE-ICE-128

Operations	AES-ECB		PE-ICE-128			
	Latencies (AHB cycles)		Latencies (AHB cycles)		Overhead vs. AES-ECB	
	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$
8 to 32-bit Write	35	25	35	25	0%	0%
8 to 32-bit Read	13	8	13	8	0%	0%
4-word Write	13	8	34	24	161%	200%
4-word Read	13	8	13	8	0%	0%
8-word Write	13	8	13	8	0%	0%
8-word Read	13	8	13	8	0%	0%
16-word Write	13	8	13	8	0%	0%
16-word Read	13	8	13	8	0%	0%

(b) PE-ICE-160 and PE-ICE-192

Operations	PE-ICE-160				PE-ICE-192			
	Latencies (AHB cycles)		Overhead vs. AES-ECB		Latencies (AHB cycles)		Overhead vs. AES-ECB	
	$R_{E/B}$=1	$R_{E/B}$=2	$R_{E/B}$=1	$R_{E/B}$=2	$R_{E/B}$=1	$R_{E/B}$=2	$R_{E/B}$=1	$R_{E/B}$=2
8 to 32-bit Write	39	27	11,5%	8%	41	29	17%	16%
8 to 32-bit Read	15	9	15,5%	12,5%	16	10	23%	25%
4-word Write	15	9	15,5%	12,5%	16	10	23%	25%
4-word Read	15	9	15,5%	12,5%	16	10	23%	25%
8-word Write	15	9	15,5%	12,5%	16	10	23%	25%
8-word Read	15	9	15,5%	12,5%	16	10	23%	25%
16-word Write	15	9	15,5%	12,5%	16	10	23%	25%
16-word Read	15	9	15,5%	12,5%	16	10	23%	25%

Concerning the AES-ECB engine the number of AES cores to implement remains unchanged when compared to the architecture considered in section 5-2.4 since 4 cycles are still required to collect one plaintext block on write operations and to output a deciphered text on read operations.

For PE-ICE-128, the limiting factor is no more the bandwidth between it and the memory controller but the AHB-32 bandwidth. Thus we must consider the higher throughput required which is the one on write operations. Hence, considering the AES intrinsic latencies and that on write operations a plaintext block is collected on average all 2.66 cycles (8 cycles to collect

the 256-bit line-payload contained in three chunks on the 32-bit AHB bus), it results[26] that five AES cores must be implemented when $R_{E/B} = 1$ and three when $R_{E/B} = 2$. For PE-ICE-160 and PE-ICE-192 a plaintext block must be collected in 4 cycles. Therefore, by following the same reasoning the number N_{AES} of AES cores to implement to obtain the latencies listed in Table 5-6 is given in Table 5-7.

Once again, PE-ICE-160 offers the best trade-off since the hardware resources required in this implementation are the same as an AES-ECB encryption engine.

Table 5-7 Number of AES cores to implement for PE-ICE and the AES-ECB engine when the off-chip memory bus width is of 64-bit

	N_{AES}	
	$R_{E/B} = 1$	$R_{E/B} = 2$
AES-ECB	3	2
PE-ICE-128	5	3
PE-ICE-160	3	2
PE-ICE-192	4	2

Another possible option to improve the latency is to run the hardware encryption core at its maximum frequency (330 MHz).

5-3. Performance Evaluation

5-3.1. SoC Designer Tool Set

In order to evaluate the performance during run-time of the PE-ICE configurations exposed above the SoC designer tool set [81] is used.

The SoC Designer Developer Suite provided by ARM consists in two separate applications:

- *SoCDesigner* is used to integrate custom components modeled in SystemC (CABA – Cycle Accurate Bit Accurate) into complex SoC platforms.

[26] The reasoning on read operations with the objectives of reaching the optimum throughput of 32-bit per cycle logically leads to the same result.

- *SoCExplorer* is a cycle accurate simulator allowing to run benchmarks and to profile the platforms defined with *SoCDesigner*.

5-3.2. Simulation Platform Modeling

The SoC platform designed to evaluate the performance overhead implied by encryption and by PE-ICE is depicted in Figure 5-9. The libraries required to model the processor core (ARM926EJ-S), 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).

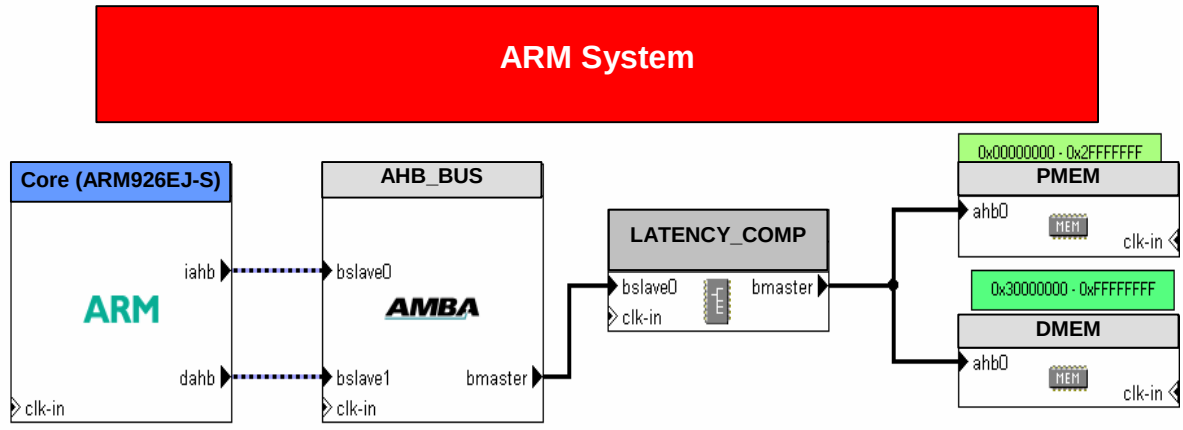


Figure 5-9 Generic architecture of the simulation platforms

To define the Base latencies we use the figures provided in the datasheet of an AHB compliant memory controller, the PL172[82]. 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
- CAS latency = 2.

In 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. Latency_comp parameters for each platform are set by adding the latencies given in Tables 5-3, 5-4 and 5-5 to the Base ones.

5-3.3. Simulation Framework

In [83, 84, 85] the proposed evaluation considers that the CPU, the AHB bus and the hardware encryption core run at the same frequencies i.e. $F_{CPU} = F_{AHB} = F_{AES}$. In this thesis 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 5-8.

Table 5-8 Architectural parameters of the simulation platforms

Processor Core	ARM926EJ-S
Processor-memory 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 (in AHB bus cycles)	9
Base off-chip Write latency (in AHB bus cycles)	1

5-3.4. Results

Four EEMBC (Embedded Microprocessor Benchmarking Consortium [86]) benchmarks (MP3 player, Huffman, CJPEG, DJPEG) and four software applications designed for embedded systems at STMicroelectronics (ADPCM, FingerPrint, DES, MP2Audio) were used in this evaluation. The simulation results for the Base platform serve as reference and are shown in IPC (Instruction Per Cycles) in Figure 5-10 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 relative to the data cache miss rate; Figure 5-11 gives this cache miss rate for each benchmark and for both sizes of the data cache.

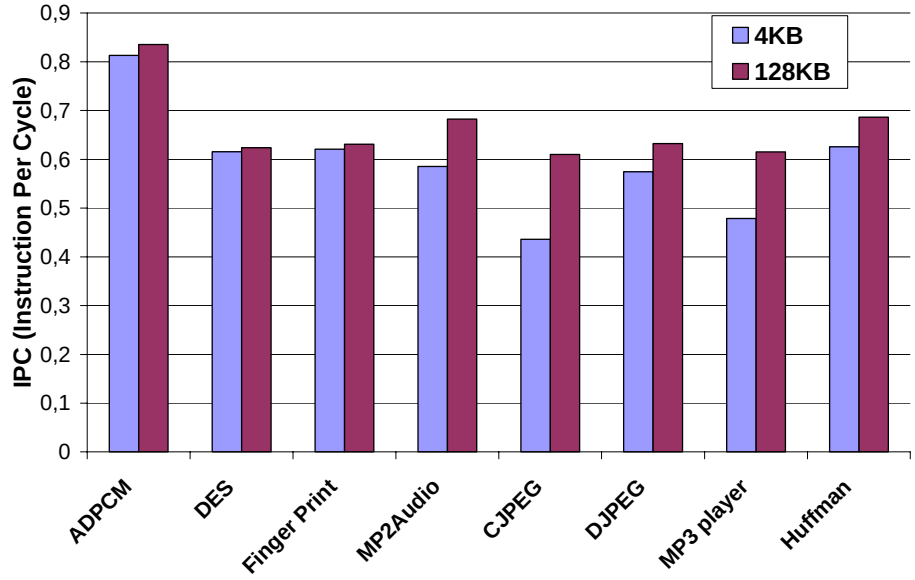


Figure 5-10 Simulation results for the Base platform

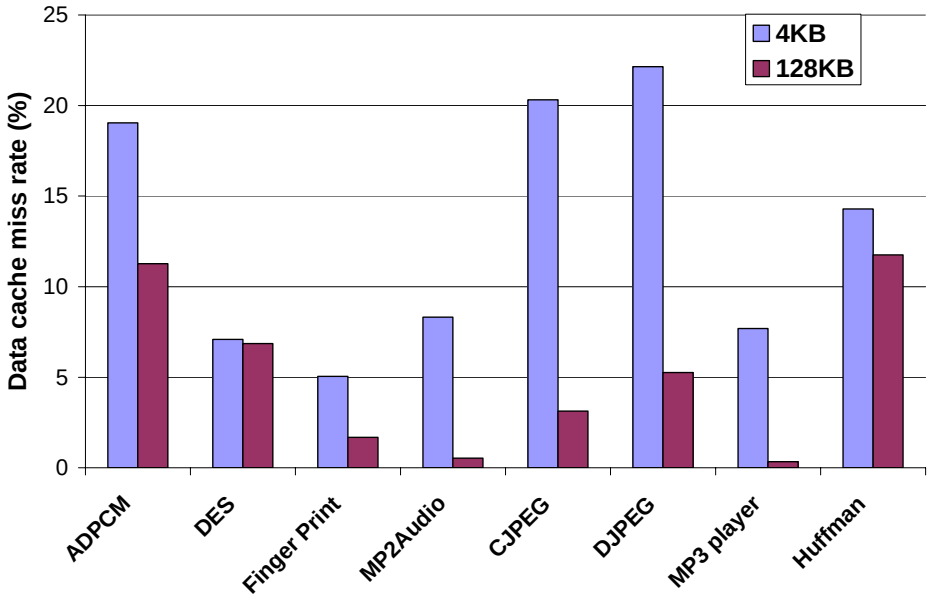


Figure 5-11 Data cache miss rate for the set of benchmarks used for the performance evaluation and for two different data cache sizes (4KB and 128KB)

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 Figure 5-12 the simulation results of the platforms emulating the AES 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% (Table 5-9) on average respectively for 4KB and 128KB data cache. This quite important performance cost can be drastically reduced by applying the improvements proposed in section 5-2.5 and by using 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 (Table 5-9), except for the DES benchmark for which the data cache miss rate remains almost the same (Figure 5-12).

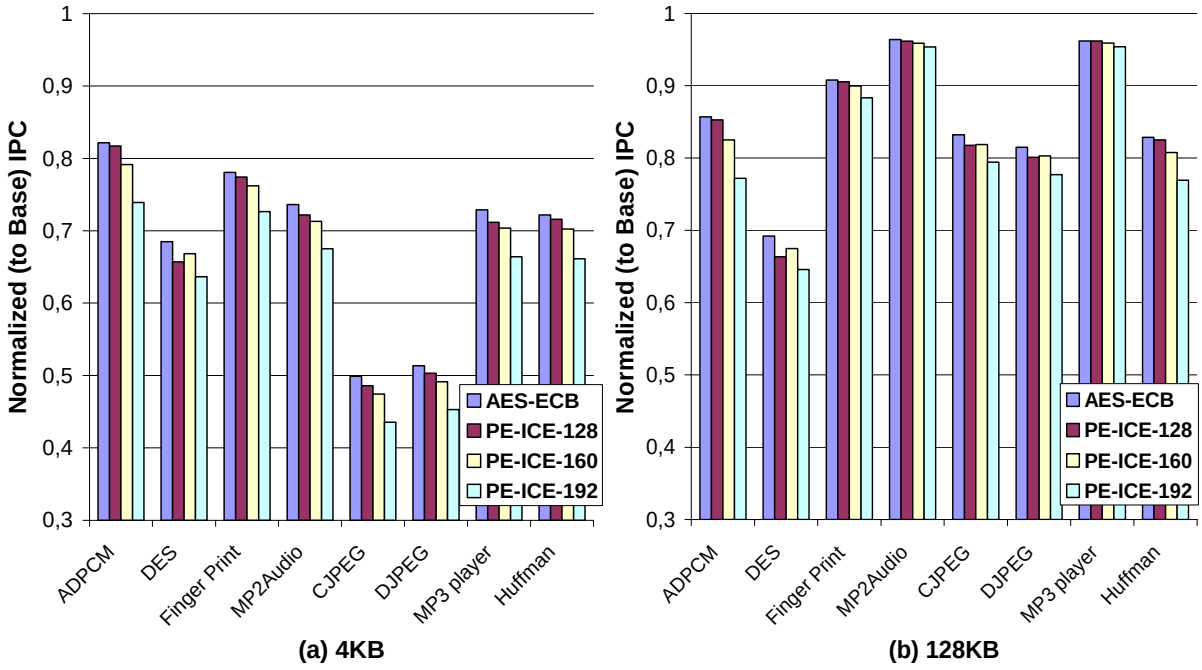


Figure 5-12 Run-time overhead of AES encryption and of the PE-ICE configurations for two data cache sizes (4KB and 128KB)

Table 5-9 Average performance slowdown implied by the AES-ECB engine and by PE-ICE configurations

	4KB	128KB
AES-ECB	31,5%	14,3%
PE-ICE-128	32,7%	15,2%
PE-ICE-160	33,7%	15,7%
PE-ICE-192	37,6%	18,1%

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 (Figure 5-13). The best results are obtained with PE-ICE-128 since on average (Table 5-10) 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) and. 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. Moreover for benchmarks requesting several 128-bit write operations PE-ICE-160 improves the performance of PE-ICE-128 (DES – CJPEG and DJPEG for 128KB data cache).

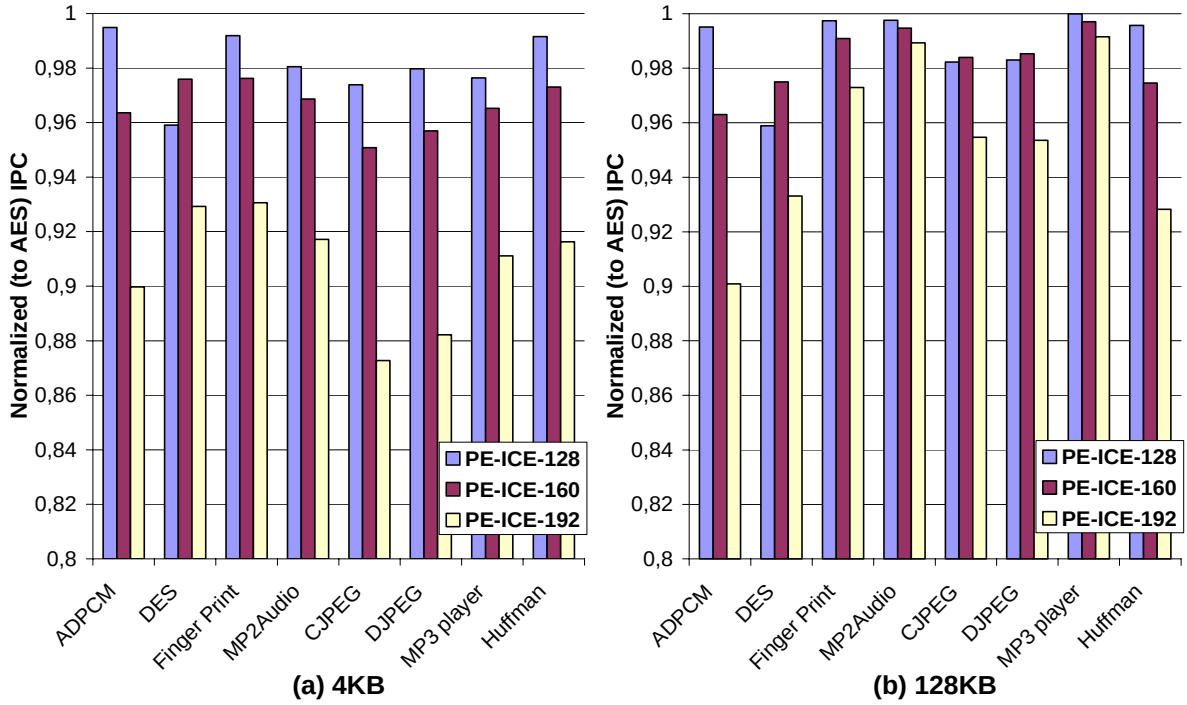


Figure 5-13 Run-time overhead of the integrity checking mechanism of PE-ICE configurations compared to AES-ECB encryption alone for two data cache sizes (4KB and 128KB)

Table 5-10 Average performance slowdown implied by the integrity checking mechanism of PE-ICE compared to AES-ECB encryption alone

	4KB	128KB
PE-ICE-128	1,9%	1,1%
PE-ICE-160	3,3%	1,7%
PE-ICE-192	9,9%	4,7%

In summary, PE-ICE-128 has the lowest overhead in terms of run-time performance when compared to AES encryption while PE-ICE-192 offers the highest strength regarding the PE-ICE security limitations. However, considering all evaluation aspects - security limitations, off-chip memory overhead, additional latencies, run-time performance and hardware cost - PE-ICE-160 is the configuration which offers the best trade-off. Therefore, in section 5-5 the comparison with a generic composition scheme is done with PE-ICE-160.

5-4. Implementation Use Case

All applications stored off-chip do not necessarily require to be protected. Thus, PE-ICE must determine at run-time if it has to handle the current memory access and which encryption key to use. Several solutions are possible, but they are deeply dependent on the underlying CPU and OS. Here we propose a use case where the design of the CPU core is not legally modifiable or at a huge cost (e.g. ARM); nevertheless it can be surely optimized when the target platform is fully specified and easily modifiable.

5-4.1. Protected Memory Region and Key Management

For clarity, in the following, we consider a simplified case where, the OS defines only two contiguous physical memory sections at application load-time and sets the following access rights in the page table (MMU) for the application (Figure 5-15):

- RO region: which contains the code and the read only data
- RW region: which contains the read write data

A straightforward solution for PE-ICE to identify the protected memory regions (PMR) is to have one table per kind of memory section, *Table_RO* and *Table_RW* (Figure 5-14). Each table entry corresponds to a given application and contains two parameters: *As* and *Ae,* the start and the end physical addresses of each memory section seen by the CPU. Such table entries are maintained by the OS that updates *As* and *Ae,* each time a new application is loaded. Thus, when a memory request is issued, PE-ICE checks if the address falls into [*As*; *Ae*] for one of the tables and, depending on the result, handles the corresponding transfer or lets it transit normally.

The same tables can be used to define the key to enroll in the encryption. As previously shown, for all PE-ICE configurations the same key is used for all RW data stored in the off-chip memory and one key per application is required for RO data. Thus for RW data a register memorizes K_{RW} and for RO data, each K_{RO} is simply stored as additional information in each *Table-RO* entry (Figure 5-13). Thus, depending on the address processed, PE-ICE selects the correct K_{RO} in *Table-RO* or K_{RW} in the RW key register.

	As (Starting address)	*Ae* (Ending address	K_{RO} (Key for Ro data)
Application 1	As_{R01}	Ae_{RO1}	K_{RO1}
Application 2	As_{R02}	Ae_{RO2}	K_{RO2}

(a) Table_RO

	As (Starting address)	*Ae* (Ending address)
Application 1	As_{RW1}	Ae_{RW1}
Application 2	As_{RW2}	Ae_{RW2}

(b) Table_RW

K_{RW}

(c) RW key register

Figure 5-14 Tables used by PE-ICE to identify the Protected Memory Regions and to select the correct encryption key

5-4.2. Physical Memory Management

As shown in Chapter 4, PE-ICE shifts the physical addressing and handles the physical address computation. However, the system must be aware of the additional memory consumption to avoid memory conflict.

Thus, the OS acts as usual at load-time i.e. it allocates an address space and sets the relevant access permission bits in the Memory Management Unit (MMU) for each section, RW and RO. However, when an application requires PE-ICE protection, the OS also computes, for each memory section defined, the size of the additional memory required for tags by multiplying R_{OF} with the size of the corresponding section and reserves in the MMU a consecutive dedicated address space AT (Additional memory for Tags) of such a size (Figure 5-15); nevertheless, access rights in the page table of the application are set to NO_ACCESS for this memory section. This way, any access to this region will fail (it will be trapped by the

MMU), and will not be passed on to PE-ICE or to the memory controller. Moreover, the OS is aware of the use of this section and will not map AT to any other application.

The solutions proposed above are transparent in terms of performance at run-time since they are done at load-time or can be parallelized on read operations with the off-chip memory access latency and on write operations with the bus cycles required to collect a plaintext block.

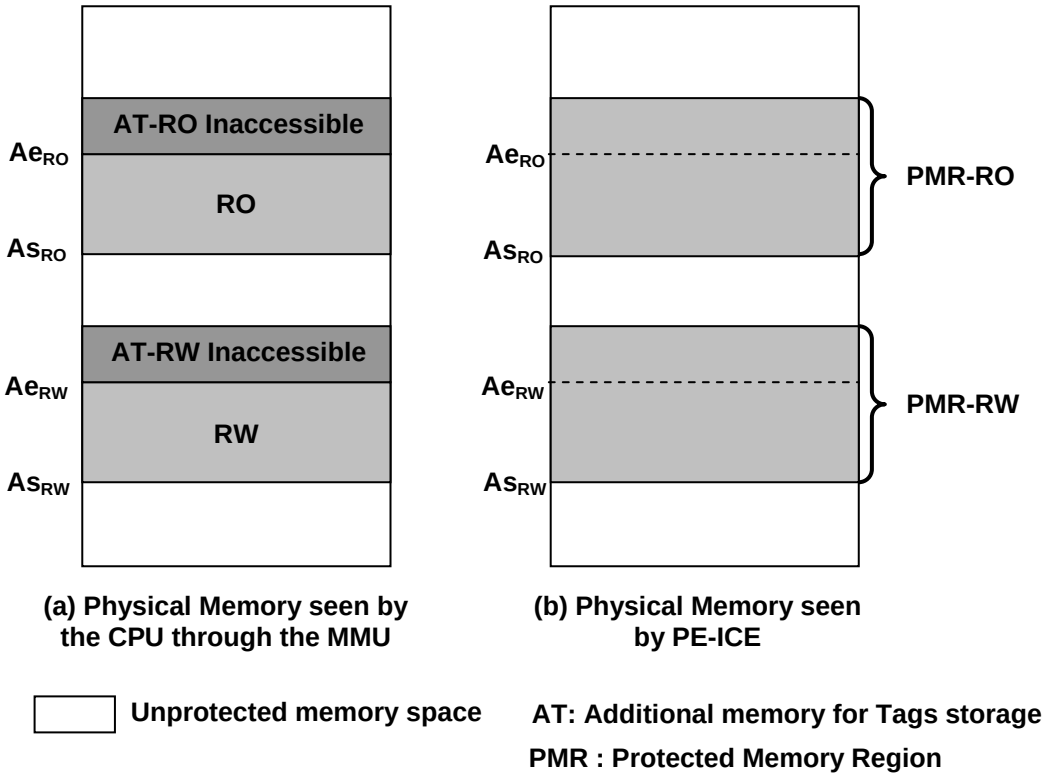


Figure 5-15 Physical memory management for PE-ICE implementation

5-5. Comparison With a Generic Composition Scheme

In this section PE-ICE is compared to a generic composition scheme referred to as GC. GC is the combination of AES-ECB encryption and of a CBC-MAC algorithm. The secure implementation of this engine in our application context is first defined and evaluated in a similar implementation context as PE-ICE (e.g. ARM9E and AHB bus). Then a comparison with PE-ICE-160 is presented to highlight the respective advantages and shortcomings of the described engines.

5-5.1. The Generic Composition Scheme: AES and CBC-MAC

5-5.1.1. Secure Implementation of GC

The tag required for the integrity checking process – described in chapter 2, section 2-2.2.3 - is computed over a chunk with a CBC-MAC algorithm. The underlying block cipher E_k for our CBC-MAC implementation is AES. The Encrypt-then-MAC construction is chosen because it is the most secured conventional method to pair an authenticated mode and an encryption mode [50]; 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 [50]. The direct implementation of the CBC-MAC presented in chapter 1 (section 1-4.4.2) is not secure since the generated tag depends only on the data and therefore is not resistant to splicing and replay attacks. That is why a 128-bit vector N is additionally enrolled in the CBC-MAC computation. 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.

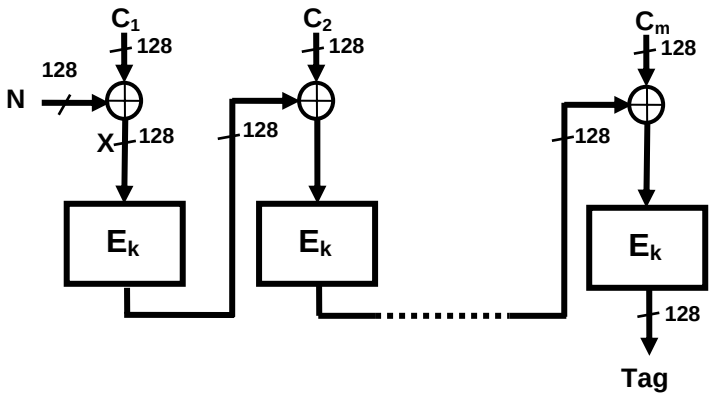


Figure 5-16 Insecure implementation of the CBC-MAC algorithm

The way N is involved in the tag computation must be carefully defined. An intuitive solution would be to simply xor N with C_1 (Figure 5-16); however such an implementation leads to a simple splicing attack. Let X be the result of the xor operation between N and C_1.

Now suppose that an adversary wants to relocate the block C_2 [27] of a chunk (C_1, C_2, …, C_m) of RO data previously recorded in memory with its tag T. This attacker knows N for RO data since it is only composed of the address and of padding. Hence, he can tamper with C_1 in order to obtain the expected X allowing to produce the correct T' corresponding to T. Such an attack will not be detected by the integrity checking engine since T will match T'.

In order to avoid such an attack, the 128-bit vector xored with C_1 must be unpredictable from an adversary point of view; therefore, N is encrypted and enrolled in the first block cipher invocation of the CBC-MAC. In this way the encryption of N is transparent since it can be parallelized with the memory access latency on read operations and with encryption on write operations[28]. The secure CBC-MAC implementation is described in Figure 5-17.

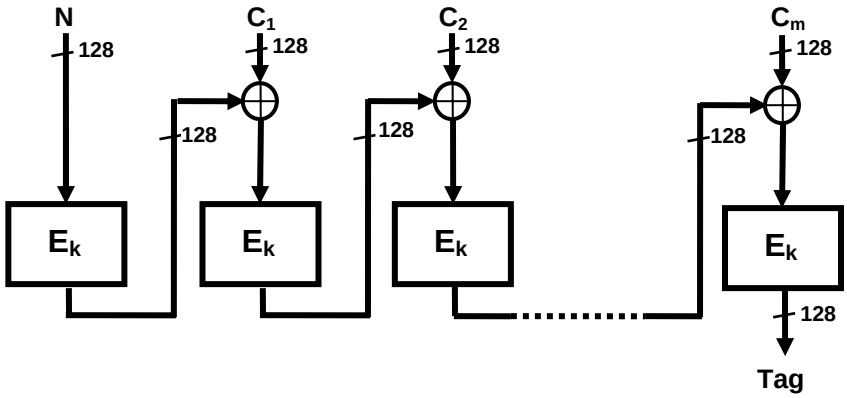


Figure 5-17 Secure implementation of the CBC-MAC algorithm

5-5.1.2. Optimized Definition of the Generic Composition Scheme

GC Encryption engine: AES encryption is done in ECB mode in order to perform a fair comparison of the integrity checking process overhead between PE-ICE and GC. We consider such an implementation secure even if the same block encrypted twice led to the same ciphertext block, which is not the case for PE-ICE. To overcome this possible security hole, CBC mode must be used with an IV containing a counter or a random value as proposed in [12].

GC MAC engine: In the literature the chunk size is defined by the cache line length. However, this choice for the CBC-MAC is inefficient for big cache block in terms of latency due to the inherent recursiveness of such a MAC algorithm. Thus, we will define a granularity

[27] Or another block after C_1 in the chunk.
[28] In [85] we enrolled N in the last block cipher invocation of the CBC-MAC computation. While secure this implementation is not optimized since the encryption of N directly impacts the memory access latency.

of integrity checking allowing an affordable trade-off between latency and hardware resources.

The number of hardware cores required for the CBC-MAC engine depends on two factors: the granularity of integrity checking (the chunk size) and the maximum length of the atomic transfers performed during run-time[29]. Indeed, in order to optimize the latencies introduced on off-chip memory access, the CBC-MAC engine must implement either the number of AES hardware cores required to reach the optimum throughput of 32-bit per cycle independently from the atomic transfer size or enough cores to be able to process in parallel all chunks contained in an atomic transfer.

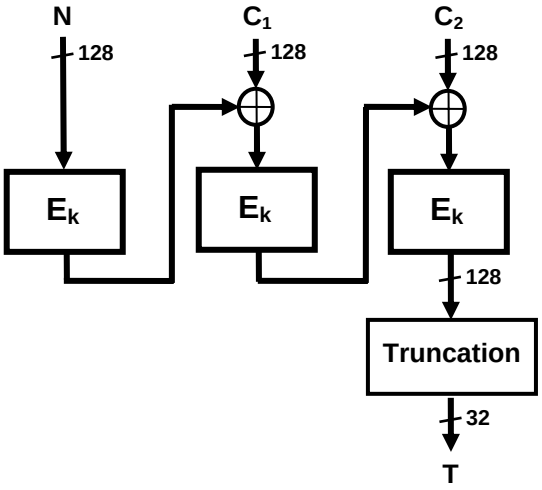


Figure 5-18 CBC-MAC implemented in the proposed generic composition scheme (GC)

The ideal implementation would be to perform per-block integrity checking (chunk size = one ciphered block *C*) with a tag computed on two blocks, *N* and *C*. The intrinsic latency of such a CBC-MAC computation is of 21 cycles for $R_{E/B} = 1$ and of 11 cycles for $R_{E/B} = 2$. Thus, the CBC-MAC engine would require respectively six and three AES cores to reach the optimum throughput of 32-bit per cycle. In order to reduce this huge hardware cost we increase the size of the granularity of the integrity checking to 256-bit (chunk size) and we only consider atomic transfers limited to 512-bit; the resulting CBC-MAC engine requires two AES cores for both values of $R_{E/B}$. This is the minimum number of core to be able, on RMW operations, to generate the new tag – for the chunk to write - and the tag reference – to check the chunk loaded to be modified.

[29] This factor is usually defined by the cache line size.

Finally, the tag is computed on $M = (N, C_1, C_2)$ and is 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 CBC-MAC implementation in GC is depicted in Figure 5-18.

5-5.1.3. Security Considerations

In the following, the defined CBC-MAC implementation is evaluated regarding the three attacks exposed in the threat model (spoofing, splicing, replay) and led on a chunk.

CBC-MAC is based on block cipher encryption; therefore its outputs are equiprobable from an adversary point of view. Concerning spoofing and splicing attacks, the chance to succeed for an attacker depends on the size of the tag and is equal to $1/2^{32}$; hence we consider that the CBC-MAC scheme protects the whole addressing space against splicing. 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 $r > 32$. Table 5-11 sums up the security limitations offered by GC.

Table 5-11 Security limitations offered by the CBC-MAC scheme implemented in GC regarding active attacks led on a chunk and evaluated in chance to succeed for an adversary

Attack	RO data	RW data
Spoofing attack	$\frac{1}{2^{32}}$	$\frac{1}{2^{32}}$
Splicing attack	$\frac{1}{2^{32}}$	$\frac{1}{2^{32}}$
Replay attack	N/A	$\frac{1}{2^{r}}$

5-5.1.4. Memory Consumption

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. For $r = 8$, this overhead is of 3.1%.

5-5.1.5. 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* can be parallelized with the memory access latency, hence the CBC-MAC latency seen on the AHB bus is only due to two consecutives AES encryptions and resulting in 21 cycles of latency for $R_{E/B} = 1$ and 11 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 25 cycles when $R_{E/B} = 1$ and of 15 cycles when $R_{E/B} = 2$.

On write operations, the CBC-MAC has the same latency (21 and 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 5-12 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 77.1% for $R_{E/B} = 1$ and of 52% for $R_{E/B} = 2$.

Table 5-12 Additional latencies introduced by GC on an AMBA-AHB bus for the operations requested by an ARM9E core

Operations	**GC (AES + CBC-MAC)**			
	Latencies (AHBcycles)		**Overhead vs. AES**	
	$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$
8 to 32-bit Write	58	38	+54%	+36%
8 to 32-bit Read	25	15	+66%	+50%
128-bit Write	32	20	+113%	+100%
128-bit Read	25	15	+66%	+50%
256-bit Write	29	14	+93%	+40%
256-bit Read	25	15	+66%	+50%
512-bit Write	29	14	+93%	+40%
512-bit Read	25	15	+66%	+50%

5-5.1.6. Hardware Cost

On read operations the 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.

The AES-ECB encryption requires three AES encryption/decryption cores for $R_{E/B}$ = 1 and two for $R_{E/B}$ = 2. As exposed in section 5-5.1.2 the proposed CBC-MAC uses two AES cores. 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 [80] 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 168 Kgates when $R_{E/B}$ = 1 and of 144 Kgates when $R_{E/B}$ = 2 in the 0.18µ technology.

5-5.1.7. Run-Time Performance

A simulation platform has been designed for GC by configuring the Latency_comp component with the latencies given in Table 5-12 for $R_{E/B}$ = 2 added to the Base ones. The simulation framework is the same as the one described in section 5-3. GC has also been evaluated for two data cache sizes: 4 KB and 128 KB.

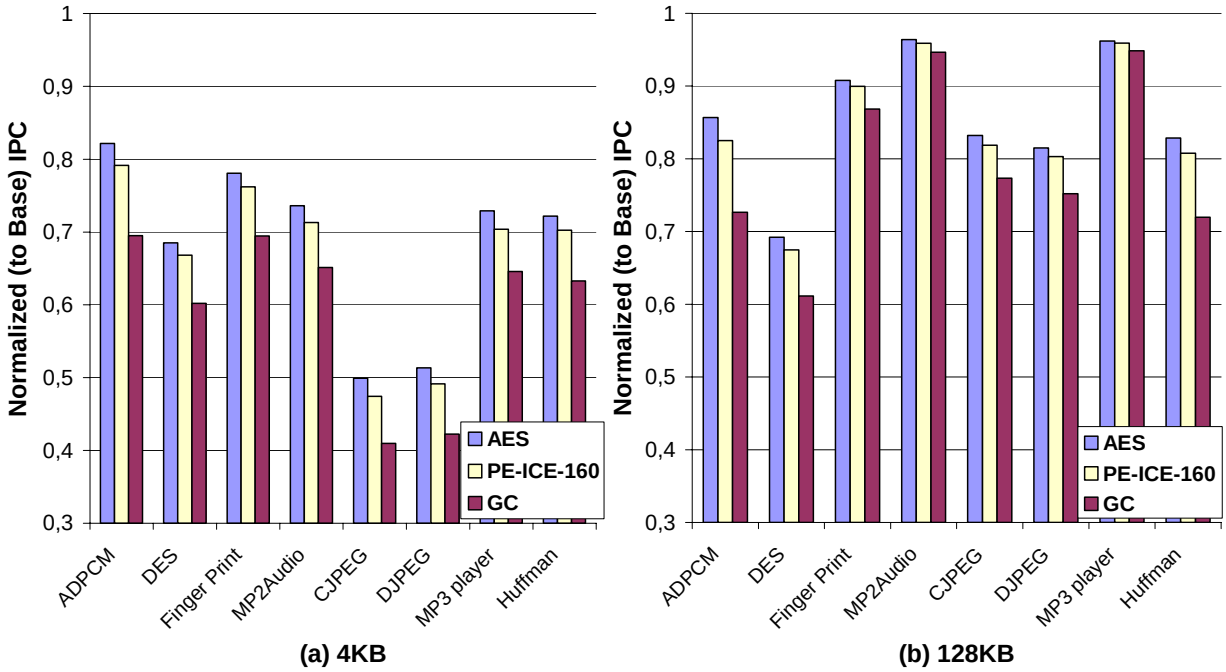


Figure 5-19 Run-time overhead of GC, of the AES-ECB engine and of PE-ICE-160 for two data cache sizes (4KB and 128KB)

Similarly to PE-ICE configurations the overhead of GC compared to the Base performance is mainly due to the encryption as shown in Figure 5-19 and in Table 5-13. Nevertheless the additional performance slowdown of the integrity checking mechanisms of GC (CBC-MAC) is non negligible (Figure 5-20) 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 (Table 5-14).

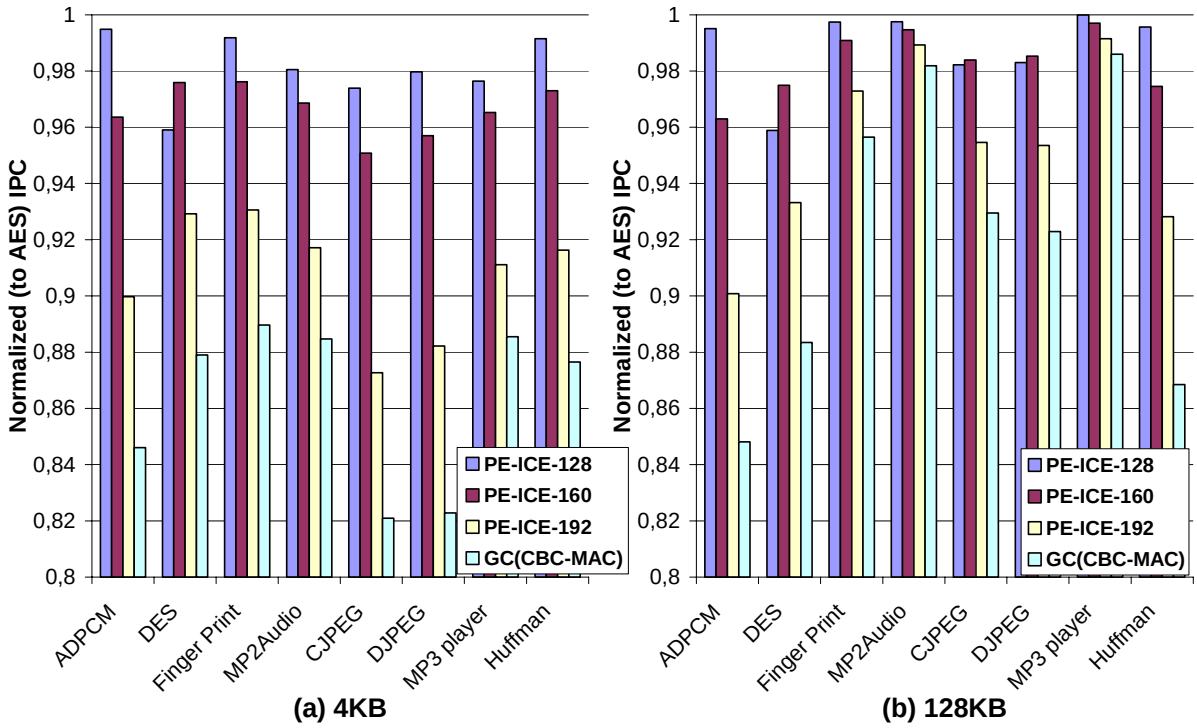


Figure 5-20 Run-time overhead of the integrity checking mechanism of GC – CBC-MAC – and of PE-ICE configurations when compared to AES-ECB encryption and for two data cache sizes (4KB and 128KB)

Table 5-13 Average performance slowdown implied by the AES-ECB engine and by GC when compared to Base

	4KB	128KB
AES-ECB	31.5%	14.3%
GC(AES-ECB +CBC-MAC)	40.6%	20.7%

Table 5-14 Average performance slowdown implied by the integrity checking mechanism of GC - CBC-MAC - compared to AES-ECB encryption alone

	4KB	128KB
GC (CBC-MAC)	13.7%	7.8%

5-5.2. 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 5-14 shows the overhead implied by the integrity checking mechanisms in PE-ICE-160 and in GC.

PE-ICE-160 does not require additional silicon area to achieve the integrity checking process while GC implies an overhead of 68% and of 80% respectively for $R_{E/B} = 1$ and for $R_{E/B} = 2$. Moreover, this feature is always valid for PE-ICE-160 while for GC it is dependent of the cache line size which fixes the maximum size of the atomic transfer during run-time.

Table 5-15 PE-ICE and GC comparison: evaluation of the cost of the integrity checking mechanisms of the two approaches when compared to the AES-ECB encryption

		GC (AES-ECB + CBC-MAC)		**PE-ICE-160**	
		$R_{E/B} = 1$	$R_{E/B} = 2$	$R_{E/B} = 1$	$R_{E/B} = 2$
Hardware cost (estimation in 0,18µ)		168 Kgates ⇔ +62%	144 Kgates ⇔ +80%	104 Kgates ⇔~0%	80 Kgates ⇔~0%
Latencies		+77.1%	+52%	+19.5%	+22%
Run-time slowdown	DC = 4KB	-	+13.7%	-	+3.3%
	DC = 128KB	-	+7.8%	-	+1.7%
Off-chip memory consumption		+12.5%		+25%	
On-chip memory consumption		$\frac{r}{256}$		$\frac{r}{128}$	

In term of latencies the overhead of GC reaches up to 77% 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. Note that all PE-ICE configurations always outperform GC during run-time (Figure 5-17).

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.

5-6. Conclusion

In this chapter we have proposed and evaluated different configurations of PE-ICE and we have highlighted that PE-ICE using the Rijndael algorithms processing 160-bit blocks offers the best trade-off between performance, security and off-chip memory consumption. Moreover, we have showed that PE-ICE improves the robustness of AES-ECB encryption and adds the integrity checking capability to block encryption without silicon area overhead and at almost free cost in terms of run-time performance. Finally the comparison with a generic composition scheme has demonstrated that the implementation of PE-ICE instead of a MAC algorithm in addition to block encryption is more efficient in terms of hardware cost and of latencies.

PE-ICE succeeds in efficiently thwarting spoofing and splicing attacks. However, as for MAC algorithms, the countermeasure against replay attacks implied an unaffordable on-chip memory overhead to store the set of reference random value when *r* is long and when a large amount of memory must be immune against replay. The next chapter proposes a scheme to overcome this issue and to reduce this on-chip memory cost to the storage of an *r*-bit value.

Chapter 6: PRV-Tree - Secure Off-chip Storage of Reference Random Values

The principle proposed by PE-ICE to thwart replay attacks is to insert randomness in plaintext blocks. However, as previously exposed the set of reference random value *(RV')* used for the tag matching process must be secret and tamper-proof from adversaries' point of view. The solution proposed in previous chapters is to store all *RV*' on-chip. While secure, this solution can be expensive in terms of on-chip memory consumption when a large amount of off-chip memory must be replay immune and when a high security level is required (big value for *r*). Hence, we present a secure way, the PRV-Tree (PE-ICE Protected Reference random Value Tree) scheme, to store this set of reference random value off-chip by mounting a tree of *RV*' and by protecting it with PE-ICE.

Note that in this chapter we only consider PE-ICE configurations for which a PE-ICE line is equivalent to a chunk i.e. PE-ICE-160 and PE-ICE-192. Thus, only the term chunk is used. Moreover the term payload implicitly denotes P_L as a line-payload.

This chapter is organized as follows. Section 6-1 introduces the notion of *m*-ary balanced tree. Section 6-2 shows how to securely store the set of reference random values off-chip. Section 6-3 describes the tree of reference random values RV-Tree and the PRV-Tree scheme allowing to store securely RV-Tree off-chip and to reduce the on-chip memory overhead to a single *r*-bit value. Section 6-4 proposes a comparison between PRV-Tree using PE-ICE-160

and hash trees. Finally, section 6-5 updates the use case presented in Chapter 4 to incorporate the PRV-Tree scheme.

6-1. *m*-ary Balanced Tree

A tree is a data structure which emulates a tree structure to link elements together. The main difference with the tree in nature is the fact that the root is the element at the top of the tree while the leaves are the elements at the bottom. The root and all the nodes of the tree until the leaves (not included) have a certain number of children. A given child (nodes or leaves) has at most one parent node and all children of the same parent node are called the siblings. The only element which does not have a parent is the root.

In this thesis, we are particularly interested in *m*-ary balanced tree where all nodes of the tree must have the same number *m* of children. Moreover, we only consider *m* values which are of a power of two[30]. Figure 6-1 depicts a 2-ary (binary) tree with three levels of elements.

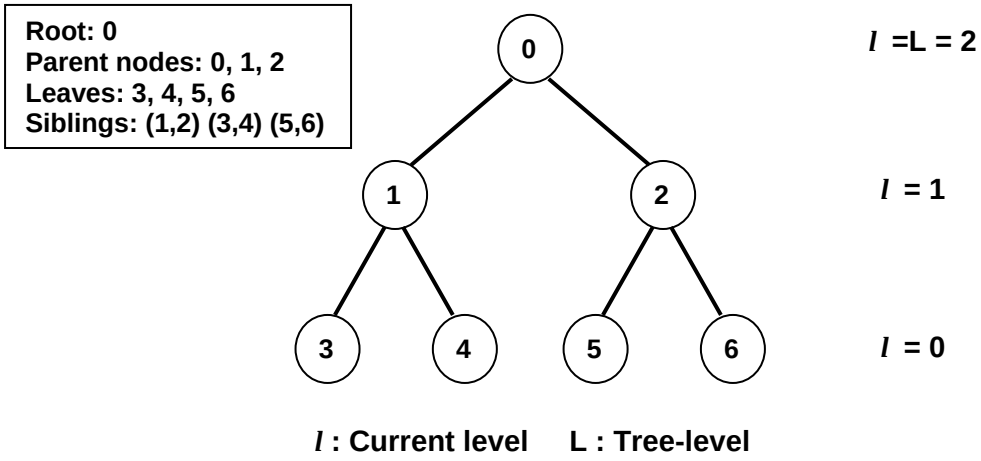


Figure 6-1 A balanced binary tree (2-ary tree)

The number L of level of nodes in the tree can be easily deduced from the number N of leave elements and from the arity m:

$$L = \log_m(N) \qquad \text{(eq. 6.1)}$$

[30] This choice is motivated by the need of easily retrieving the nodes from a memory.

The number E of elements in the tree is computed by adding the number of nodes and leaves at each level of the tree:

$$E = \sum_{l=0}^{L} m^l = \frac{m^{L+1} - 1}{m - 1}. \qquad \text{(eq. 6.2)}$$

6-2. Secure Storage Principle of the Reference Random Values

As explained in Chapter 4, the set of reference random values used for the tag matching process must be secret and tamper-proof from an adversary point of view. The solution exposed in Chapters 4 and 5 – the set of *RV'* is stored on-chip – is secure but could be expensive when a large amount of memory must be replay immune with a strong level of detection (large random value). For instance, if 256MB of RW data must be protected against replay with 32-bit random values, 64 MB of memory must be embedded for PE-ICE-160 as for PE-ICE-192. Hence, the set of *RV'* should be stored off-chip while ensuring its confidentiality – secrecy requirement – and its authentication – the tamper-proof requirement.

Therefore, we propose to apply PE-ICE to the set of *RV'*: the reference random values of consecutive chunks of data – named in the following data-chunks - in memory are concatenated to form a payload. Then this payload is processed by PE-ICE, meaning that it is concatenated with a tag consisting partially (or totally) of a random value (*RV*) before being encrypted. The resulting block is called an RVS (Reference random Value Storage) chunk or an RVS-block and is depicted in Figure 6-2. The reference random value (*RV'*) of such a chunk is stored on-chip.

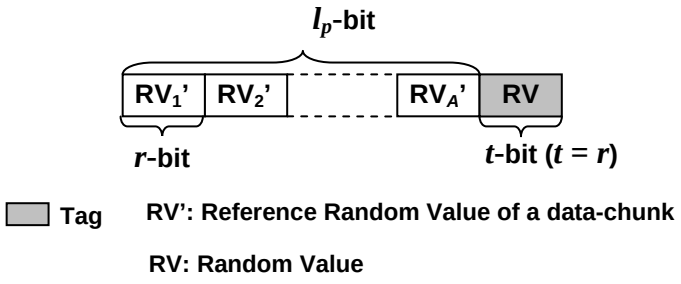


Figure 6-2 An RVS-chunk before PE-ICE encryption (for $t = r$)

The size of *RV* used in the RVS-chunk must be at least equal to *r*, the size of the random value used for the data-chunk; otherwise an adversary has more chance to succeed in a replay of a data-chunk with its RVS-block than in a replay of the data-chunk alone. In the following

we consider that the *RV* length is always equal to *r* whatever the payload processed (reference random value or RW data) by PE-ICE. The number *A* of *RV*' fitting in a RVS-chunk depends on l_p (payload size of the underlying PE-ICE configuration) and on *r* and is equal to[31]:

$$A = \left\lfloor \frac{l_p}{r} \right\rfloor . \qquad \text{(eq. 6.3)}$$

Thus, for operations on a payload PE-ICE works as follows:

- *On read operations*:
    i) load the data-chunk with the RVS-block containing the required *RV*'
    ii) decrypt both chunks
    iii) check that *RV* = *RV'* for the RVS-block by reading *RV'* on-chip
    iv) retrieve the required *RV'* in the RVS payload and check that *RV* = *RV'* for the data-chunk.
- *On write operations* of a payload:
    i) encrypt P_L with PE-ICE and a new *RV*
    ii) write the resulting data-chunk in memory
    iii) load the RVS-chunk containing the corresponding *RV'* on-chip
    iv) decrypt and authenticate it by checking that *RV*= *RV'*
    v) update the reference random value (*RV'* ← *RV*) of the data-chunk in the RVS payload
    vi) generate a new *RV* for the RVS-block and update *RV'* in the dedicated on-chip memory (*RV'* ← *RV*)
    vii) encrypt the new RVS payload with PE-ICE and the new *RV*

This scheme allows to reduce the on-chip memory overhead by a factor *A*. However, this may be insufficient. Take up again the example of 256MB of RW data to protect against replay with 32-bit random values, the amount of on-chip memory required is of 16 MB, which remains unaffordable in a SoC. The solution is to recursively repeat the application of PE-ICE on the set of reference random value *RV'* used to authenticate RVS-chunks until having only one *RV'* to store on-chip. That amounts to creating an *A*-ary tree of Reference random Values (RV-Tree) protected by PE-ICE (PRV-Tree scheme).

[31] $\lfloor X \rfloor$ denotes *X* rounding down.

6-3. PRV-Tree scheme (PE-ICE protected of the Reference Value Tree)

6-3.1. Principle

The ultimate objective of PRV-Tree is to authenticate RW data and to securely store off-chip the set of reference random value (*RV'*) required for this task by keeping only one reference random value on-chip. By securely, we mean that the confidentiality and the authenticity of the set of *RV'* is ensured.

In the following, we call RV-Tree the structure which links the reference random values together and with data while we call PRV-Tree the scheme allowing the secure storage of RV-Tree and performing the authentication process.

Each element of an RV-tree is *r*-bit long. The RW memory section to protect is divided in N_r *r*-bit blocks which are the leaves of the tree. Each node in the tree is the reference random value used to authenticate the *A*-leaves or the *A*-nodes below it – its *A* children. The top of the tree is the *RV'*-root which is kept on-chip where it cannot be tampered with. The resulting tree is securely stored off-chip (PRV-Tree) by constructing payload with *A*-sibling nodes or leaves and by encrypting them with PE-ICE resulting respectively in RVS-chunks or in data-chunks. Note that data-chunks are the chunks processed by PE-ICE in previous chapters. A 4-ary RV-Tree is depicted in Figure 6-3.

PRV-tree works as follows for RW data at run-time:

On read operations – authentication process:

i) Read the chunk to authenticate (and to decrypt) and its parent chunk in memory which contains the required *RV'*

ii) Decrypt both chunks

iii) Retrieve *RV* in the chunk's tag and *RV'* in the parent's payload

iv) Compare *RV* to *RV'*,

- *if* $RV \neq RV'$ raise the integrity checking flag and stop execution
- *else* forward the payload for speculative execution and continue the authentication process by rising up RV-Tree.

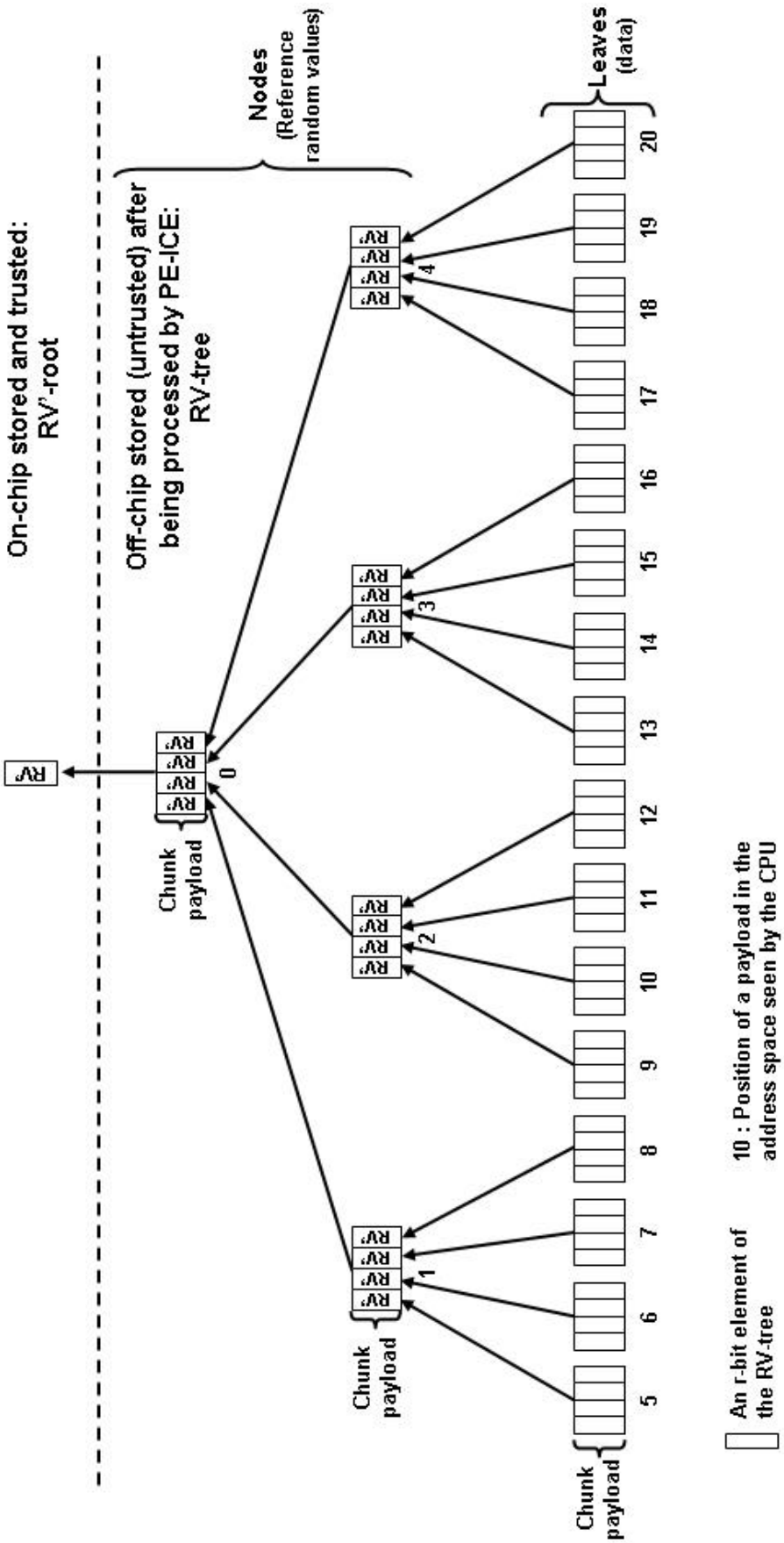


Figure 6-3 4-ary RV-Tree: Reference (random) Value Tree

This process is repeated to authenticate the parent chunk until the root of the tree. Note that the first iteration concerns the authentication of a data-chunk while the following ones concern the authentication of RVS-chunks. Moreover, starting from the second iteration only the parent chunk needs to be loaded since the chunk to authenticate has already been read during the previous iteration. The number of checks (⇔iterations) to achieve the authentication process depends on the number L of level of the RV-Tree that is easily computed from N_r and from the tree-arity A:

$$L = \log_A(N_r). \qquad \text{(eq. 6.4)}$$

On write operations the tree must be updated, thus the following steps are required:

v) read (authentication) the matching chunk (with the process described for read operations)

vi) perform the write in the decrypted payload

vii) generate a new *RV* and concatenate it with the resulting payload

viii) Encrypt and store the chunk in the off-chip memory

ix) Update the parent node with *RV'* (= *RV*) ⇔ Write RV in the parent chunk.

Similarly to read operations, this process is repeated with the parent node until *RV'*-root. Note that the first step of authentication is only required for the first iteration since it implies the authentication of all the parent's nodes belonging to the same tree branch.

The initialization of the tree is done at load-time by performing the write operations described above on all the RW memory section to protect but by turning off the integrity checking process (meaning without the first step of authentication).

6-3.2. Physical Address Computation

In this section we consider an *A*-ary RV-Tree of L level which protects N_r r-bit blocks of data or in other words N_r/A payloads of data. Moreover, l denotes the level in the tree an r-bit block of payload (of data or of *RV'*) belongs to.

For PRV-Tree to be efficient the computation of the parent's chunk addresses must be straightforward.

An RVS-chunk is retrieved in memory in two steps:

- First the address of the corresponding payload is calculated in the address space seen by the CPU.
- Then, the physical address is computed by applying the shift implied by PE-ICE processing.

To achieve the first task we use the method proposed in [47]: all elements of a tree are numbered consecutively starting from 0 from the root to the leaves as depicted in Figure 6-4 for a 4-ary tree. The position of a parent node can be easily found by subtracting one from one of its children number, by dividing the result by the arity and by rounding down.

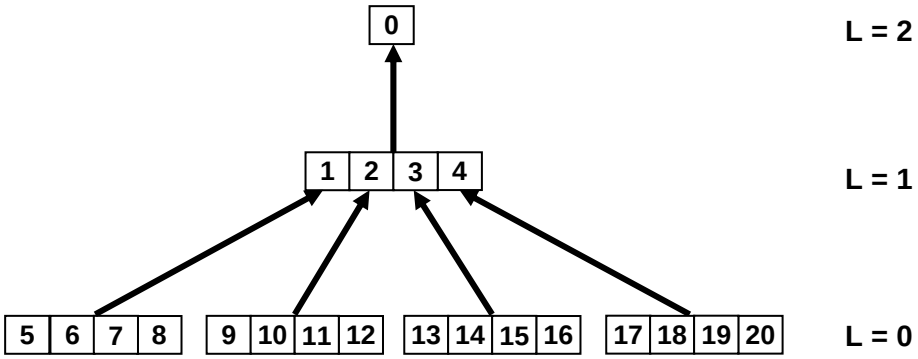


Figure 6-4 Principle of computation of parent chunk addresses on a 4-ary tree

The method exposed above can be easily applied to the tree constructed with the payloads in the RV-tree. The payloads are numbered as depicted for the RV-Tree in Figure 6-3. Considering A_S the starting address of the PMR-RW and the address provided by the CPU, the position P_0 of a payload leave is computed by (as in eq. 4.1):

$$P_0 = \frac{A_{CPU} - A_S}{l_{pb}}. \qquad \text{(eq. 6.5)}$$

Then the position of a parent payload P_l is recursively computed from the position of its child P_{l-1} or from the position of the payload leave to authenticate:

$$P_l = \left\lfloor \frac{P_{l-1} - 1}{A} \right\rfloor \quad \text{or} \quad P_l = \left\lfloor \frac{P_0 - l}{A^l} \right\rfloor \qquad \text{(eq. 6.6)}$$

When P becomes negative, this means that the parent node searched for is not in an off-chip payload but is RV-root which is stored on-chip. The computation of P shows that A must be a power of 2 to allow a straightforward hardware implementation.

Considering that the RVS-chunks are stored separately from the data-chunks starting from address $A_{T,}$ the address A_{RVS} of an RVS-chunk in the address space seen by the CPU can be easily found with:

$$A_{RVS} = A_T + P_l \times l_{pb} \qquad \text{(eq. 6.7)}$$

Finally, the shift provoked by PE-ICE processing can be applied on A_{RVS} to retrieve the physical address $A_{P\text{-}RVS}$ of an RVS-chunk (similarly as in Chapter 4 with $A_{CPU} = A_{RVS}$):

$$A_{P-RVS} = A_{RVS} + P_l \times l_{tb} \qquad \text{(eq. 6.8)}$$

6-3.3. Off-chip Memory Consumption

The off-chip memory overhead of the PRV-Tree scheme corresponds to the amount of memory required to store the RV-Tree starting from the first level of reference random value, $l = 1$, to the penultimate one, $l = L - 1$, plus the consumption implied by the PE-ICE protection of this amount of memory.

The cost of the RV-tree storage is defined by the ratio R_{Tree} between the numbers of nodes stored off-chip (level 1 to level L-1) and the number of leaves (level 0):

$$R_{Tree} = \frac{\sum_{l=1}^{L-1} A^l}{A^L} = \frac{1 - \frac{1}{A^L}}{A-1} \approx \frac{1}{A-1} \qquad \text{(eq. 6.9)}$$

by considering $\frac{1}{A^L}$ negligible.

Thus, the overall overhead of the PRV-Tree is equal to:

$$R_{PRV-Tree} = \frac{1 + R_{OF}}{A-1} \qquad \text{(eq. 6.10)}$$

6-4. Comparison between PRV-Trees (PE-ICE-160) and Hash Trees

Hash tree is the existing solution which handles the specific issue of replay attacks without generating on-chip memory overhead. In this section, we present the PRV-Tree scheme configured with PE-ICE-160 and we discuss its advantages compared to hash trees.

Table 6-1 shows the possible configurations of a PRV-Tree scheme working with PE-ICE-160 depending on the value of *r*. Those *r* values have been deduced from eq.6.3 and from the fact that *A* must be a power of 2.

The number of checks can be huge, for example when 256MB of RW data required to be protected against replay, *L* is equal to 7, 9 and 13 respectively for *r* = 8, 16 and 32 (Table 6-2). All tree schemes [37, 47] face this same issue and usually implement Authentication Speculative Execution (ASE) meaning that data are forwarded to the CPU before the end of the integrity checking process. But, PRV-Tree with PE-ICE-160 (except *r* = 32) or with PE-ICE-192 offer the possibility of performing a first step of authentication (against spoofing and splicing) by checking the address bits contained in the tag. In addition, the principle of cached tree described in Chapter 2 and proposed in [47] can be easily adapted to PRV-Tree – by caching intermediate reference random values – to reduce the number of checks to achieve.

Table 6-1 PRV-Tree with PE-ICE-160 for different *r*-values

	$r = 8$	$r = 16$	$r = 32$
A (RV-Tree arity)	16	8	4
L (number of check)	$\log_{16}(N_8)$	$\log_{8}(N_{16})$	$\log_{4}(N_{32})$
Off-chip memory overhead of PRV-Tree	8.3%	17.9%	41.7%
Overall off-chip memory overhead of PE-ICE-160 with PRV-Tree	33.3%	42.9%	66.7%

Moreover, the interesting point of PRV-Tree when compared to existing solutions like Hash trees is the drastic reduction of the memory bandwidth pollution. The memory bandwidth consumption of tree schemes can be approximated by multiplying *C* the chunk size by *L* the number of checks required to reach the tree root and to terminate the authentication process - one check requires loading at least one chunk. For PE-ICE the chunk length is always of the block size processed by the underlying block cipher (for PE-ICE-160, *C* = 160-bit). *L* can be adapted - for a given amount of RW data to be immune against replay - by playing on the *r* value to change the arity of RV-Tree (Table 6-1) and to define the

performance-security trade-off wanted. On the other hand, the hash tree principle is based on collision resistance to be secure; thus, to thwart the birthday attack the hash must be long. If we consider the implementation proposed in [37, 47], hashes are 128-bit long and chunks are 512-bit long (A = 4). The modification of the chunk size influences the tree-arity (and vice versa) but the memory bandwidth consumption is almost constant because the hash size remains the same. Concrete examples are given in Table 6-2, the memory bandwidth consumption of PRV-Tree (PE-ICE-160) and of Hash-tree are computed for two RW memory region sizes to protect against replay (16MB and 256MB) and for different configurations of both engines. We show that PRV-Tree reduces the memory bandwidth pollution by a factor 3 for a comparable security level (PRV-Tree with r = 32). By decreasing the security level, this factor reaches 4.1 on average when r = 16 and 5.3 when r = 8. This improvement provided by PRV-Tree is mainly the result of the fine granularity of integrity checking of PE-ICE.

Table 6-2 Memory bandwidth consumption of tree schemes for two different sizes of the RW memory section to protect against replay, 16MB and 256 MB

(a) PRV-Tree (PE-ICE-160) for different *r* values

	PE-ICE-160 with PRV-Tree C^* = 160-bit					
	$r = 8; A = 16$: $L = \log_{16}(N_8)$		$r = 16; A = 8$ $L = \log_8(N_{16})$		$r = 32; A = 4$ $L = \log_4(N_{32})$	
	RW=16MB	RW=256MB	RW=16MB	RW=256MB	RW=16MB	RW=256MB
L value (Number of checks)	6	7	8	9	11	13
Memory bandwidth consumption for data authentication	120B	140B	160B	180B	220B	260B

(b) Hash-Tree [37, 47] for different chunk sizes

	Hash Tree (hash of 128-bit)					
	$C^* = 256; A = 2$ $L = \log_2(N_{128})$		$C^* = 512; A = 4$ $L = \log_4(N_{128})$		$C^* = 1024; A = 8$ $L = \log_8(N_{128})$	
	RW=16MB	RW=256MB	RW=16MB	RW=256MB	RW=16MB	RW=256MB
L value (Number of checks)	20	24	10	12	7	8
Memory bandwidth consumption for data authentication	640B	768B	640B	768B	896B	1024B

* C is the chunk size in bit; N_r is the protected memory size expressed in r-bit blocks where r is the size of the random values for PE-ICE and the hash length for hash tree.

Another non negligible advantage of PRV-Tree is that the update process of the tree on write operations is fully parallelizable since the tag (random value and possibly address-bits) in PE-ICE is produced independently from the payload it protects. For Hash trees, a hash parent is computed over the data or over the hash children, thus the process is done in series.

Moreover, PRV-Tree does not induce an overhead in terms of hardware since it uses the same hardware as PE-ICE.

The main shortcoming of PRV-Tree with PE-ICE to achieve both data confidentiality and integrity is the off-chip memory overhead. For PRV-Tree plus PE-ICE-160 this overhead reaches 66.7% while for a comparable security level the Hash-Tree scheme implemented with an OTP engine in [37] consumes 41% of memory. Nevertheless, PRV-Tree can be configured to achieve an acceptable security level (r = 16) while requiring almost the same memory consumption of 42.9% (Table 6-1).

6-5. Implementation Use Case

In this section the use case presented in Chapter 5 (section 5-4) is updated to incorporate the PRV-Tree scheme.

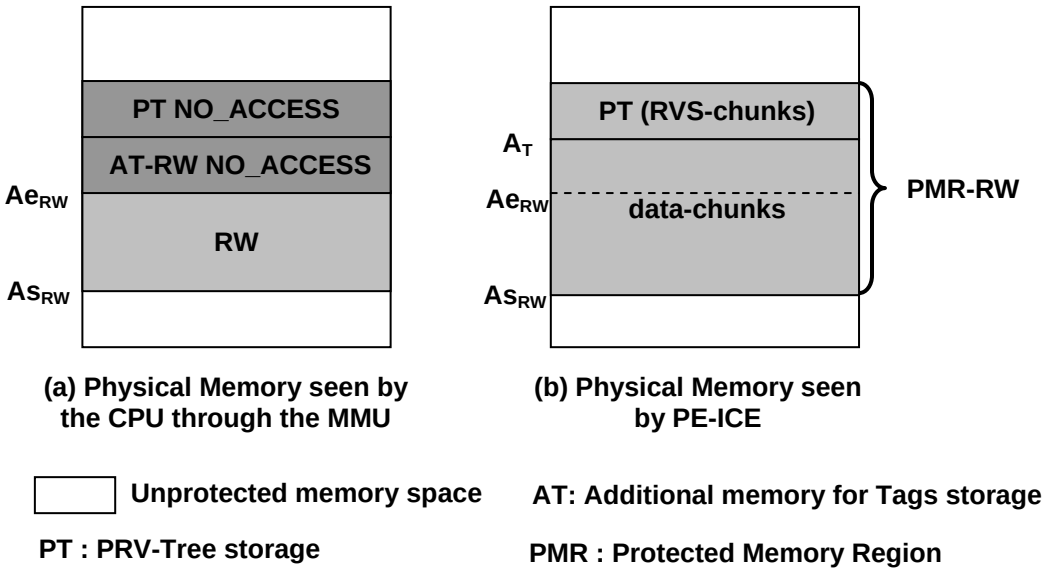


Figure 6-5 Physical memory management for PE-ICE with PRV-Tree

At load-time the OS computes the size of the additional memory required to store the RVS-chunks (PRV-Tree starting from l = 1 to $l = L$ - 1) by multiplying $R_{PRV\text{-}Tree}$ with the size

of the RW memory section. Then it reserves in the MMU a consecutive dedicated address space PT (PRV-Tree) of such a size (Figure 6-5); nevertheless, access rights in the page table of the application are set to NO_ACCESS for this memory region. In this way, any access to this region will fail (it will be trapped by the MMU), and will not be passed on to PE-ICE or to the memory controller. Moreover, the OS is aware of the use of this section and will not map PT to any other application.

The encryption key used for PRV-Tree can be K_{RW} since *RV'* values are processed like RW data by PE-ICE (same tag generation). Moreover an entry must be added in Table_RW (Figure 6-6) to store the start address of the PT region for PE-ICE to be able to compute the address of the parent chunks.

	As (Start address of RW section)	*Ae* (End address of RW section)	A_T (Start address of PT region)
Application 1	As_{RW1}	Ae_{RW1}	A_{T1}
Application 2	As_{RW2}	Ae_{RW2}	A_{T2}

Figure 6-6 Table_RW used by PE-ICE to identify the Protected Memory Regions of RW data and to retrieve parent chunks in memory

6-6. Other applications of PRV-Tree

PRV-Tree can be used to reduce the on-chip memory overhead implied by security mechanisms with replay attack countermeasures requiring to store reference values on-chip. For example, the reference random values needed by the generic composition scheme GC - presented in Chapter 5, section 5-5 – to thwart replay attacks can be stored off-chip by using the PRV-Tree scheme.

Moreover, as mentioned in Chapter 4 the value used as tag in PE-ICE does not necessarily need to be random. It could be a nonce. However, when the nonce reaches its limit, all the memory protected with this nonce must be re-encrypted with a new encryption key.

PRV-Tree can also be used to protect reference value generated from nonces. However, as for PE-ICE when the nonce reaches its limit, the whole tree must be re-encrypted with a new

key. Otherwise, an adversary can predict when he can replay an old node of the tree with its children.

6-7. Conclusion

In this Chapter we presented a scheme (PRV-Tree) allowing to efficiently protect RW data against replay by allowing to store the reference values required to achieved such a goal off-chip. Compared to existing tree schemes achieving the same objective – Hash trees – PRV-Tree offers numerous advantages like the reduction of the memory bandwidth pollution at run-time – thanks to the fine granularity of integrity checking of PE-ICE - the parallelization of the operations on write (tree update) and the use of the same hardware as encryption engines. Moreover, PRV-Tree allows the designer a scalable choice in the trade-off between security and performance by playing on the r value. Finally, in certain configurations of PE-ICE (tag for RW data composed of address bits with random bits), PRV-Tree allows partial integrity checking: it gives a first response against spoofing and splicing attacks.

PRV-Tree can be easily applied to existing integrity checking scheme requiring to store reference values on-chip. Moreover, those reference values do not necessarily need to be generated from random but can be also generated from nonces.

However, PRV-Tree must be implemented with the cached tree scheme to clearly quantify its benefit when compared to Cached Hash trees.

Chapter 7: Conclusion

This chapter concludes the thesis. Section 7-1 summarizes the contributions. Section 7-2 describes the remaining works. Finally, section 7-3 proposes further ideas to improve the concept of PE-ICE and of PRV-Tree.

7-1. Contributions

The objective of countermeasures against physical attacks at the board level (bus probing and memory tampering) is to provide a private and authenticated tamper resistant environment to applications running on the targeted devices. Thus it is required to guarantee the confidentiality and the integrity of data transferred between the processor and the off-chip memory.

The conventional way to ensure those security services in a System on Chip is to design a hardware engine dedicated to each security service. Such an approach leads to a non-optimal usage of the required silicon area and implies non-parallelizable latencies generated by the underlying computation: encryption/decryption, tag computation/tag matching process (integrity checking).

In this thesis we explored the addition of redundancy and of randomness in block encryption at the block level to efficiently ensure the confidentiality and the integrity of data transferred on the processor-memory bus of embedded systems. By verifying such redundancy and randomness after decryption, we add the integrity checking capability to block encryption. In this way, a single algorithm is implemented to achieve both security services. Moreover, the key point of the proposed engine PE-ICE is that the tag is generated

independently from the data it protects. As a result that PE-ICE enables the following features:

- *Latency optimization*: The encryption (respectively decryption) and the tag computation (respectively integrity checking) processes are done in parallel, allowing to optimize the latencies introduced on write and read memory access latencies.
- *Hardware resources optimization*: the same hardware resources are used to achieve all processes: encryption/decryption and integrity checking.
- *Optimization of the RMW operations processing*: We introduced the notion of fine granularity of integrity checking allowing to reduce the memory bandwidth pollution on RMW operations.

The insertion of randomness dedicated to handle the specific issue of replay attacks in PE-ICE generates a huge on-chip memory overhead to store the reference random value required for the tag matching process (integrity checking). The proposed solution to reduce this overhead to the on-chip storage of a single random value, PRV-Tree, relies on the PE-ICE principle to mount a tree of reference random values and to securely store them off-chip. PRV-Tree improves existing countermeasures against replay based on tree schemes (hash trees) in the following aspects:

- *Reduction of the memory bandwidth pollution*: The authentication process relying on a tree scheme requires successive loading of tree elements. PRV-Tree reduces the memory bandwidth pollution by decreasing the size of the chunks (a chunk defining the granularity of the integrity checking process).
- *Full parallelization*: PRV-Tree is parallelizable on read and on write operations. This feature is enabled by the fact that the tags are generated in PE-ICE independently from the data it protects.
- *First step of authentication*: Contrary to existing tree schemes, PRV-Tree does not require the end of the whole authentication process to give a response to spoofing or splicing attacks. If address bits are contained in tags, after the first check, PRV-Tree is able to detect data relocation (splicing) or random insertion (spoofing).
- *Scalability*: By choosing the size of the random value the designer easily defines the desired trade-off between security, performance and off-chip memory overhead.

Moreover, PRV-Trees can be easily applied to integrity checking scheme requiring to store reference values on-chip. In addition, the reference values protected by PRV-Tree do not necessarily need to be random but can be generated from nonce.

PE-ICE and PRV-Tree are secured regarding the attacks described in our threat model if the assumptions considered in this work are respected:

i) the SoC package must be tamper-resistant to be able to trust data stored inside.

ii) the random number generator must not leak information on the values it outputs. Those values are used to construct tags for RW data which must be unpredictable.

In this way the security of the proposed mechanisms relies on the security of the underlying block cipher since an adversary has only access to ciphertexts. Moreover, concerning the three active attacks considered (spoofing, splicing, replay), the designer must be aware of the security limitations given in this thesis.

7-2. Further Works

During this thesis the implementation of the concept of PE-ICE has been started on the LEON2 processor. This hardware experimentation must be terminated to refine the SoC integration issues of PE-ICE.

Moreover, the PRV-Tree must be evaluated at run-time to quantify its improvement in terms of performance. An ideal implementation should include a cached tree mechanism to efficiently decrease the number of checks usually required by a tree scheme.

In this thesis, our work focuses on uniprocessor systems. Recently several research projects [87, 88] studied the off-chip memory protection of multiprocessor platforms. The main additional issue of such platforms compared to uniprocessor is to distinguish a modification performed by one of the CPU from a corruption conducted by a malicious party. Thus, a challenge would be to explore how to extend the protection provided by our proposed mechanisms to multiprocessor systems.

7-3. Further Idea: PE-ICE-OTP

The principle of PE-ICE cannot be simply extended to stream cipher since parallelizable mode, such as CTR, does not provide error propagation or diffusion: a bit change in the

ciphertext only impacts the matching bit in the text resulting from decryption. Hence, if a tag is appended to a payload before encryption, an adversary could localize the encrypted tag in the ciphertext and replace it with a chosen one, thus inserting fake text at will (the tag is not computed on data).

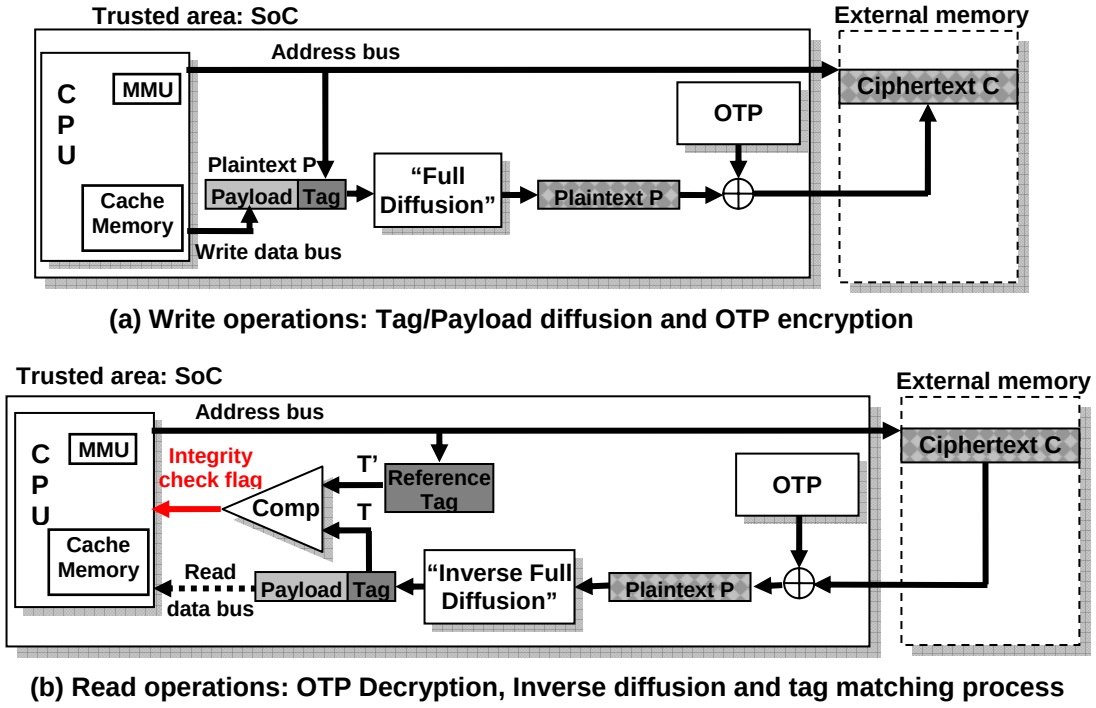


Figure 7-1 PE-ICE-OTP Principle

The idea is to work on an un-keyed diffusion function implemented before an OTP encryption. This function must provide full diffusion meaning that every bit at its output must depend on all bits at its input. For instance, the AES algorithm fulfills such requirement in two rounds [15]. Thus PE-ICE-OTP could work as follows:

- On *write operations* (Figure 7-1a), the payload is mixed with a tag with the full diffusion function in parallel of the OTP pad computation and before the encryption step strictly speaking (xor operation).
- On *read operations* (Figure 7-1b), the ciphertext loaded is first decrypted with the OTP scheme and then the inverse transformation of the diffusion function is applied to the decrypted text. Finally, the tag matching process is done to authenticate the payload. Thus, if an adversary tampers with or modifies one or some bits in the ciphertext, the corruption will impact all plaintext bits and will be detected during the tag matching process.

Contrary to PE-ICE, decryption and integrity checking processes are serialized but the preprocessing feature of OTP compensates those non-parallelizable computations.

PE-ICE-OTP can further decrease the latencies on read and write operations when compared to PE-ICE and speeds up the authentication process of PRV-Tree by profiting from the preprocessing feature of the OTP scheme.

UNIVERSITE MONTPELLIER II
SCIENCES ET TECHNIQUES DU LANGUEDOC

Résumé de thèse – French Summary

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE MONTPELLIER II

Discipline :Génie Informatique, Automatique et Traitement du Signal
Formation Doctorale : Systèmes Automatiques et Microélectroniques
Ecole Doctorale : Information, structures, Systèmes

présentée et soutenue publiquement

par

Reouven ELBAZ

Le 12 Décembre 2006 à Montpellier

Mécanismes Matériels pour des Transferts Processeur Mémoire Sécurisés dans les Systèmes Embarqués

JURY

M. Jean Claude Bajard, Professeur, Université de Montpellier II,	Président du jury
M. Viktor Fischer, Professeur, Université de St Etienne,	Rapporteur
M. Olivier Sentieys, Professeur, Université de Rennes I,	Rapporteur
M. Joan Daemen, Docteur, Société STMicroelectronics,	Examinateur
M. Pierre Guillemin, Société STMicroelectronics,	Examinateur
M. Jean Baptiste Rigaud, Maître de Conférence, Centre Microélectronique de Provence,	Examinateur
M. Gilles Sassatelli, Chargé de Recherche CNRS, LIRMM – UMII,	Examinateur
M. Lionel Torres, Professeur, Université de Montpellier II,	Directeur de Thèse

Sommaire

Liste des Figures

Liste des Tableaux

Mécanismes Matériels pour des Transactions Processeur-Bus Sécurisées dans les Systèmes Embarqués

1. Introduction

Les systèmes embarqués actuels (téléphone portable, assistant personnel) ne sont pas considérés comme des hôtes de confiance car le propriétaire ou toute personne y ayant accès, sont des attaquants potentiels [1]. Les données contenues dans ces systèmes peuvent être sensibles (données privées du propriétaire, mot de passe, code d'un logiciel…) et sont généralement échangées en clair entre le Système sur Puces (SsP) et la mémoire dans laquelle elles sont stockées. Le bus qui relie ces deux entités constitue donc un point faible : un attaquant peut observer ce bus et récupérer le contenu de la mémoire, ou bien a la possibilité d'insérer du code afin d'altérer le fonctionnement d'une application s'exécutant sur le système. Un exemple illustrant parfaitement cette problématique est l'attaque menée par Markus Kuhn [31] sur un micro contrôleur 8 bits implémentant un système de chiffrement. Le principe de son attaque est le suivant i) observer le bus ii) identifier les instructions qui permettent de faire sortir des données de la mémoire externe sur le port parallèle iii) construire un code qui fait sortir la totalité du contenu de la mémoire externe sur le port

parallèle iv) insérer ce code sur le bus lorsque le processeur adresse un emplacement mémoire adéquat.

Afin de prévenir ce type d'attaque, des mécanismes matériels doivent être mis en place afin d'assurer la confidentialité et l'intégrité des données. Dans notre contexte, garantir la confidentialité consiste à rendre incompréhensibles les données circulant sur le bus ou stockées en mémoire externe, tandis que la vérification de l'intégrité a pour but de détecter toute insertion ou altération de codes ou de données.

Dans le domaine de la sécurisation des communications entre un processeur et sa mémoire externe, les systèmes de protection des données utilisent la méthode conventionnelle, c'est-à-dire un système matériel dédié à chacune des propriétés : confidentialité et intégrité. La conséquence d'une telle approche est la réalisation en série des opérations requises par chaque composant, et l'addition des surcoûts en performance. Par ailleurs, les ressources matérielles ne sont pas optimisées car deux algorithmes doivent être implantés en matériel.

L'objectif de PE-ICE (Moteur de chiffrement et de contrôle d'intégrité parallélisé de l'anglais Parallelized Encryption and Integrity Checking Engine) est de fournir une solution dédiée aux transactions de bus entre un processeur et sa mémoire afin d'optimiser, d'une part, les latences introduites par les mécanismes matériels de sécurité lors de tout type d'opérations (lecture et écriture) et d'autre part la surface de silicium requise pour leur implémentation sur un SsP. Son principe est basé sur l'utilisation d'un unique algorithme de chiffrement par bloc afin de fournir aux données les propriétés de confidentialité et d'intégrité.

Cette thèse est organisée de la façon suivante. La section 2 présente notre modèle de menace. La section 3 présente brièvement l'outil cryptographique, puis expose le contexte de l'étude et enfin décrit les techniques permettant d'assurer l'intégrité et la confidentialité des données ainsi que les engins dédiés à la protection des mémoires externes proposés dans l'état de l'art. La section 4 explique les spécificités de PE-ICE permettant de conférer à un algorithme de chiffrement par bloc la capacité de vérification d'intégrité. Un exemple d'implémentation ainsi que son évaluation (en terme de performance et de surcoût mémoire et en surface de silicium) sera également exposé dans cette section. De plus, une évaluation d'un engin de chiffrement de bus ne garantissant que la confidentialité est proposée afin d'estimer le coût de l'intégrité engendré par PE-ICE. Finalement, les perspectives et les futures améliorations sont exposées en section 5.

2. Modèle de Menace

Les attaques sur le Système sur Puce (SsP) ne sont pas considérées, seules celles menées au niveau de la carte et plus particulièrement impliquant l'observation ou l'insertion de données sur le bus entre le processeur et la mémoire ou directement en mémoire doivent être contrées (Figure 1).

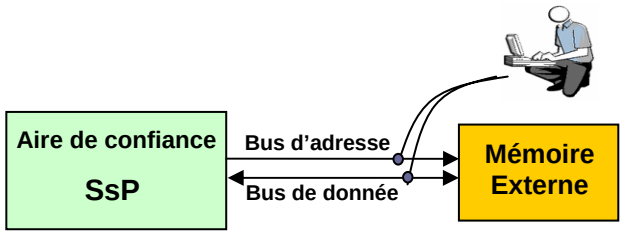


Figure 1. Attaque au niveau de la carte par observation de bus

La famille d'attaques appelée « l'homme au milieu » nous concerne tout particulièrement. Son protocole comporte deux étapes. La première phase est passive et consiste à récupérer des données sur le bus et à les analyser. Elle a pour objectif principal de mettre en défaut la confidentialité des données. La seconde phase est active : l'attaquant remplace ces données par d'autres de son choix, remettant en cause l'intégrité de ces données. L'observation de l'exécution des blocs mémoire interceptés lors de la première étape de l'attaque peut aider l'attaquant dans le choix de ceux à insérer sur le bus (ou en mémoire) lors de cette seconde phase. Trois types d'attaques actives sont alors définis en fonction du choix fait par l'attaquant du bloc à insérer sur le bus ou en mémoire:

- *L'attaque par insertion aléatoire* : l'attaquant remplace un bloc mémoire intercepté sur le bus par un autre choisi aléatoirement.
- *L'attaque par relogement* : l'attaquant échange un bloc mémoire récupéré sur le bus par un autre préalablement enregistré sur ce même bus. Cette attaque peut être vue comme une permutation spatiale de blocs mémoire.
- *L'attaque par rejeu* : l'attaquant récupère un bloc mémoire à une adresse spécifique et le réinjecte plus tard sur le bus lorsque le processeur adresse ce même espace mémoire. Cette attaque peut être vue comme une permutation temporelle de blocs mémoire.

3. Contexte et Etat de l'Art

3.1. L'Outil Cryptographique

La confidentialité des données est assurée par leur chiffrement avec des algorithmes de chiffrement symétrique. Ce type d'algorithme est divisé en deux familles : le chiffrement par flux – le texte en clair est traité bit par bit – et le chiffrement par bloc – le texte en clair est dans un premier temps divisé en blocs de même longueur puis les blocs sont traités un à un. Pour plus de détails sur les techniques de chiffrement consulter [16, 24].

Le contrôle de l'intégrité est effectué grâce à des fonctions dites de hachage qui permettent de calculer une image compressée de la donnée à protéger. Cependant, il est parfois requis d'identifier également la source qui émet ces données. Des algorithmes de MAC (de l'anglais Message Authentication Code) sont alors utilisés ; ils consistent en une fonction de hachage qui prend en entrée une clef secrète (détenue à la fois par l'entité source et par le destinataire d'un message) en plus de la donnée. Lors de la transmission d'un message *M*, une étiquette *T* est alors calculée par un algorithme de MAC et apposée à *M* avant envoi. Le destinataire utilise alors le même algorithme de MAC et la même clef que celle employée par l'entité source pour recalculer une étiquette de référence *T*'. Il compare *T* et *T*'. Si elles sont différentes, cela signifie que le message a été corrompu au cours de sa transmission.

3.2. Contexte

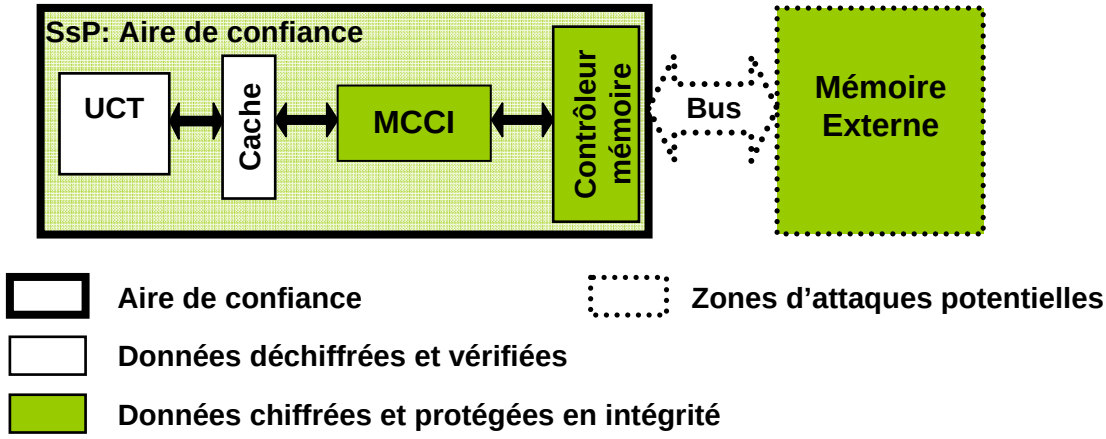
3.2.1. Emplacement des Mécanismes Matériels de Sécurité

Les mécanismes matériels de sécurité sont généralement placés sur le SsP entre la mémoire cache et le contrôleur de mémoire externe (figure 2) pour les raisons suivantes :

- *Performance* : cette localisation permet de stocker les données déchiffrées et vérifiées en termes d'intégrité, dans la mémoire cache embarquée dans le système sur puce. De cette manière les latences introduites par les fonctions cryptographiques sous-jacentes n'ont pas d'impact sur tous les accès mémoire.
- *Sécurité* : les valeurs qui doivent rester secrètes telles que les clefs de chiffrement utilisées par les algorithmes de cryptographie, sont stockées sur l'aire de confiance (le

SsP). Elles peuvent ainsi être considérées comme inaccessibles et secrètes pour un attaquant.

- *Compatibilité* : la conception des mécanismes de sécurité est complètement indépendante du type de mémoire utilisée.



MCCI: Mécanismes matériels de Chiffrement et de Contrôle d'Intégrité

Figure 2 Localisation des mécanismes matériels de chiffrement et de contrôle de l'intégrité sur le Système sur Puce

3.2.2. Principe de chiffrement sur Bus

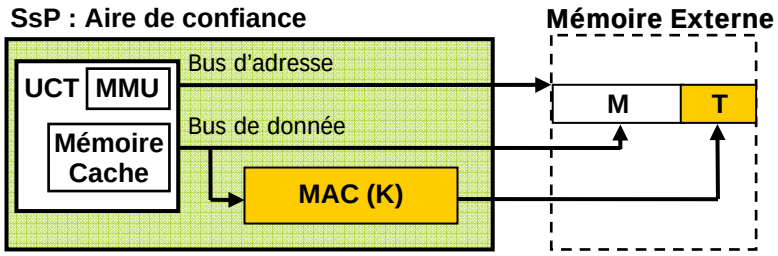
Garantir la confidentialité du contenu des mémoires externes consiste à éviter toute fuite d'informations sensibles sur le bus ou dans la mémoire. Par conséquent, les données transitant sur le bus entre le processeur et la mémoire et stockées dans la mémoire doivent être rendues incompréhensibles pour un attaquant. Le chiffrement du bus permet d'atteindre cet objectif. Le concept a été introduit par Best [8, 9, 10] à la fin des années 70. Le principe est simple : les données sont chiffrées lorsqu'elles quittent l'aire de confiance (SsP) lors des opérations d'écriture et déchiffrées dans le SsP lors des opérations de lecture. De cette manière, les données ne sont accessibles à un attaquant (sur le bus ou en mémoire) que dans leur version chiffrée et donc théoriquement incompréhensibles.

3.2.3. Principe de contrôle de l'intégrité du contenu mémoire

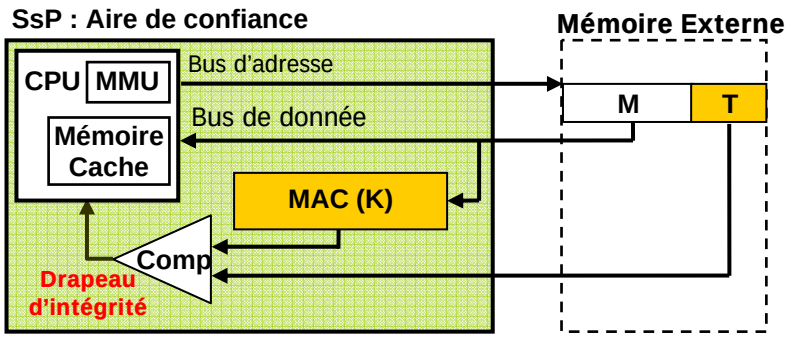
La vérification de l'intégrité du contenu mémoire consiste à contrôler qu'une donnée n'a pas été modifiée lors de sa transmission sur le bus ou directement en mémoire. Afin d'atteindre cet objectif, une étiquette *T* est ajoutée à la fin de chaque bloc mémoire *M* lors de

son écriture (Figure 3a). Cette étiquette est destinée à identifier le contenu et la source du bloc mémoire auquel elle est rattachée lors de la lecture de ce dernier. Il faut donc être capable de créer une relation unique entre *M* et *T*. Pour cela, *T* est calculée avec des algorithmes de MAC (de l'anglais Message Authentication Code, c.à.d. Code d'Authentification de Message).

Le processus de vérification de l'intégrité a lieu lors de la lecture (Figure 3b) de *M*. Il consiste à rapatrier *T* correspondant au *M* chargé, puis à recalculer une étiquette de référence *T'* sur le SsP et enfin à comparer *T* à *T'*. Si les deux étiquettes sont différentes, cela signifie que la donnée a été modifiée. Généralement dans ce cas, un drapeau est prévu dans le dispositif pour prévenir l'UCT (Unité Centrale de Traitement) de ne pas traiter la donnée correspondante.



(a) Opération d'écriture: Calcul de l'étiquette



(b) Opération de lecture: Vérification de l'étiquette

Figure 3 Principe de contrôle de l'intégrité du contenu des mémoires externes

3.3. Etat de l'Art

Dans cette partie, les techniques permettant d'assurer aux données les deux services de sécurité, à savoir confidentialité et intégrité, sont tout d'abord décrites. Ensuite, les engins basés sur une de ces techniques et destinés à la protection des mémoires externes sont présentés.

3.3.1. Techniques Existantes

3.3.1.1. Composition Générique

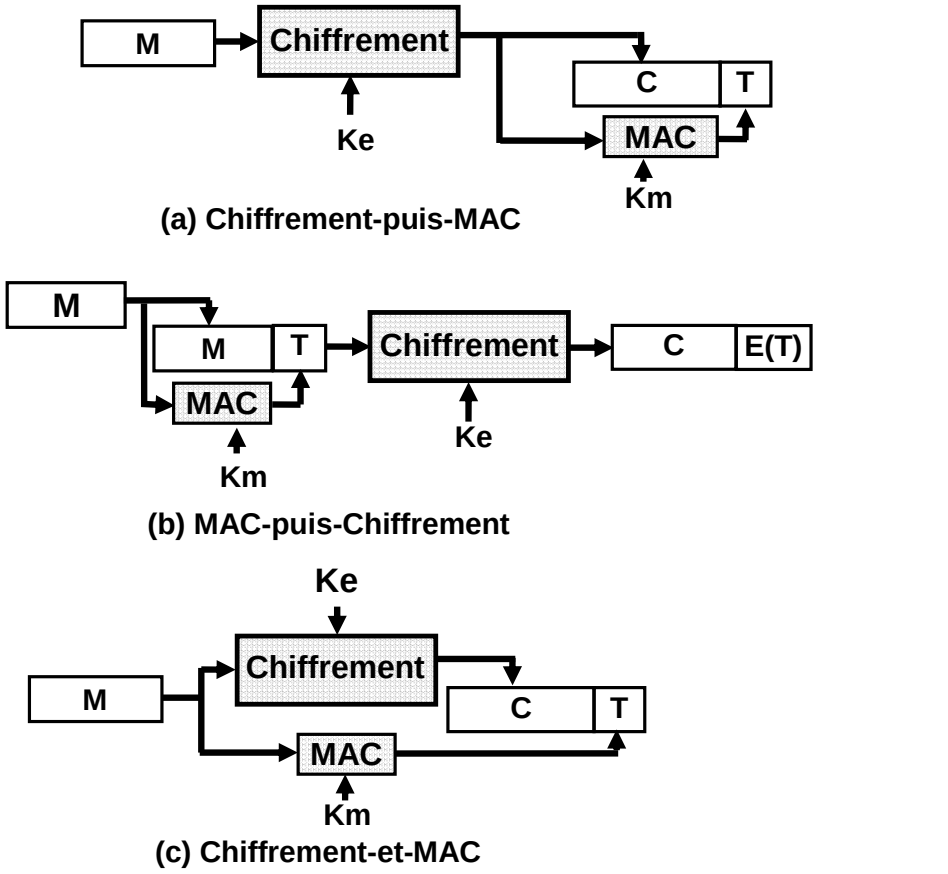


Figure 4 Approche conventionnelle pour assurer la confidentialité et l'intégrité de données

Dans notre contexte, les systèmes proposés dans l'état de l'art [12, 13, 37, 39, 40, 52, 14] utilisent l'approche conventionnelle pour assurer l'intégrité et la confidentialité des données. Cette approche consiste à implémenter un algorithme consacré au chiffrement (confidentialité) et un autre dédié au calcul de l'étiquette (intégrité). Trois possibilités d'implémentation s'offrent alors au concepteur. La première « Chiffrement-puis-MAC » (Figure 4a) consiste à chiffrer le texte en clair *M* puis à calculer l'étiquette *T* en appliquant le texte chiffré *C* à l'entrée de l'algorithme de MAC. Concernant la seconde méthode « MAC-puis-Chiffrement » (Figure 4b), *T* est d'abord obtenue en appliquant *M* en entrée du MAC,

puis est apposée à la fin de *M*. Finalement *M* et *T* sont chiffrés. La dernière possibilité « Chiffrement-et-MAC » (Figure 4c) consiste à procéder au calcul de *T* à partir de *M* et à le placer à la fin de *C* - texte résultant du chiffrement de *M*.

Le principal inconvénient d'une telle approche est son caractère non parallélisable lors des opérations de lecture ou d'écriture. En effet, concernant la méthode « Chiffrement-puis-MAC », lors d'une opération d'écriture, il faut attendre la fin du chiffrement de *M* pour commencer le calcul de l'étiquette. Pour les deux autres possibilités, il faut attendre lors d'une opération de lecture la fin du déchiffrement de *C* pour démarrer le calcul de l'étiquette étant donné que cette opération requiert le texte en clair.

Bellare et al. ont prouvé dans [50] que la méthode de composition générique la plus sécurisée est « Chiffrement-puis-MAC ».

3.3.1.2. Technique AREA (Ajout de Redondance Authentification Explicite)

La technique AREA (de l'anglais Added Redundancy Explicit Authentication) consiste à insérer de la redondance dans le message en clair avant chiffrement et de la vérifier lors du déchiffrement. De tels mécanismes sont construits à partir de modes de chiffrement avec propagation d'erreurs infinies en chiffrement et en déchiffrement. La propagation d'erreurs est infinie en chiffrement si un bloc de texte chiffré peut être exprimé en fonction de tous les blocs de texte en clair. De même, la propagation d'erreur est infinie en déchiffrement lorsqu'un bloc de texte en clair peut être exprimé en fonction de tous les blocs de texte chiffré. Des exemples de modes de chiffrement ayant de telles propriétés sont PCFB [58] et PCBC[53].

Pour authentifier un message en plus de le chiffrer en utilisant la technique AREA, une valeur choisie par l'émetteur du message est ajoutée à la fin du texte en clair. Son chiffrement dépendra de tous les blocs de texte en clair contenus dans le message. Cette valeur est envoyée en clair avec le message chiffré. De cette manière le destinataire peut vérifier que le déchiffrement du dernier bloc lui correspond bien.

Ce type de technique paraît très efficace car une seule passe sur les données est requise. Cependant, afin d'obtenir une propagation d'erreurs infinie, les opérations de chiffrement et de déchiffrement doivent être réalisées en série, rendant toute parallélisation impraticable.

3.3.1.3. Mode de Chiffrement Authentifié

Dans [50] et [53], il est montré qu'il peut être dangereux de garantir la confidentialité et l'intégrité des données avec des schémas de construction AREA et des compositions génériques. C'est pourquoi, la communauté de recherche en cryptographie travaille sur l'élaboration de modes de chiffrement authentifié à travers le processus de standardisation du NIST [54].

Deux modes ont récemment été recommandés par le NIST : CCM [55] (Counter CBC-MAC) et GCM [63] (Galois Counter Mode). CCM implémente une composition générique d'un mode compteur de chiffrement et de l'algorithme de MAC CBC-MAC. Son principal inconvénient est le fait que le calcul de l'étiquette avec un CBC-MAC est réalisé en série. Concernant le GCM, ce mode authentifié est extrêmement performant (mode compteur [16] et fonction de hash réalisé en moins d'un cycle) mais le NIST ne recommande pas son utilisation lorsque les étiquettes doivent être courtes. Dans notre contexte, il est préférable d'avoir des étiquettes les plus petites possibles car elles sont stockées en mémoire externe. Par ailleurs tous les modes proposés au processus de standardisation du NIST doivent être améliorés afin de répondre à la problématique des attaques par rejeu.

3.3.2. Engin de Protection du Contenu des Mémoires Externes

Les engins de protection mémoire assurant l'intégrité et la confidentialité des mémoires externes sont tous basés sur des compositions génériques. Les travaux présentés dans cette partie ont pour objectifs de fournir une architecture sécurisée ; cependant ils ont été conçus pour des applications différentes.

3.3.2.1. XOM (eXecute Only Memory)

XOM [39, 40] propose une architecture sécurisée avec des contre-mesures aux attaques logicielles et physiques pour des applications telles que la distribution sécurisée de propriétés intellectuelles ou la protection contre la copie de ces dernières. Les données stockées en mémoire externe sont chiffrées puis protégées en intégrité par un algorithme de MAC adressé (l'adresse est ajoutée en entrée de l'algorithme de MAC) (Chiffrement-puis-MAC). Cependant, l'architecture de l'engin de protection mémoire n'est pas décrite et les dégradations correspondantes ne sont pas détaillées. Concernant la sécurité du système, un

algorithme de MAC adressé seul protège contre les attaques par insertion aléatoire et par relogement mais pas contre le rejeu. En effet, l'étiquette ne dépendant que des données et de l'adresse, il est impossible de différencier deux données stockées à une même adresse si elles sont rejouées avec leur étiquette.

3.3.2.2. AEGIS

L'objectif du processeur AEGIS [12, 13, 37] est de fournir une architecture sécurisée généraliste avec une attention particulière portée aux applications de calculs distribués. Les attaques logicielles comme les attaques physiques sont considérées. Concernant ce dernier type d'attaque, Suh et al. implémentent une composition générique en Chiffrement-puis-MAC d'un « One Time Pad » (OTP, chiffrement par bloc en mode compteur ⇔ chiffrement par flux [16]) et d'un système d'arbre de hash (ou arbre de Merkle [49]). Les pertes en performance sont évaluées à 25% en moyenne pour un système embarqué et à 73% dans le pire cas. Le surcoût en mémoire externe est d'environ 40% (stockage de l'arbre de hash et de meta-data utiles pour l'implémentation sécurisée d'un OTP).

3.3.2.3. SP – Secret Protected

L'objectif des travaux menés par Lee et al. à travers l'architecture SP [52, 14] est de fournir une solution matérielle permettant de protéger les données critiques, telles les clefs de chiffrement, requises par les applications sécurisées. Par conséquent, seules ces données et le code permettant de les traiter, sont chiffrés et déchiffrés (et vérifiés) lors des échanges avec la mémoire externe. L'unité matérielle se chargeant de cette tâche est la composition générique en « Chiffrement-puis-MAC » d'un chiffrement par bloc en CBC et d'un CBC-MAC. Les pertes en performance sont évaluées à moins de 1%. Pour contrer les attaques par rejeu, l'implémentation d'un système d'arbre de Merkle est proposé mais n'est pas compris dans l'estimation donnée.

4. PE-ICE - de l'anglais Parallelized Encryption and Integrity Checking Engine

L'objectif de PE-ICE [61, 74, 75, 83, 84, 85] est de fournir une solution permettant d'assurer la confidentialité et l'intégrité des données transmises sur le bus

processeur/mémoire ainsi que d'optimiser, d'une part, les latences introduites par les mécanismes matériels de sécurité lors de tout type d'opérations requises par un processeur, et d'autre part, la surface de silicium requise pour leur implémentation. Pour cela, PE-ICE utilise uniquement un algorithme de chiffrement par bloc. La confidentialité est alors garantie par le chiffrement, tandis que la vérification d'intégrité s'appuie sur la propriété de diffusion des algorithmes de chiffrement par bloc et sur l'utilisation des ressources disponibles sur le SsP.

4.1. Confidentialité

PE-ICE assure la confidentialité des données par leur chiffrement à l'aide de l'algorithme Rijndael [15] ; c'est un algorithme de chiffrement par bloc qui traite des blocs et des clefs de taille comprise entre 128 bits et 256 bits et multiple de 32 bits. Pour PE-ICE, nous nous intéresserons aux versions de cet algorithme qui utilisent une clef de 128 bits et des tailles de blocs de 128, 160 et 192 bits. A noter que le Rijndael chiffrant des blocs de 128 bits avec une clef de 128, 192 ou 256 bits est l'algorithme standardisé par le NIST (National Institute of Standard and Technology) sous le nom AES [20] (Advanced Encryption Standard).

4.2. Vérification de l'Intégrité avec PE-ICE

4.2.1. La Propriété de Diffusion des Algorithmes de Chiffrement par Bloc

Le principe de contrôle de l'intégrité des données dans PE-ICE repose sur la propriété de diffusion des algorithmes de chiffrement par bloc. Cette propriété a été identifiée par Shannon comme nécessaire au chiffrement par bloc pour qu'il soit sécurisé. Théoriquement, un chiffrement par bloc doit être équivalent à une permutation de bits aléatoires avec des résultats en sortie équiprobable. En d'autres termes, après un tel chiffrement, chaque bit d'un bloc de texte chiffré *C* dépend de tous les bits du bloc de texte en clair *P* correspondant. Par conséquent si *P* est composé de deux parties *Pu* et *T*, il est impossible après chiffrement de distinguer dans *C* la partie correspondant à *Pu* de celle correspondant à *T* (Figure 5). Par ailleurs, si un seul bit est modifié dans *C*, il y a une très grande probabilité que, par exemple, la partie *T* résultante du déchiffrement soit erronée. Cette probabilité dépend de la taille de *T*.

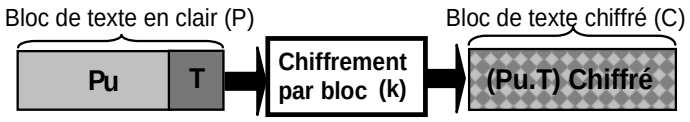
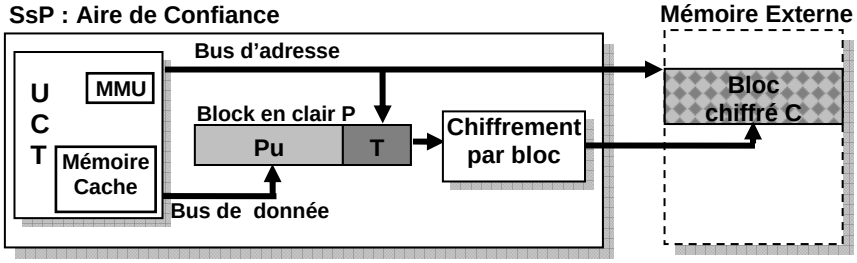


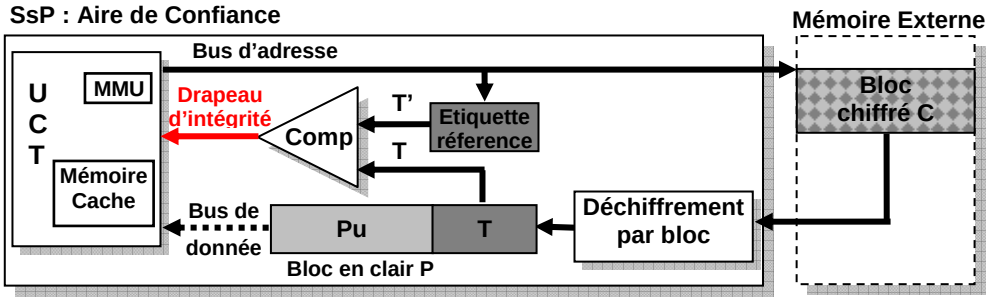
Figure 5 Propriété de diffusion des algorithmes de chiffrement par bloc

En supposant que *t* soit la longueur de *T* en bits et que *n* soit la taille d'un bloc de chiffrement, le nombre de textes en clair comportant la même partie *T* après le déchiffrement d'un bloc *C* corrompu est alors de 2^{n-t}. Par conséquent, la probabilité *D* que *T* ait la même valeur après déchiffrement est de $D = \frac{2^{n-t}}{2^n} = \frac{1}{2^t}$ et la probabilité $\overline{D}$ que *T* soit erronée est de $1 - D$.

4.2.2. Processus de Contrôle de l'Intégrité avec PE-ICE



(a) Opérations d'écriture : Insertion de l'étiquette et chiffrement



(b) Opérations de lecture : Déchiffrement et contrôle d'intégrité (Comparaison des étiquettes)

Figure 6 Principe de fonctionnement de PE-ICE

La propriété précédente est utilisée afin d'ajouter la capacité de vérification de l'intégrité au chiffrement par bloc. Ainsi le principe de contrôle d'intégrité du contenu mémoire avec PE-ICE fonctionne comme suit:

- *Opérations d'écriture* (Figure 6a) : La donnée à écrire *Pu* est concaténée avec une étiquette *T* afin de produire un bloc de texte en clair *P*. En théorie, une telle

étiquette doit être un nombre utilisé une seule fois – appelé nonce – pour une clef de chiffrement donnée et ne nécessite pas d'être calculée avec un algorithme spécifique ; elle peut par exemple être générée avec un compteur. Après chiffrement, une unique paire *Pu/T* est créée et le bloc chiffré résultant est écrit en mémoire externe.

- *Opérations de lecture* (Figure 6b): *C* est chargé sur le SsP et déchiffré. L'étiquette *T* provenant du bloc déchiffré est alors comparée à une étiquette de référence *T'* re-générée sur le SsP. Si *T* et *T'* ne sont pas égales, cela signifie qu'au moins un bit de *C* a été corrompu durant sa transmission sur le bus ou directement en mémoire. PE-ICE lève alors un drapeau pour prévenir le CPU afin qu'il n'exécute pas la donnée correspondante.

En résumé, PE-ICE peut être vu comme l'application de la technique AREA au niveau du bloc de chiffrement plutôt qu'au niveau du message, la propagation d'erreurs infinie étant assurée par la propriété de diffusion des algorithmes de chiffrement par bloc.

On notera que le mode de chiffrement utilisé est ECB – Electronic Code Book. Par ailleurs le bloc chiffré résultant est appelé un chunk. Un chunk peut également être défini comme le bloc atomique chargé lors d'opérations de lecture pour déchiffrement et vérification d'intégrité.

Le principe exposé au-dessus n'est une contre-mesure que contre les attaques par insertions aléatoires mais ne précise pas comment les étiquettes sont générées ni comment fonctionnent les contre-mesures contre le relogement et le rejeu. La section suivante traite de ces problématiques.

4.2.3. La Génération des Etiquettes

Dans notre contexte, le SsP est le seul à réaliser les processus de chiffrement et de déchiffrement. Par conséquent, le SsP doit conserver en mémoire la valeur de l'étiquette de chaque bloc chiffré entre le chiffrement et le déchiffrement ou alors doit être capable de le re-générer pour procéder au contrôle de l'intégrité. Le challenge est d'atteindre cet objectif en stockant le moins d'informations possibles sur l'étiquette dans le SsP.

La composition de l'étiquette est différente pour chaque type de données traitées par le processeur - donnée en lecture seule (RO) et donnée en écriture / lecture (RW) - et dépend de leurs propriétés respectives.

Les données RO sont écrites une seule fois en mémoire externe lors du chargement d'une application et ne sont pas modifiées au cours de l'exécution. Par conséquent, l'étiquette dédiée à un bloc de texte en clair RO peut être fixe pour une charge utile donnée. De plus, elle peut être publique car un attaquant a besoin de la clef secrète pour créer une paire charge utile / étiquette P_u/T acceptée. Cependant, cet attaquant ne doit pas pouvoir choisir l'étiquette de référence ou ne doit pas influencer sa génération. C'est pourquoi l'étiquette, pour les données RO, est constituée des bits les plus significatifs de l'adresse de chaque charge utile stockée en mémoire (Figure 7a). De cette manière, si un attaquant essaie une attaque par relogement l'adresse utilisée par le processeur (SsP) pour charger une donnée RO et pour générer l'étiquette de référence ne correspondra pas à l'étiquette *T* contenue dans le bloc déchiffré.

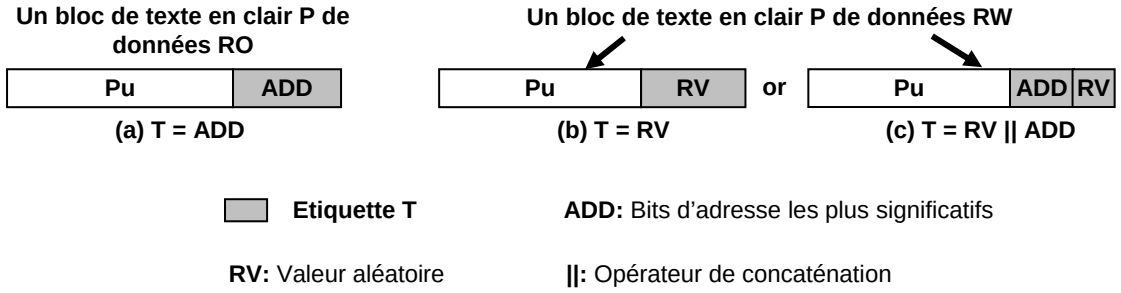


Figure 7 Composition d'un bloc de texte en clair et d'une étiquette avant chiffrement

Les données RW sont modifiées au cours de l'exécution d'une application et sont donc sensibles aux attaques par rejeu. L'utilisation de l'adresse seule comme étiquette ne suffit pas pour prévenir une telle attaque car les bits d'adresses ne rendent pas compte d'une modification de la charge utile entre deux écritures. Pour cette raison, l'étiquette est composée d'un vecteur *RV* qui est changé à chaque écriture (Figure 7b). *RV* est une valeur aléatoire générée sur le SsP. De cette manière l'étiquette est imprédictible pour un attaquant qui ne peut deviner lorsque deux blocs chiffrés ont la même étiquette à partir de l'observation de ces textes chiffrés. Cependant lors des opérations de lecture PE-ICE doit être capable de retrouver valeur aléatoire correspondant au bloc chargé – nommé par la suite la valeur aléatoire de référence *RV*' – afin de reconstituer l'étiquette de référence *T*' pour le processus de contrôle d' intégrité. Par ailleurs, l'ensemble des valeurs aléatoires de référence *RV*' doit être gardé

secret et infalsifiable. En effet, si *RV'* n'est pas gardé secret, un attaquant peut prévoir son attaque et rejouer un bloc chiffré s'il s'aperçoit que deux blocs sont authentifiés par la même *RV'*, et dans le cas où *RV'* est falsifiable, il peut choisir la donnée à rejouer en rejouant la *RV'* correspondant. Afin de remplir toutes ces conditions, l'ensemble des valeurs aléatoires de référence est stocké sur le SsP.

Cette étiquette permet de contrer tous les types d'attaques présentés dans le modèle de menace (insertion aléatoire, relogement et rejeu). Cependant, le niveau de sécurité correspondant dépend de la longueur de *RV*. Or plus *RV* est grand plus le surcoût en mémoire embarquée est important (stockage de RV'). C'est pourquoi nous proposons une seconde configuration pour la construction de l'étiquette où une valeur aléatoire (*RV*) est concaténée avec les bits les plus significatifs de l'adresse (Figure 7c). De cette manière il est possible de diminuer le surcoût en mémoire interne tout en maintenant une forte contre-mesure à l'encontre des attaques par relogement.

On notera que l'adresse utilisée dans la génération de l'étiquette est l'adresse physique.

4.3. Analyse de la Sécurité

Concernant les attaques passives remettant en cause la confidentialité des données, seules les attaques à texte chiffré connu – l'attaquant ne connaît que le texte chiffré – et à texte en clair connu – l'attaquant connaît le texte chiffré ou une partie du texte en clair, en l'occurrence l'adresse contenue dans l'étiquette pour ce dernier cas – peuvent être menées. C'est pourquoi le choix de l'algorithme de chiffrement par bloc utilisé pour l'implémentation de PE-ICE est primordial : cet algorithme doit être résistant à ces deux d'attaques. Les algorithmes de chiffrement par bloc utilisés pour configurer PE-ICE dans la suite de ce document remplissent ces conditions minimales.

Concernant les attaques actives définies dans le modèle de menace, la résistance de PE-ICE dépend de quatre paramètres : la longueur *t* en bit de l'étiquette, la longueur *r* en bit de la valeur aléatoire *RV*, le nombre de bits d'adresse *a* contenus dans l'étiquette et la taille du bloc chiffré *b* en octet de l'algorithme de chiffrement sous-jacent à PE-ICE.

Lorsque des bits d'adresse sont utilisés pour construire une étiquette, l'espace mémoire immunisé contre les attaques par relogement est calculé à partir de a et de b et est égal à $2^a \times b$. Cet espace mémoire est appelé par la suite un segment de relogement.

Nous estimons la résistance de PE-ICE aux attaques actives en chances de réussite pour un attaquant (Tableau 1).

Tableau 1 Robustesse de PE-ICE évaluée en chances de réussir une attaque active (insertion aléatoire, relogement ou rejeu)

Attaque		**Donnée RO**	**Donnée RW**	
			t = a + r	t = r
Attaque par insertion aléatoire		$\frac{1}{2^t}$	$\frac{1}{2^t}$	$\frac{1}{2^t}$
Attaque par relogement Taille d'un segment de relogement en octet: $2^a \times b$	A l'intérieur d'un segment de relogement	0	0	$\frac{1}{2^t}$
	En dehors d'un segment de relogement	0	$\frac{1}{2^r}$	
Attaque par rejeu		N/A	$\frac{1}{2^r}$	$\frac{1}{2^t}$

4.4. Exemple de Configuration

Dans cette section nous présentons une configuration de PE-ICE avec l'algorithme Rijndael qui traite des blocs de 160 bits et avec une clef de 128 bits (Rijn-160). Cette configuration est appelée PE-ICE-160. PE-ICE-160 est évalué en terme de surcoût mémoire, de latences introduites et de dégradations de performances. A titre de comparaison, la même évaluation est proposée pour un chiffrement AES (version standardisée de l'algorithme Rijndael – qui ne procure que la confidentialité des données) en mode ECB dans les mêmes conditions. Cet engin de chiffrement est nommé par la suite AES-ECB.

4.4.1. PE-ICE-160: Sécurité et Gestion des Clefs

Un chunk ou un bloc de texte en clair de PE-ICE-160 est composé de 128 bits de charge utile et de 32 bits d'étiquette comme décrit Figure 8.

L'étiquette est de 32 bits donc un attaquant à $1/2^{32}$ chances de réussir une attaque pas insertion aléatoire. Par ailleurs, pour les données RO, les 32 bits de l'étiquette sont des bits d'adresse. Par conséquent, si l'architecture de processeur utilisée est de 32 bits une telle étiquette protège tout l'espace d'adressage (4GB) contre le relogement de données RO et une seule clef est nécessaire par application.

Tableau 2 Robustesse de PE-ICE-160 évaluée en chances de réussir une attaque active (insertion aléatoire, relogement ou rejeu)

Attaques		**Données RO**	**Données RW**	
			t = a + r	t = r
Attaque par insertion aléatoire		$\frac{1}{2^{32}}$	$\frac{1}{2^{32}}$	$\frac{1}{2^{32}}$
Attaque par relogement Taille d'un segment de relogement en octet: $2^a \times b$	A l'intérieur d'un segment de relogement	0	0	$\frac{1}{2^{32}}$
	En dehors d'un segment de relogement	0	$\frac{1}{2^r}$	
Attaque par rejeu		N/A	$\frac{1}{2^r}$	$\frac{1}{2^{32}}$

Pour les données RW, les chances de réussir pour un attaquant dépendent de la définition par le concepteur de PE-ICE des paramètres a et r. Dans le cas où $t = r = 32$, tous les types d'attaques actives (insertion aléatoire, relogement, rejeu) ont la même chance de réussir : $1/2^{32}$. Dans le cas où $t = a + r = 32$, les chances de réussir une attaque par insertion aléatoire ou par rejeu sont respectivement de $1/2^{32}$ et de $1/2^r$. Une seule clef est également requise pour les données RW.

Le Tableau 2 résume la résistance fournie par PE-ICE-160 aux attaques actives (insertion aléatoires, relogement et rejeu).

4.4.2. Latences

Les latences introduites par PE-ICE-160 entre la mémoire cache et le contrôleur mémoire sont évaluées sur un bus AHB (de l'anglais Advanced High-perforance Bus [76]) de 32 bits. Ces latences sont données dans le Tableau 3 pour toutes les opérations demandées pas un processeur ARM9E [89] et pour deux valeurs différentes du rapport R entre la fréquence de l'algorithme de chiffrement sous-jacent (Rijn-160) et la fréquence du bus AHB : R = 1 et R = 2. Une évaluation des latences introduites par un chiffrement AES (confidentialité uniquement) dans les mêmes conditions est également indiquée dans le Tableau 3.

Tableau 3 Latences introduites par PE-ICE-160 et par un engin AES-ECB sur un bus AHB pour toutes les opérations requises par un ARM9E au cours de l'exécution d'une application

Opérations	AES-ECB		PE-ICE-160			
	Latences (cycles AHB)		Latences (cycles AHB)		Surcoût vs. AES-ECB	
	R = 1	R = 2	R = 1	R = 2	R = 1	R = 2
Ecriture de 8 à 32 bits	38	28	42	30	10,5%	7%
Lecture de 8 à 32 bits	15	10	17	11	13,5%	10%
Ecriture de 4 mots de 32 bits	15	10	17	11	13,5%	10%
Lecture de 4 mots de 32 bits	15	10	17	11	13,5%	10%
Ecriture de 8 mots de 32 bits	15	10	18	12	20%	20%
Lecture de 8 mots de 32 bits	15	10	18	12	20%	20%
Ecriture de 16 mots de 32 bits	15	10	20	14	33,5%	40%
Lecture de 16 mots de 32 bits	15	10	20	14	33,5%	40%

4.4.3. Ressources Matérielles Requises

Les ressources matérielles requises pour l'implémentation de PE-ICE peuvent être estimées en nombre de cœurs de chiffrement Rijn-160 ou AES[32] à prévoir pour atteindre le débit maximum à la sortie de PE-ICE ou de l'AES-ECB en lecture ou en écriture.

Pour AES-ECB, le débit maximum en sortie est défini par celui du bus AHB qui est de 32 bits par cycle. Sachant que l'AES traite des blocs de 128 bits et que sa latence intrinsèque en mode ECB [23] vue du bus AHB est de 11 cycles pour R = 1 et de 6 cycles pour R = 2, respectivement trois et deux cœurs AES doivent être implémentés pour atteindre le débit optimum de 32 bits par cycle.

Pour PE-ICE-160, le débit est plus important en écriture qu'en lecture car l'étiquette de référence est collectée en parallèle de la charge utile, ce qui n'est pas le cas en lecture où les données arrivent en série de la mémoire externe. Le calcul du nombre de cœurs de chiffrement se fait donc par rapport au débit requis en écriture. PE-ICE-160 doit être capable de collecter 128 bits de charge utile tous les 4 cycles de bus sachant qu'un chiffrement avec

[32] Etant basé sur le même algorithme, un cœur Rijn-160 et un cœur AES requièrent les mêmes ressources matérielles. Par conséquent, dans la suite du document nous ne ferons référence qu'à des cœurs AES.

Rijn-160 demande 12 cycles de latences (vues du bus AHB) pour R = 1 et 6 cycles pour R = 2. Le nombre de cœurs Rijn-160 est alors de trois pour R = 1 et de deux pour R = 2.

En considérant les évaluations en nombre de portes d'un cœur AES (et de son cœur d'expansion de clef) de la société Ocean Logic [80], l'implémentation matérielle de PE-ICE-160 et d'un AES-ECB demande une surface de 80 K portes en technologie 0,18µ.

4.4.4. Dégradation des Performances durant l'Exécution

Les dégradations engendrées par un chiffrement AES-ECB et par PE-ICE-160 ont été évaluées en utilisant l'outil de simulation SoCDesigner [81] fourni par ARM. Le cœur de processeur simulé est le ARM9E et les paramètres architecturaux choisis sont indiqués dans le Tableau 5. Les latences d'accès mémoire dites de base – sans mécanismes matériels de sécurité - en lecture et en écriture émanent de la documentation d'un contrôleur mémoire compatible AHB : le PL172 [82]. Les latences les plus petites ont été choisies afin d'estimer la dégradation de performances impliquée par PE-ICE dans le pire cas.

Huit logiciels (MP3 player, Huffman, CJPEG, DJPEG, ADPCM, FingerPrint, DES, MP2Audio) destinés aux systèmes embarqués ont été utilisés afin d'évaluer les dégradations engendrées par les différentes versions de PE-ICE et par un chiffrement AES-ECB. Deux cas ont été considérés : une cache destinée aux données de 4 Ko et de 128 Ko ; le taux de défauts de cache relatif à chaque application d'évaluation utilisée est indiqué Figure 9. Les résultats de simulation (Tableau 4 : paramètres architecturaux) de l'AES-ECB et de PE-ICE-160 sont donnés Figure 10 en les normalisant par rapport aux résultats de simulation obtenus sur une plateforme conçue sans mécanismes de sécurité (Figure 8). Nous indiquons également en Figure 10 les évaluations d'autres configurations de PE-ICE : PE-ICE-128 (configuration avec l'AES comme algorithme de chiffrement par bloc sous-jacent) et PE-ICE-192 (configuration avec le Rijndael traitant des blocs de 192 bits).

Tableau 4 Paramètres architecturaux choisis pour les simulations

Processeur	ARM926EJ-S
Bande passante du bus processeur mémoire	32 bits
F_{CPU} / F_{AHB}	2
Cache line	256 bits
$R = F_{AES} / F_{AHB}$	2
Latence d'accès mémoire en lecture de "Base" (en cycles de bus AHB)	9
Latence d'accès mémoire en écriture de "Base" (en cycles de bus AHB)	1

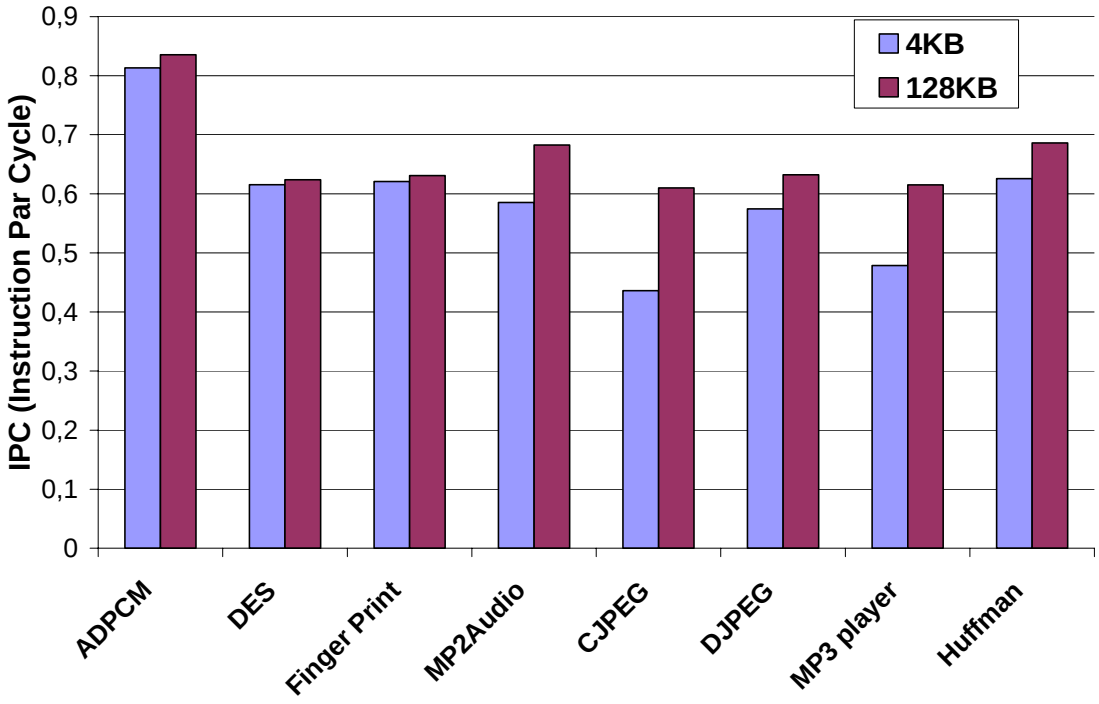


Figure 8 Résultats de simulations des applications d'évaluation sur une plateforme définie sans mécanisme de sécurité pour deux tailles de mémoire cache dédiée aux données

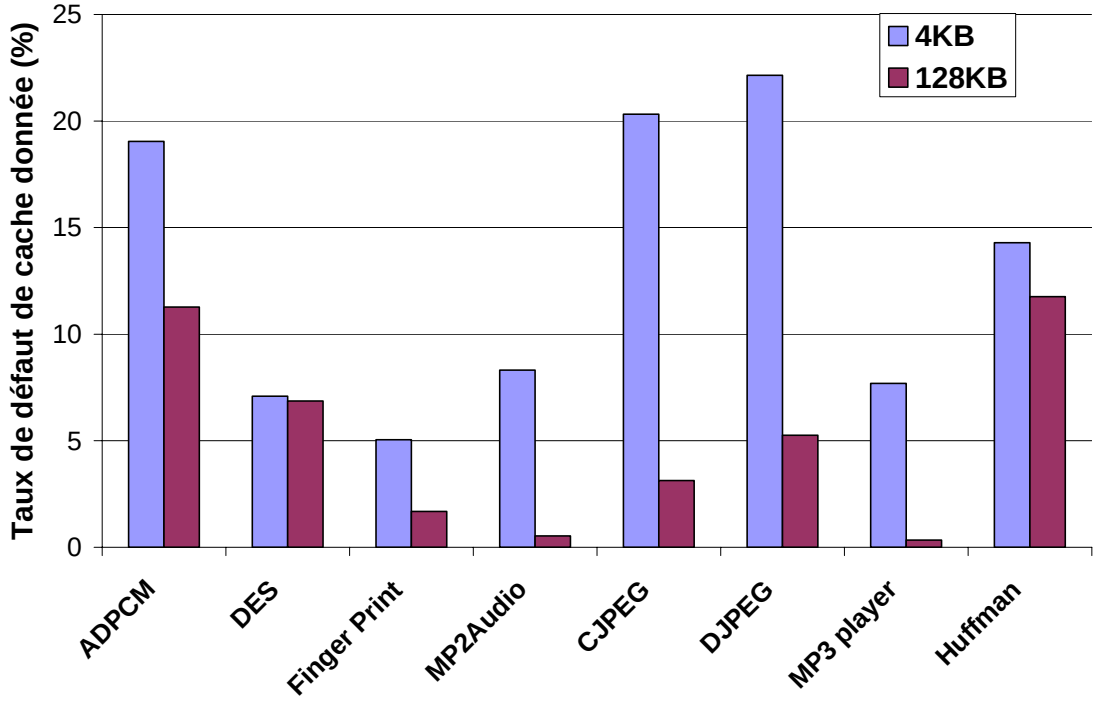


Figure 9 Taux de défauts de cache de données lors de l'exécution des applications utilisées pour l'évaluation des dégradations des performances engendrées par les mécanismes matériels de sécurité.

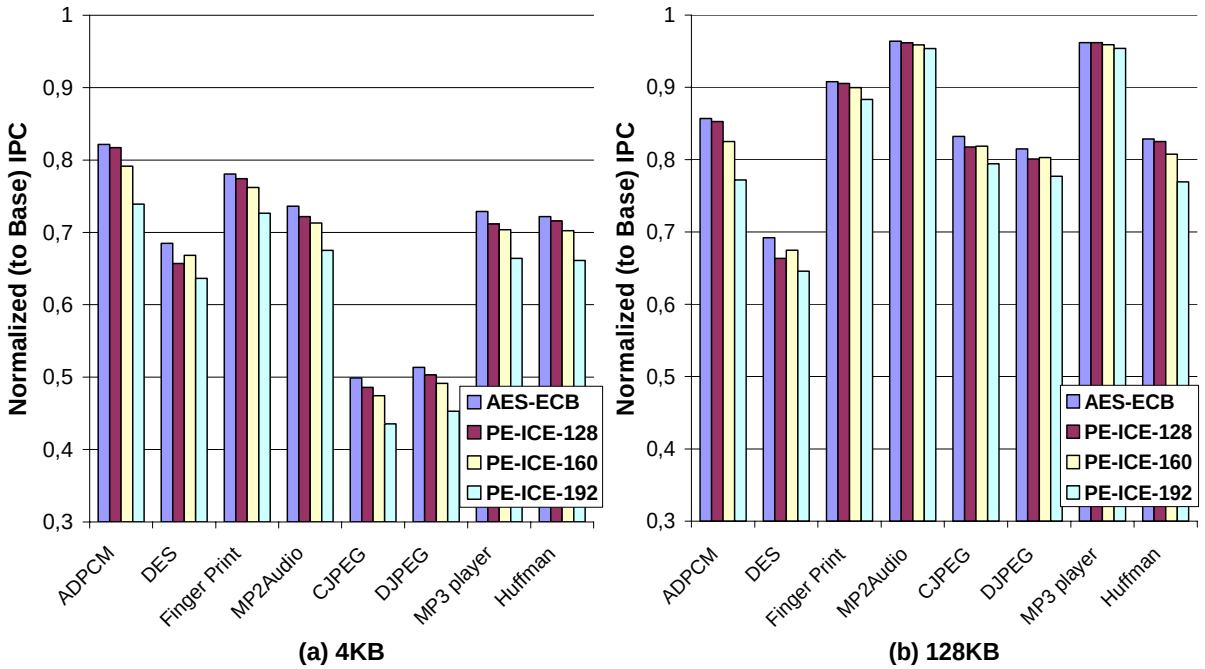


Figure 10 Dégradation engendrée par les mécanismes de sécurité (PE-ICE et AES-ECB) pour deux tailles de mémoire cache dédiée aux données. Les résultats sont donnés pour différentes versions de PE-ICE: PE-ICE-128 (AES), PE-ICE-160 (Rijn-160) et PE-ICE-192 (Rijn-192).

La Figure 10 montre clairement que les dégradations de performances sont principalement dues au chiffrement car les différents engins de sécurité (PE-ICE et AES-ECB) implémentent tous un chiffrement ECB et engendrent une perte en performance relativement similaire. Le chiffrement AES-ECB implique une dégradation de 50% dans le pire cas (CJPEG – 4KB) et de 31,5% et de 14,3% en moyenne respectivement pour une cache de donnée de 4 KB et de 128 KB. Augmenter la taille de la mémoire cache diminue logiquement les dégradations en réduisant le nombre d'accès à la mémoire externe.

Concernant PE-ICE, les dégradations engendrées par rapport à un chiffrement AES-ECB correspondant au contrôle d'intégrité des données ont été évaluées en normalisant les résultats des différentes versions proposées (PE-ICE-128, PE-ICE-160 et PE-ICE-192) à ceux obtenus avec l'engin AES-ECB. La Figure 11 montre que le surcoût en performance est négligeable par rapport à un chiffrement des données seules : en moyenne PE-ICE-160 rajoute une dégradation de 3,3% et de 1,7% pour une cache de données de respectivement 4 Ko et 128 Ko.

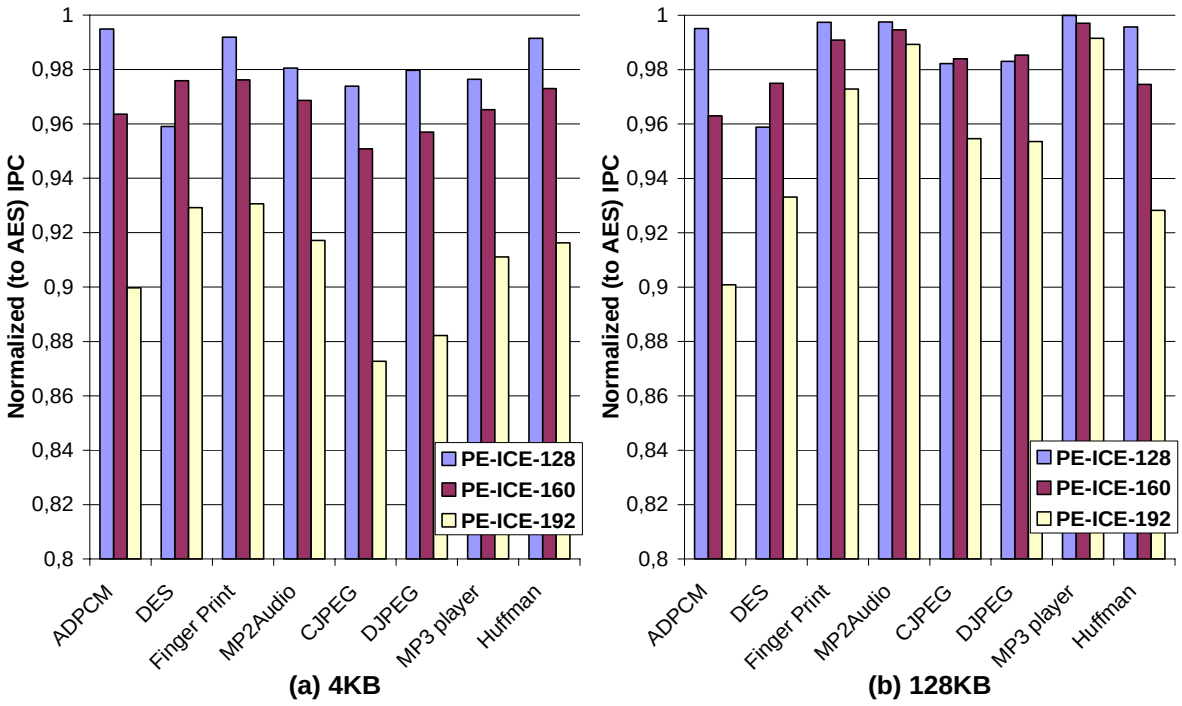


Figure 11 Dégradation des performances engendrée par le mécanisme de contrôle de PE-ICE. Les résultats sont donnés pour différentes versions de PE-ICE: PE-ICE-128 (AES), PE-ICE-160 (Rijn-160) et PE-ICE-192 (Rijn-192) et sont normalisés par rapport au chiffrement AES-ECB

5. Conclusion

L'objectif des mécanismes de protection contre les attaques physiques au niveau système (observation du bus ou corruption du contenu mémoire) est de fournir aux applications exécutées sur un processeur un environnement confidentiel, authentifié et résistant aux falsifications. Un tel environnement requiert d'assurer la confidentialité et l'intégrité des données transférées entre un processeur et sa mémoire externe.

La méthode conventionnelle utilisée dans les Systèmes sur Puces (SsP) est de dédier un engin matériel à chaque objectif sécuritaire : confidentialité et intégrité. Cette approche mène à une utilisation non optimisée des ressources matérielles et implique des latences générées par les calculs sous-jacents au chiffrement/déchiffrement et aux calculs d'étiquettes, non parrallélisables.

Dans cette thèse, nous proposons d'explorer l'ajout de redondances et d'aléas dans chaque bloc de texte en clair avant leur chiffrement - par bloc - afin d'assurer efficacement la confidentialité et l'intégrité des données transitant sur le bus processeur mémoire des

systèmes embarqués. En vérifiant la redondance ou l'aléa après déchiffrement, nous ajoutons la capacité de contrôle de l'intégrité au chiffrement par bloc. De cette manière, un unique algorithme de chiffrement est implémenté pour garantir les deux services de sécurité. De plus, un point clef de PE-ICE est le fait que la génération de l'étiquette se fait indépendamment des données que ce dernier protège. Il en résulte que PE-ICE offre les avantages suivants :

- Optimisation des latences : Les processus de chiffrement (respectivement de déchiffrement) et de calcul de l'étiquette (respectivement de contrôle d'intégrité) sont réalisés en parallèle, permettant d'optimiser les latences d'accès mémoire en écriture et en lecture.
- Optimisation des ressources matérielles : le même matériel est utilisé pour assurer la confidentialité et l'intégrité des données.

Nous avons également mis en place un mécanisme permettant de stocker les valeurs aléatoires de référence en mémoire externe (PRV-Tree ; voir version de la thèse en anglais) afin d'annuler le surcoût engendré par leur stockage sur le SsP. Par ailleurs nous implémentons actuellement PE-ICE sur un processeur LEON2 [77].

References

[1] Paul Kocher, Ruby B. Lee, Gary McGraw, Anand Raghunathan, and Srivaths 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] C. Tang, "Summary of Mobile Threats for Year 2005", available at: http://www.it-observer.com/pdf/dl/mobile_threat_sum.pdf.

[4] Tiago Alves and Don Felton. "Trustzone: Integrated hardware and software security", ARM white paper, July 2004.

[5] 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.

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

[7] 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.

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

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

[10] R. M. Best, “Crypto Microprocessor that Executes Enciphered Programs”, U.S. Patent No. 4 465 901, August 14, 1984.

[11] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark 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.

[12] 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.

[13] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, “Efficient Memory Integrity Verification and Encryption for Secure Processors”, in Proceedings of the 36th Int’l Symposium on Microarchitecture, pages 339–350, December 2003.

[14] Ruby B. Lee, Peter C. S. Kwan, John Patrick McGregor, Jeffrey Dwoskin, and Zhenghong 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.

[15] Joan Daemen, Vincent Rijmen, “AES Proposal: Rijndael”, March 1999, available at:
http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.

[16] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. Handbook of Applied Cryptography. CRC Press, 1996.

[17] Auguste Kerckhoffs, “La cryptographie militaire”, Journal des sciences militaires, vol. IX, pp. 5–38, Jan. 1883, pp. 161–191, February 1883.

[18] G.S. Vernam, “Cipher printing telegraph systems for secret wire and radio telegraphic communications”, Journal of the American Institute of Electrical Engineers, vol. 45 (1926), pp. 109--115 or US patent #1,310,719.

[19] Claude Shannon, “Communication theory of secrecy systems”, Bell System Technical Journal, 28, 1949.

[20] National Institute of Science and Technology (NIST), FIPS PUB 197: “Advanced Encryption Standard (AES)”, November 2001.

[21] National Institute of Science and Technology (NIST), FIPS PUB 46.2: “Data Encryption Standard (DES)”, December 1993.

[22] Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design by the Electronic Frontier Foundation (ISBN 1565925203).

[23] Alireza Hodjat, David Hwang, Bo-Cheng Lai, Kris Tiri and Ingrid 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: 60-63.

[24] A. W. Dent and C. J. Mitchell, “User's Guide to Cryptography and Standards”, Artech House, 2005.

[25] W. Diffie and M. E. Hellman, “New Directions in Cryptography”, IEEE Transactions on Information Theory, vol. IT-22, November 1976, pp: 644-654.

[26] R. Rivest, A. Shamir and L. Adleman, “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”, Communications of the ACM, Vol. 21 (2), pp.120–126. 1978.

[27] C. Mclvor, M. McLoone and J.V. McCanny, “Fast Montgomery modular multiplication and RSA cryptographic processor architectures”, in Proceedings of the 37th IEEE Computer Society Asilomar Conference on Signals, Systems and Computers, Monterey, USA, pp379-384, November 2003.

[28] National Institute of Science and Technology (NIST), FIPS PUB 180-2: “Secure Hash Standard”, August 2002.

[29] Youtao Zhang, Jun Yang, Yongjing Lin, Lan Gao, "Architectural Support for Protecting User Privacy on Trusted Processors", The Workshop on Architectural Support for Security and Anti-Virus, In conjunction with the 11th ASPLOS, Boston, MA, October 2004.

[30] DG Abraham, GM Dolan, GP Double and JV Stevens, “Transaction Security System”, in IBM Systems Journal vol.30 no2 (1991) pp 206-229.

[31] M. G. Kuhn, “Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP”, IEEE Trans. Comput., vol. 47, pp. 1153–1157, October. 1998.

[32] David A. Patterson and John L. Hennessy “Computer Organization and Design: The Hardware/Software Interface”, Morgan Kaufmann Publishers, 1997.

[33] John L. Hennessy and David A. Patterson, “Computer Architecture: A Quantitative Approach”, Morgan Kaufmann Publishers, 2002

[34] Tanguy Gilmont, Jean-Didier Legat and Jean-Jacques Quisquater, “An Architecture of Security Management Unit for Safe Hosting of Multiple Agents”, pages 79-82, November 1998.

[35] Tanguy Gilmont, Jean-Didier Legat and Jean-Jacques Quisquater, “Hardware Security for Software Privacy Support”, In IEE Electronics Letters, Volume 35, pages 2096-2098, January 1999.

[36] Tanguy Gilmont, Jean-Didier Legat and Jean-Jacques Quisquater, “Enhancing the Security in the Management Unit”, in Proceedings of the 25th EuroMicro Conference, pages 449-456, January 1999.

[37] Gookwon Edward Suh, “AEGIS: A Single-Chip Secure Processor”, PhD thesis, Massachusetts Institute of Technology, September 2005.

[38] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas 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.

[39] David Lie, Chandramohan Thekkath and Mark Horowitz, "Implementing an Untrusted Operating System on Trusted Hardware", in Proceedings of the 19th ACM Symposium on Operating Systems Principles, October, 2003.

[40] David Lie, "Architectural Support for Copy and Tamper-Resistant Software", Ph.D Thesis, Stanford University, December 2003.

[41] Jun Yang, Lan Gao, and Youtao Zhang, "Improving Memory Encryption Performance in Secure Processors", IEEE Transactions on Computers. pp. 630-640, Vol. 54, No. 5, May 2005.

[42] "DS5250 - High-Speed Secure Microcontroller" available at: http://www.maxim-ic.com/products/microcontrollers/secure/

[43] Richard Takahashi and Daniel N. Heer "Secure Memory Management Unit for Microprocessor", U.S. Patent (from VLSI Technology, Inc.) No. 5 825 878, October 20, 1998.

[44] Brant Candelore and Eric Sprunk. "Secure processor with external memory using block chaining and block re-ordering", U.S. Patent (from General Instrument Corporation) No. 6 061 449, May 9, 2000.

[45] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, C. Anguille, Michel Bardouillet, Christian Buatois, Jean-Baptiste Rigaud, "Hardware Engines for Bus Encryption: A Survey of Existing Techniques", DATE 2005, p. 40-45.

[46] Jun Yang, Youtao Zhang and Lan Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering", ACM/IEEE 36th International Symposium on Microarchitecture, pp. 351-360, December 2003.

[47] Blaise Gassend, G. Edward Suh, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas, "Caches and Merkle Trees for Efficient Memory Integrity Verification", In Proceedings of Ninth International Symposium on High Performance Computer Architecture, February 2003.

[48] David Lie, John Mitchell, Chandramohan Thekkath and Mark Horowitz. "Specifying and Verifying Hardware for Tamper-Resistant Software" In

Proceedings of the 2003 IEEE Symposium on Security and Privacy. May, 2003.

[49] R. C. Merkle, “Protocols for Public Key Cryptography” In IEEE Symp. on Security and Privacy, pages 122–134, 1980.

[50] 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.

[51] NIST Special Publication 800-67: "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", available at: http://csrc.nist.gov/publications/nistpubs/800-67/SP800-67.pdf

[52] John Patrick McGregor and Ruby B. Lee, “Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing”, Computer Architecture News, vol. 33., no. 1, pp. 16-26, March 2005, and Proceedings of the Workshop on Architectural Support for Security and Antivirus (WASSA) held in conjunction with ASPLOS-XI, October 2004.

[53] Chris J. Mitchell, “Cryptanalysis of Two Variants of PCBC Mode When Used for Message Integrity”, ACISP 2005: 560-571.

[54] C S R C (Computer Security Resource Center) - Modes of Operation at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/

[55] D. Whiting, R. Housley, and N Ferguson, Counter with CBC-MAC (CCM), available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ccm/ccm.pdf.

[56] M. Bellare, P. Rogaway, and D. Wagner, “EAX: A Conventional Authenticated-Encryption Mode”, Cryptology ePrint Archive, Report 2003/069, 2003, available at: http://eprint.iacr.org/2003/069.

[57] T. Iwata and K. Kurosawa, “OMAC: One-Key CBC MAC”, March 2003, available at: http://eprint.iacr.org/2002/180.pdf

[58] H. Hellström, "Propagating Cipher Feedback", 2001, available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/pcfb/pcfb-spec.pdf.

[59] C. S. Jutla, "Encryption Modes with Almost Free Message Integrity", available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/iacbc/iacbc-spec.pdf.

[60] V. D. Gligor and P. Donescu, "Fast encryption and authentication: XCBC encryption and XECB authentication Modes", 2001, available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/xcbc/xcbc-spec.pdf.

[61] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin and M. Bardouillet, "MC-VIP: Moteur de Chiffrement et de Vérification d'Intégrité Parrallélisé", 9iemes édition des Journées Nationales du Réseau Doctoral en Microélectronique JNRDM, May 2006.

[62] Phillip Rogaway, Mihir Bellare, John Black and Ted Krovetz, "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption", 2001, available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ocb/ocb-spec.pdf.

[63] David A. McGrew and John Viega, "The Galois/Counter Mode of Operation (GCM)", March 2005 available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-revised-spec.pdf.

[64] M. N. Wegman and J. L. Carter, "New Hash Functions and their Use in Authentication and Set Equality", Journal of Computer and System Sciences, 22:265–279, 1981.

[65] Bo Yang, Sambit Mishra, and Ramesh Karri, "High Speed Architecture for Galois/Counter Mode of Operation (GCM). Cryptology ePrint Archive, Report 2005-156, May 2005, available at: http://eprint.iacr.org/2005/146.

[66] T. Kohno, J. Viega, and D. Whiting, "The CWC Authenticated Encryption (Associated Data) Mode", May 2003, available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/cwc/cwc-spec.pdf.

[67] Dana Neustadter, Michael Bowler, Tom St. Denis and Mike Borza, “Comments on CWC and GCM Modes for Authenticated Encryption”, June 2005, available at: http://csrc.nist.gov/CryptoToolkit/modes/comments/CWC-GCM/Elliptic-Semiconductor.pdf.

[68] T. Kohno, J. Viega, and D. Whiting, “The CWC-AES Dual-use Mode”, Internet Draft, Crypto Forum Research Group, May 20, 2003. Work in progress, available at: http://www.zork.org/cwc/draft-irtf-cfrg-cwc-01.txt.

[69] W. Erik Anderson, Cheryl L. Beaver, Timothy J. Draelos, Richard C. Schroeppel and Mark D. Torgerson, "Cipher-State (CS) Mode of Operation for AES" available at: http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/cs/cs-spec.pdf.

[70] http://www.uclinux.org/.

[71] Morris Dworkin, NIST Special Publication 800-38C: “Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality”, May 2004, available at: http://csrc.nist.gov/CryptoToolkit/modes/800-38_Series_Publications/SP800-38C.pdf.

[72] Morris Dworkin, NIST Special Publication 800-38D DRAFT (April, 2006): “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication”, available at: http://www.csrc.nist.gov/publications/drafts/Draft-NIST_SP800-38D_Public_Comment.pdf.

[73] Niels Ferguson, "Authentication Weaknesses in GCM", May 2005, available at http://csrc.nist.gov/CryptoToolkit/modes/comments/CWC-GCM/Ferguson2.pdf.

[74] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and J.B. Rigaud, “How to Add the Integrity Checking Capability to Block Encryption Algorithms”, In Proc. Of the IEEE International Conference, PhD Research In Microelectronics and Electronics, PRIME, June 2006.

[75] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin and M. Bardouillet, "PE-ICE: Parallelized Encryption and Integrity Checking Engine", In Proceedings of the 9th IEEE workshop on Design & Diagnostics of Electronic Circuits & Systems, DDECS, April 2006.

[76] AMBA (Advanced Microcontroller Bus Architecture) Specification available at: http://www.gaisler.com/doc/amba.pdf.

[77] http://www.gaisler.com/.

[78] http://www.arm.com/pdfs/ARM9E_flyer_063_4.pdf

[79] http://www.model.com/products/default.asp.

[80] AES Core family Rev 1.5 - available at: http://www.ocean-logic.com/pub/OL_AES.pdf.

[81] http://www.arm.com/products/DevTools/MaxSim.html

[82] 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.

[83] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and A. Martinez, "A Comparison of Two Approaches Providing Encryption and Authentication on a Processor Memory Bus", In Proc. Of the Power And Timing Modeling, Optimization and Simulation International workshop, PATMOS September 2006.

[84] 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", In Proceedings of the 43rd Design Automation Conference DAC July 2006.

[85] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and A. Martinez, "Efficient Combination of Encryption and Integrity Checking for Embedded Systems", In Proceedings of the Reconfigurable Communication-centric System on Chip workshop, ReCoSoC July 2006.

[86] The Embedded Microprocessor Benchmark Consortium (EEMBC). http://www.eembc.org/.

[87] Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, and Chenghuai Lu, "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems", in Proceedings of the International Conference on Parallel Architecture and Compilation Techniques, pages 123–134, PACT September 2004.

[88] Youtao Zhang, Lan Gao, Jun Yang, Xiangyu Zhang, and Rajiv Gupta, "SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors", in Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pages 352–362, February 2005.

[89] ARM926EJ-S: Technical Reference Manual available at: http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[90] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication", Advances in Cryptology - Crypto 96 Proceedings, Lecture Notes in Computer Science Vol. 1109, pp1–15, N. Koblitz ed, Springer-Verlag, 1996.

Bibliography Relative to the Study

Publications in International Conference Proceedings

[DATE05] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, C. Anguille, Michel Bardouillet, Christian Buatois, Jean-Baptiste Rigaud, "Hardware Engines for Bus Encryption: A Survey of Existing Techniques", DATE 2005, p. 40-45.

[DDECS06] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin and M. Bardouillet, "PE-ICE: Parallelized Encryption and Integrity Checking Engine", In Proceedings of the 9th IEEE workshop on Design & Diagnostics of Electronic Circuits & Systems, DDECS, April 2006.

[PRIME06] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and J.B. Rigaud, "How to Add the Integrity Checking Capability to Block Encryption Algorithms", In Proc. Of the IEEE International Conference, PhD Research In Microelectronics and Electronics, PRIME, June 2006. **Golden Leaf**.

[DAC06] 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", In Proceedings of the 43rd Design Automation Conference DAC July 2006.

[RECOS06] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and A. Martinez, "Efficient Combination of Encryption and Integrity Checking for Embedded Systems", In Proceedings of the Reconfigurable Communication-centric System on Chip workshop, ReCoSoC July 2006.

[PATM06] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet and A. Martinez, "A Comparison of Two Approaches Providing Encryption and Authentication on a Processor Memory Bus", In Proc. Of the Power And Timing Modeling, Optimization and Simulation International workshop, PATMOS September 2006.

Patent

Patent pending:

PE-ICE-OTP: Stream Encryption Combined to a Full Diffusion Function to Provide Data Integrity in Addition to Confidentiality

Publications in National Conference Procedings

[JNRDM06] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin and M. Bardouillet, "MC-VIP: Moteur de Chiffrement et de Vérification d'Intégrité Parrallélisé", 9iemes édition des Journées Nationales du Réseau Doctoral en Microélectronique JNRDM, May 2006.

MECANISMES MATERIELS POUR DES TRANSFERTS PROCESSEUR MEMOIRE SECURISES DANS LES SYSTEMES EMBARQUES

RESUME

Les systèmes embarqués actuels (téléphone portable, assistant personnel…) ne sont pas considérés comme des hôtes de confiance car toute personne y ayant accès, sont des attaquants potentiels. Les données contenues dans ces systèmes peuvent être sensibles (données privées du propriétaire, mot de passe, code d'un logiciel…) et sont généralement échangées en clair entre le Système sur Puces (SoC – System on Chip) et la mémoire dans laquelle elles sont stockées. Le bus qui relie ces deux entités constitue donc un point faible : un attaquant peut observer ce bus et récupérer le contenu de la mémoire, ou bien a la possibilité d'insérer du code afin d'altérer le fonctionnement d'une application s'exécutant sur le système. Afin de prévenir ce type d'attaque, des mécanismes matériels doivent être mis en place afin d'assurer la confidentialité et l'intégrité des données. L'approche conventionnelle pour atteindre cet objectif est de concevoir un mécanisme matériel pour chaque service de sécurité (confidentialité et intégrité). Cette approche peut être implantée de manière sécurisée mais empêche toute parallélisation des calculs sous-jacents.

Les travaux menés au cours de cette thèse ont dans un premier temps, consisté à faire une étude des techniques existantes permettant d'assurer la confidentialité et l'intégrité des données. Dans un deuxième temps, nous avons proposé deux mécanismes matériels destinés à la sécurisation des transactions entre un processeur et sa mémoire. Un moteur de chiffrement et de contrôle d'intégrité parallélisé, PE-ICE (Parallelized Encryption and Integrity Checking Engine) a été conçu. PE-ICE permet une parallélisation totale des opérations relatives à la sécurité aussi bien en écriture qu'en lecture de données en mémoire. Par ailleurs, une technique basée sur une structure d'arbre (PRV-Tree – PE-ICE protected Reference Values) comportant la même propriété de parallélisation totale, a été spécifiée afin de réduire le surcoût en mémoire interne impliqué par les mécanismes de sécurité.

MOTS-CLES: Architecture des processeurs, Systèmes embarqués, Sécurité, Confidentialité, Intégrité, Authentification, Cryptographie, Systèmes sur Puce, Observation de bus, Attaques, Mémoire, Performance.

HARDWARE MECHANISMS FOR SECURED PROCESSOR-MEMORY TRANSACTIONS IN EMBEDDED SYSTEMS

ABSTRACT

Today's embedded systems are considered as non trusted hosts since the owner, or anyone else who succeeds in getting access, is a potential adversary. The bus between the System on Chip (SoC) and the external memory is one of the weakest points of such systems because external memories contain sensitive data (end users private data, software code…) which are usually exchanged in clear form over the bus. Therefore an adversary may probe this bus in order to read private data or to retrieve software code (data confidentiality concern). Another possible attack relies on code injection (data integrity concern). Thus, hardware mechanisms must be designed to ensure data confidentiality and integrity. The conventional way to reach such a goal is to implement a dedicated hardware engine for each security service. Being secured, this approach prevents parallelizability of the underlying computations.

In this thesis, after a study of existing techniques and engines guaranteeing data confidentiality and integrity, two hardware mechanisms dedicated to the security of processor-memory transactions are proposed. First, a Parallelized Encryption and Integrity Checking Engine (PE-ICE) has been designed to provide an effective solution to ensure both security services to data. PE-ICE allows full parallelizations on processor read and write operations while optimizing the hardware resources required. Then, a technique based on a tree structure (PRV-Tree – PE-ICE protected Reference Values) with the same property of full parallelization, is specified to decrease the on-chip memory overhead implied by security mechanisms.

KEYWORDS: Processor Architecture, Embedded Systems, Security, Confidentiality, Integrity, Authentication, Cryptography, System on Chip, Board level attacks, Memory, Performance.

DISCIPLINE : Microélectronique

Université de Montpellier II: Sciences et Techniques du Languedoc
LIRMM : Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier
161 Rue Ada - 34392 Montpellier Cedex 5