SCA1 Model: Towards a concrete security approach to the design of cryptosystems secure against side-channel attacks[1,2]

Filipe Rosado da-Fonseca

Lisbon, Portugal

Email: frdafonseca@mind-security.com

URL: http://www.mind-security.com

August 20, 2004

Abstract

When implementing cryptosystems on general purpose cryptographic hardware, one takes profit of the Application Programming Interfaces (APIs) displaced by the hardware to code the required cryptosystems. The functions made available by these APIs are divided into two groups, the group of the non-cryptographic functions and the group of the cryptographic primitives. When using these functions, one assumes that the functions of the first group are protected against simple side-channel attacks and the functions of the second group are protected against both simple and differential side-channel attacks. Nonetheless, the cryptosystems that make use of these functions may leak information through side-channels. To close this gap of security, a new model is introduced here. It deeply explains how the functions made available by the hardware's APIs must be protected against side-channel attacks and how this hardware must manage memory. In addition, it introduces an adversary that can undertake side-channel attacks against the cryptosystems to test; and it teaches how to represent these attacks in pseudo-code. This paper terminates with both the introduction of some security notions and the presentation of the results of testing some well known cryptosystems in accordance with the latter security notions.

Keywords: side channels, power-analysis, timing-analysis, electromagnetic-analysis, optical-analysis, smart cards, symmetric authentication, pseudorandom permutations, pseudorandom functions, birthday attack.

1 An introduction to the attacks described in this paper was presented at the Rump Session of CHES 2004 (see http://mind-security.com/papers/1).

2 This is the short version of the paper; its long version will be made available at http://mind-security.com/papers/3 .

1 - Table of Contents

2 - Introduction

The creation of side-channel attacks (SCAs) was stimulated due to the increase of both block and key sizes in recent block-ciphers. This type of attack is characterized by the acquisition of information unintentionally leaked by the cryptosystem to attack.

Side-channel attacks may be based on one or more types of leaked information. For instance, they may be based on timing-information (see [39]), power-information (see [41] and [40]), electromagnetic-information (see [21] and [3]), or optical-information (see [17]).

In a side-channel attack, the attacker is characterized by observing the leakages associated with calls to the instance of the cryptosystem to attack[3]. This is so because these leakages depend on the instruction that is processed and, most of times, on its input also.

Accordingly, an attack based on timing-information aims finding out how the time that is taken to process some function varies depending on this function's inputs; an attack based on power-information aims finding out how the power that is consumed to process some function varies depending on this function's inputs; an attack based on electromagnetic-information aims finding out how the electromagnetic information associated with the magnetic fluxes and the electric fields that are created during a function's execution varies depending on this function's inputs; and an attack based on optical-information aims finding out how the information that is displaced by the optical components of the cryptographic hardware to attack varies depending on the inputs of the function to analyze.

Depending on the attacker's aim, side-channel attacks are of one of the following two types:

- simple side-channel attacks (SSCAs) - this type of attack is characterized by the analysis of the leakages that are acquired during a call to the instance of the cryptosystem to attack. The aim of this type of attack is, first, acquiring the information that characterizes the executed processing path[4] and, finally, using the information that characterizes the executed processing path to infer the values that are responsible for its execution;
- differential side-channel attacks (DSCAs) - this type of attack is characterized by the analysis of a number of leakages that correspond to the processing of one specific instruction (which is called during the execution of calls to the instance of the cryptosystem to attack). The aim of this type of attack is, first, acquiring some number of leakages that correspond to the execution of the instruction to analyze and, finally, using these acquired leakages to find a correlation between the values that are known to the attacker and the value to infer.

Both SSCAs and DSCAs have been created with the aim of recovering the secret key used by an instance of either a private key or a public key cryptosystem. To counteract SSCAs and DSCAs several methods have been developed throughout the years in both hardware and software (see, for instance, [40], [13], [24], [28], and [23]). Despite this, it is considered as a rule of thumb that none of these two protection types is sufficient to guarantee a good protection against SCAs.

3 In the art, the information that represents the leakages associated with a call to the instance of the cryptosystem to attack is designated by trace. However, as this designation is associated with the use of an oscilloscope and not all SCAs (for instance, timing-analysis) require the use of such an equipment, here, this information is referred to by leaked information.

4 A function's processing path, for some value of its inputs, is the sequence of instructions that this function executes to carry out the call made to it. Accordingly, a function has only one processing path, if its definition does not define the use of any conditional clause, and the function has more than one processing path, if its definition defines the use of, at least, one conditional clause.

Cryptographic hardware designers not to restrict the possible cryptosystems that may be implemented by use of their solution and to take the most profit of their hardware most usually only implement non-cryptographic functions (functions xor, and, or, not, $\wedge$, $\vee$, $\neq$, =, $\leq$, $\geq$, <, >, :=, mod, +, -, *, /, ||, if, for, while, ceiling, floor, shift left, shift right, rotate left, rotate right, and memory management functions) and cryptographic primitives (block-cipher primitive, hash compression-function primitive, asymmetric encryption trapdoor primitive, and asymmetric authentication trapdoor primitive). These two types of functions are provided later as functions of Application Programming Interfaces (APIs).

Since the introduction of both timing- [39] and power-analysis [41] attacks, more care is being taken with the implementation of the functions displaced by the cryptographic hardware's APIs. However, most of times, the code of the implemented cryptosystems does not have any software protection against SCAs other than the protection that is provided by the latter functions.

For the most profit of the hardware for which the APIs are created, the latter, commonly, are coded in a low-level programming language (for instance, assembly). This is so because creating code in a low-level programming language has the following three main advantages: (1) by taking profit of the knowledge of the internals of the microprocessor, it is possible to both create more efficient implementations and choose the instructions that emanate the least; (2) the instructions made available by the creators of processors usually leak less information than the instructions made available by higher-level APIs; and (3) the compilers associated with low-level programming languages, usually, do not change the programmed code during compilation, what assures one that the protections against SCAs implemented in this code are not broken during the compilation process[5].

Despite these advantages, coding in a low-level programming language has the following two main disadvantages: (1) the code created for one microprocessor (in case this code takes advantage of the particulars of the microprocessor) is not portable to other microprocessors; and (2) programming in this type of language is much more difficult, slower, and prone to error than programming in higher-level programming languages.

As a result, even if the code created in a low-level programming language originates implementations that are both more secure in what regards SCAs and more efficient, its high costs limit its creation to a minimum.

For this reason, it is common to implement both the non-cryptographic functions and the cryptographic primitives in low-level programming languages and to implement the needed cryptosystems in high-level programming languages. Actually, it is common that the non-cryptographic functions and the cryptographic primitives are implemented in a low-level programming language (usually, assembly) by the cryptographic hardware designers and, later, the needed cryptosystems are implemented in an high-level programming language (usually, either C or Java) by the cryptographic solutions providers.

In addition, it is common that only the code that is implemented by the cryptographic hardware designers is customly protected against SCAs. In this description, it is assumed that the cryptographic primitives are protected against both DSCAs and SSCAs and the non-cryptographic functions are protected against SSCAs.

5 The compilers of high-level programming languages, most of times, (during the compilation process) change the implemented code in an attempt to make it more efficient. These changes, however, are very dangerous because they may compromise the security against SCAs of the compiled program. Indeed, here, it is assumed that this happens when the instructions that never affect the output of the "program" (dummy instructions) are included in the source code but the compiler, by not understanding what is its purpose, does not include them on the compiled code. For this reason, one should not count on dummy instructions for the protection against SCAs of the code implemented in high-level programming languages.

As one can see, in this configuration, cryptosystems can be attacked (for instance) by undertaking DSCAs against the non-cryptographic functions defined by them. The success of the DSCAs depending on the way the non-cryptographic functions are defined by the latter cryptosystems.

Take, for instance, the symmetric authentication cryptosystem designated by CBC-MAC [33] (see fig. 1 for a representation of the tagging algorithm, algorithm $T_K(.)$, and of the verification algorithm, algorithm $V_K(.,.)$, associated with this cryptosystem); this cryptosystem is not leakage free in this configuration. This is so because, in this configuration, one is capable of recovering intermediary results of the authentication, which may be used later to forge new authentication tags.

```
T_K(M)
{
    if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return error;
    n_m:=|M|/128;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*128...j*128-1];
    }
    y_0:=<<0>>_128;
    for(i:=1; i≤n_m; i++)
    {
        x1_i:=xor(m_i, y_i-1);
        y_i:=AES_K(x1_i);
    }
    τ:=y_n_m;
    return τ;
}
```

```
V_K(M, τ)
{
    if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return error;
    if (|τ|≠128) then return error; // |tag|=128
    n_m:=|M|/128;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*128...j*128-1];
    }
    y_0:=<<0>>_128;
    for(i:=1; i≤n_m; i++)
    {
        x1_i:=xor(m_i, y_i-1);
        y_i:=AES_K(x1_i);
    }
    τ_:=y_n_m;
    if (τ_=τ) then return 1;
    return 0;
}
```

Fig. 1

With reference to the fig. 1, one can see that to recover intermediary values of the authentication, one has just to undertake a differential side-channel attack to the operation (operation $xor(m_i, y_{i-1})$) that "xores" blocks of the message (values m_i) with intermediary blocks of the authentication (values y_{i-1}).

Thus, even if the functions made available by the cryptographic hardware's APIs are protected as described above, the cryptosystems that make use of these functions may leak information through side-channels. Hence, it was shown here that in this configuration there is a gap on the implementation of cryptosystems against SCAs.

Thus, the present description aims helping the people that create cryptosystems to test them against SCAs, when they are implemented in the above referred configuration. With this purpose, a model in which the adversary is allowed to undertake SCAs against the cryptosystems to test is introduced. This model has the following three main aims:

1. forming the basis of the creation of security notions that take side-channels attacks into account;
2. enabling the test of cryptosystems for different types of the information processed by them (information chosen by the attacker, information both selected uniformly at random and known to the attacker, or information both selected uniformly at random and unknown to the attacker);

3. providing an easy way of showing - through the use of pseudo-code - how the security of cryptosystems can be broken by undertaking side-channel attacks against them.

The model introduced here can be used to test the security of various types of cryptosystems, as, for instance, symmetric encryption cryptosystems, symmetric authentication cryptosystems, asymmetric encryption cryptosystems, asymmetric authentication cryptosystems, and signcryption cryptosystems.

In this version of the paper, however, only the symmetric authentication setting is studied. More precisely, only one of the goals defined for the symmetric authentication setting is studied. Specifically, this goal aims testing the unforgeability of the tags of symmetric authentication schemes.

2.1 – The model

The main aim of the present description is showing that the use by a cryptosystem of functions securely implemented against SCAs is not enough to assure that this cryptosystem is leakage free.

Accordingly, for a cryptosystem to be leakage free, it is necessary that it does not leak any information through side-channels. Nonetheless, depending on the goals to be met by this cryptosystem, some leakages may exist that do not compromise its security.

In this way, the model introduced here aims (1) enabling one to test whether or not a cryptosystem leaks information when it is under SCAs and (2) serving as the basis for the creation of security notions[6] that model each goal to be met by a cryptosystem. This model is hereinafter designated by "SCA Concrete Security Model" or, simply, by "SCA1 Model".

One of the main aims of the SCA1 Model is enabling one to easily show, through the use of pseudo-code, how the security of a cryptosystem can be broken when this cryptosystem is under SCAs. To accomplish this aim, it is thus necessary that one includes in the pseudo-code representation of attacks the enough information to completely describe them.

For this reason, a throughout characterization of side-channel attacks is given in Section 2.1.1 and a complete description of how to represent these attacks in pseudo-code is given in Section 2.1.2. These two Sections may seem a bit long at first reading but, after reading them, one will see how valuable the information on these Sections is for the quest.

The resources that a real-world adversary may have available at any given moment in time are always bounded by some maximum quantity. Thus, to show how feasible an attack is, the resources that it is predicted that a real-world adversary may have available at any given moment in time are studied in Section 2.1.3.

2.1.1 – Side-channel attacks

A Side-Channel Attack (SCA) is an attack characterized by two phases. First, the acquiring phase, the attacker records the information that is leaked by the cryptographic hardware to attack, whenever this

6 A Security Notion is an attempt to describe a goal in a formal way.

hardware executes the instance of the cryptosystem to attack and, finally, the analysis phase, the attacker does an analysis of the recorded leakages.

To facilitate the description of the model, it is assumed here that the acquired leakages provide the enough information for the attacker to distinguish what is the type of the function that is executed at each point of the processing path.

Depending on the type of the analysis that the attacker undertakes in the second phase of a SCA, the attack is defined as a simple side-channel attack (SSCA) or as a differential side-channel attack (DSCA).

In a Simple Side-Channel Attack (SSCA), the attacker, first, acquires the leakages that correspond to the processing of one execution of the instance of the cryptosystem to attack, secondly, analyzes these leakages in order to acquire the information that characterizes the executed processing path and, finally, if the cryptosystem to attack has more than one processing path, analyzes the executed processing path to infer the values that are responsible for its execution.

In other words, in a SSCA, the attacker has the following two aims: first, acquiring the information regarding what are the instructions that are processed, how many times they are processed, and at what place of the processing path they are processed; and, finally, using the latter information to infer the values that are responsible for the executed processing path.

In a Differential Side-Channel Attack (DSCA), the attacker acquires a number of L0 leakages corresponding to the processing of the instruction to analyze, when this instruction has as input both the value to infer (value tValue) and the value that is provided by the attacker at each one of the L0 calls to this instruction (value kValue). This latter value (value kValue) is used with the aim that a correlation is found between it and the value to infer.

In other words, in a DSCA, the attacker acquires a minimum of L0 leakages corresponding to the processing of the function to analyze, with the aim that a correlation is found between the value to infer and the values that are provided by him to the latter function.

To accomplish a DSCA, the attacker has to find out what is the point of the processing path at which the instruction to analyze is processed. Thus, before undertaking the analysis of the acquired leakage, the attacker has to acquire the information that characterizes the executed processing path.

In a DSCA, the analysis undertaken by the attacker to the acquired leakages are based on both, or either, statistical analysis and error correcting codes.

The number L0 of leakages that the attacker has to acquire in order to successfully accomplish a DSCA is dependent on the type of the hardware to attack (for instance, on the type of its bus size, processor clock, cache length, and memory); on the type of the operation to attack (for instance, an exponentiation function, a multiplicative function, or an additive function); on the level of both jitter and circuit noise; and on whether or not the adversary may undertake Template Attacks (see [3]).

In this model, instead of defining a different value L0 for each one of the possible combinations of the above described parameters, the value L0 that corresponds to the most disruptive attack is assumed. Accordingly, it is assumed that the most disruptive attack is the one in which (1) the cryptographic hardware to attack is a smartcard, which is characterized by an 8 bit microprocessor, by not having internal cache, and by the possibility of accessing as much memory as a server[7]; (2) the adversary is capable of basing his attack

7 The possibility that the hardware associated with the most disruptive attack – a smartcard - has of accessing as much memory as a server is only to be considered as being possible in this model and not in the real world. This possibility that the cryptographic hardware to attack has of accessing as much memory as a server is required in order that one can test it when it processes large amounts of information.

on Template Attacks; (3) there is neither jitter nor circuit noise; and (4) the operation to attack is the xor (bitwise-exclusive-or) of two 8-bit values.

Besides depending on the above referred parameters, the number L0 of leakages that the adversary has to analyze to successfully accomplish the DSCA depends on whether the values kValue that he uses for the attack are chosen by him or known to him. Accordingly, it is assumed that these values are chosen by the attacker, when he is able to control each one of their bits; and it is assumed that these values are known to the attacker, when these values are both well distributed in their domain and known to the attacker but at least some of their bits cannot be directly controlled by him.

The distinction between the case in which the attacker is able to choose the values kValue and the other case is very important because, in the first case, as the attacker is able to choose (at each request) the best value to correlate with the value to infer, his analysis needs fewer acquisitions than if values not completely controlled by him were used.

As so, here, the value L0 is defined by L0=L1, in case the attacker is able to choose the values to correlate with the value to infer, and the value L0 is defined by L0=L2, with L2≥L1, in the other case. Moreover, here, the value L2 is a small multiple of the value L1. For instance, L2=a1*L1, with a1 defined by $1 \leq a1 \leq 2^{3}$. Indeed, hereinafter, it is assumed that the value L2 is given by $L2=2^{3}*L1$.

With reference to the results of [43] and taking the most disruptive attack into account, it is advisable to think that the value L1 is a number near the range of 50 to 100 acquisitions. Accordingly, if powers of 2 are considered, then it is advisable that the value L1 is defined in the smallest powers of 2 interval that includes the range of 50 to 100 acquisitions. This interval is given by $2^{\lfloor \log2(50) \rfloor} \leq L1 \leq 2^{\lceil \log2(100) \rceil}$ or, simply, by $2^{5} \leq L1 \leq 2^{7}$.

Taking into account that in this model the value L2 is a small multiple of the value L1 ($L2=2^{3}*L1$) and values represented in the form of powers of 2 are easy to understand, it is advisable to consider that the value L2 is defined in the smallest powers of 2 interval that includes the range of 50 to $2^{3}*100$ acquisitions. Accordingly, this interval is given by $2^{\lfloor \log2(50) \rfloor} \leq L1 \leq 2^{\lceil \log2(800) \rceil}$ or, simply, by $2^{5} \leq L1 \leq 2^{10}$.

The way in which some cryptosystems are defined allows one to test the value to infer at more than one point of the processing path, whether or not the instructions that process the value to infer are of the same type. As a result, for these cryptosystems, the adversary can mount a DSCA characterized by the possibility of analyzing more than one leakage, associated with the same value to infer, for each call to the instance of the cryptosystem to attack. This type of DSCA is hereinafter designated by high-order-DSCA[8].

In a high-order-DSCA, the number of calls to the instance of the cryptosystem to attack is lesser than the number of leakages that the attacker has to acquire to successfully accomplish the DSCA. More precisely, in a high-order-DSCA, the number of requests that the adversary has to undertake to the instance of the cryptosystem to attack (to successfully accomplish the DSCA) is given by $\lceil L2/n\_p \rceil$, where the value n_p (with n_p>1) represents the number of points of the processing path at which a leakage related to the same value to infer is acquired and the value L2 represents the number of leakages that the attacker has to acquire in order to successfully accomplish the attack.

In addition, in this type of attack, it is assumed that the value L0 is given by L0=L2. This is so because in a high-order-DSCA the attacker is not always capable of choosing the best value to correlate with the value to infer.

The protection of functions against DSCAs is most usually based on both splitting and the use of random masks. This type of protection incurs a penalty on the processing needs of a function and, hence, slows down

8 In the art, it is usual to call a high-order-DSCA characterized by the possibility of analyzing the same value to infer at two points of the processing path a second-order-DSCA. Here, however, this distinction is not made.

this function's execution. Therefore, for taking the most profit of the capabilities of the cryptographic hardware, the non-cryptographic functions are not protected usually against DSCAs.

The protection of functions against SSCAs is most usually efficient, as, most of times, the only thing that is needed is that these functions are implemented in such a way that the processing of values of equal length takes the same amount of time. As so, it is hereinafter assumed that both the non-cryptographic functions and the cryptographic primitives are always protected against SSCAs.

With reference to what has been said above, it is hereinafter assumed that:

- only the functions made available by the cryptographic hardware's APIs are customly protected against SCAs;
- the protection against SCAs of the code implemented in high-level programming languages must not be based on the execution of dummy instructions;
- the hardware protections against SCAs of the cryptographic hardware to attack are disabled, when this hardware has this type of protections;
- the functions made available by the cryptographic hardware's APIs are of two different types: the first type is characterized by grouping the non-cryptographic functions (functions xor, and, or, not, $\wedge$, $\vee$, $\neq$, =, $\leq$, $\geq$, <, >, :=, mod, +, -, *, /, ||, if, for, while, ceiling, floor, shift left, shift right, rotate left, rotate right, and memory management functions) and the second type is characterized by grouping the cryptographic primitives (block-cipher primitive, hash compression-function primitive, asymmetric encryption one-way trapdoor primitive, and asymmetric authentication one-way trapdoor primitive);
- the cryptographic primitives are customly protected against both SSCAs[9] and DSCAs[10];
- the non-cryptographic functions are customly protected against SSCAs but not against DSCAS.

2.1.2 – Pseudo-code representation of side-channel attacks

To facilitate the description of the SCA1 Model, it is considered that neither the acquisitions of the leakages nor the analysis undertaken to these acquisitions are accomplished directly by the adversary. This is, it is assumed that both these steps are accomplished by an oracle (oracle SCA-oracle) that is provided to the adversary.

To further facilitate the model, the description of this oracle is defined in a way in which one can easily create pseudo-code representations of side-channel attacks.

Accordingly, there are three types of requests to the oracle SCA-oracle. These requests are defined as follows:

9 Given the knowledge that an adversary has regarding the inputs and the outputs to/from a function, it is said that this function is protected against SSCAs, when the attacker is not capable of mounting a SSCA in which he is able to acquire information he would not be able to acquire otherwise. In this description, it is assumed that the adversary always knows the length of both the inputs and the outputs to/from the function to attack.

10 Given the knowledge that an adversary has regarding the inputs and the outputs to/from a function, it is said that this function is protected against DSCAs, when the attacker is not capable of mounting a DSCA in which he is able to acquire information he would not be able to acquire otherwise. In this description, it is assumed that the adversary always knows the length of both the inputs and the outputs to/from the function to attack.

1. request acquiring_request – this type of request aims requesting the oracle SCA-oracle to record the leakages associated with a call to the instance of the cryptosystem to attack[11,12];
2. request ssca_request – this type of request aims requesting the oracle SCA-oracle to do a simple-side-channel analysis of a recorded acquisition[13];
3. request dsca_request – this type of request aims requesting the oracle SCA-oracle to analyze one of the L0 recorded acquisitions that characterize a DSCA[14].

The first type of request to the oracle SCA-oracle (request acquiring_request) is characterized by having three inputs and no outputs. These inputs are the following:

- value acquiring_request – this value represents a flag that identifies the type of request;
- value recordingIndex - this value represents an index that identifies the place of the acquisition database at which the new recording is stored;
- value oracleToAttack – this value identifies the oracle to attack.

This type of request to the oracle SCA-oracle is characterized by being made before the request to the oracle to attack is made (request requestToOracleToAttack) and by only finishing after the request to the latter oracle is finished. For this reason, the request to the oracle to attack is represented under braces in this request to the oracle SCA-oracle. As so, the third input to this type of request becomes superfluous and, therefore, from hereinafter, it is represented as an empty parameter.

In accordance with what has been said above, the first type of request to the oracle SCA-oracle is represented by:

```
SCA(acquiring_request, recordingIndex, )
{
   requestToOracleToAttack;
}
```

The second type of request to the oracle SCA-oracle (request ssca_request) is characterized by having four inputs and one output. These inputs are the following:

- value ssca_request – this value represents a flag that identifies the type of request;
- value recordingIndex - this value represents an index that identifies the place of the acquisition database at which the required recording is stored;
- value analysisGoal – this value represents a flag that identifies the analysis' goal;
- value expressionToAnalyze - this value represents a string that uniquely identifies the expression to analyze[15,16].

11 To facilitate the present description, it is assumed that the instance of the cryptosystem to attack is embodied by an oracle. This oracle, when referred in a generic way, is called oracle to attack.

12 This request corresponds to the first phase of a SCA.

13 This request corresponds to the second phase of a SCA. Specifically, this request corresponds to the second phase of a SSCA.

14 This request corresponds to the second phase of a SCA. Specifically, this request corresponds to the second phase of a DSCA.

15 In order that one can uniquely identify the expression expressionToAnalyze, it is necessary that one represents the cryptosystem to test in such a way that all the represented lines of "code" (not counting with the comments) are different from each other.

16 Throughout the present description, it is assumed that the value expressionToAnalyze uniquely identifies the expression to analyze. In this way, in a SSCA, if the expression to analyze is characterized by, for instance, the xor of the values a and b, resulting in the value c, then

This type of request to the oracle SCA-oracle is characterized by four different types of analysis' goals. These analysis' goals are the following:

- analysis' goal processedOrNot – this type of analysis' goal aims finding out whether or not the expression expressionToAnalyze is processed at the required point of the executed processing path;
- analysis' goal expressionAtPlace – this type of analysis' goal aims finding out what is the expression that is processed at some point of the executed processing path (for instance, what is the expression that corresponds to the fourth expression that was processed in the sequence of instructions which characterizes the executed processing path);
- analysis' goal allocatedMemory – this type of analysis' goal aims finding out how many "words" of memory are allocated (at the required point of the executed processing path) to store the value that results from the execution of the expression expressionToAnalyze[17];
- analysis' goal copiedMemory – this type of analysis' goal aims finding out how many "words" of memory are copied (at the required point of the executed processing path) to the memory area associated with the value that results from the evaluation of the expression expressionToAnalyze[18];

For this type of request to the oracle SCA-oracle, its answer (value ssca_answer) varies according to the type of the analysis' goal. Accordingly, depending on the type of the analysis' goal, the answer of the oracle SCA-oracle is defined as follows:

- analysis' goal processedOrNot – its answer for this type of analysis' goal is represented by the value that informs whether or not the expression expressionToAnalyze is processed at the required point of the executed processing path[19,20];
- analysis' goal expressionAtPlace – its answer for this type of analysis' goal is represented by the value that identifies the expression (value analyzedExpression[21]) that is processed at the required point of the executed processing path[22];

---

the value expressionToAnalyze is given by expressionToAnalyze="c:=xor(a, b)". Moreover, in case any of the values represented by the expression to analyze is identified by an index, then, in order to uniquely identify this expression, the values of the indexes must be given. In this way, if the expression to analyze is characterized by, for instance, the xor of the values a_i and b_j, resulting in the value c, then the value expressionToAnalyze (considering that the values i and j are equal to, respectively, 8 and 10) is given by expressionToAnalyze="c:=xor(a_i, b_j), with i=8 and j=10". Furthermore, in case the expression to analyze is defined inside a loop and cannot be uniquely identified by the values represented by it, then, to uniquely identify this expression, the value that identifies the iteration of the loop in which the expression to analyze is to be analyzed must be given. In this way, if the expression to analyze is characterized by, for instance, the xor of the values a and b, resulting in the value c, and this expression is to be analyzed at the round 10 of a loop, then the value expressionToAnalyze is given by expressionToAnalyze="c:=xor(a, b), with loop=10".

17 The way in which, in this model, memory is allocated is described in Section 3.1.

18 The way in which, in this model, memory is copied is described in Section 3.1.

19 This answer, depending on whether or not the expression expressionToAnalyze is processed at the required point of the executed processing path, is represented by, respectively, the flags expressionProcessed or expressionNotProcessed.

20 More precisely, the answer of the oracle SCA-oracle for this type of analysis' goal is represented by the value that informs whether or not an expression of the type of the expression expressionToAnalyze is evaluated at the required point of the executed processing path. This is so because if an algorithm is characterized by having (at least) two equal processing paths, then if the expression expressionToAnalyze is processed along one of those, one cannot be assured that the expression that is evaluated corresponds to the expression expressionToAnalyze.

21 The value analyzedExpression is represented in the same way as the value expressionToAnalyze.

22 If the expression that is processed at the required point of the executed processing path can be evaluated by one of (at least) two equal processing paths, then one cannot be assured that the value analyzedExpression corresponds to the expression that was evaluated at the required point of the executed processing path.

- analysis' goal allocatedMemory – its answer for this type of analysis' goal is represented by the value that informs how many memory's "words" are allocated (at the required point of the executed processing path) to store the result of executing the expression expressionToAnalyze[23];
- analysis' goal copiedMemory – its answer for this type of analysis' goal is represented by the value that informs how many memory's "words" are copied (at the required point of the executed processing path) to the memory area associated with the value that results from the evaluation of the expression expressionToAnalyze[24];

The value expressionToAnalyze, for the second type of analysis' goal, instead of representing the expression to analyze, represents the integer that identifies the sequential point of the executed processing path for which the adversary wants to know what is the expression that was evaluated (analysis' goal expressionAtPlace).

With reference to what has been said above, the second type of request to the oracle SCA-oracle is represented by:

ssca_answer:=SCA(ssca_request, recordingIndex, analysisGoal, expressionToAnalyze);

The third type of request to the oracle SCA-oracle (request dsca_request) is characterized by having five inputs and one output. These inputs are the following:

- value dsca_request - this value represents a flag that identifies the type of request;
- value recordingIndex - this value represents an index that informs what is the place of the acquisition database at which the required recording is stored;
- value DSCA_Index - this value represents an index that identifies the DSCA's analysis being undertaken[25];
- value expressionToAnalyze - this value uniquely identifies the expression to analyze[26,27];

---

23 More precisely, the answer of the oracle SCA-oracle for this type of analysis' goal is represented by the value that informs how many memory's "words" are allocated to store the result of processing an expression of the type of the expression expressionToAnalyze, which is processed at the required point of the executed processing path. This is so because if an algorithm is characterized by having (at least) two equal processing paths, then if the expression expressionToAnalyze is processed along one of those, one cannot be assured that the expression that is evaluated corresponds to the expression expressionToAnalyze.

24 More precisely, the answer of the oracle SCA-oracle for this type of analysis' goal is represented by the value that informs how many memory's "words" are copied to store the result of evaluating an expression of the type of the expression expressionToAnalyze, which is evaluated at the required point of the executed processing path. This is so because if an algorithm is characterized by having (at least) two equal processing paths, then if the expression expressionToAnalyze is processed along one of those, one cannot be assured that the expression that is evaluated corresponds to the expression expressionToAnalyze.

25 For each DSCA's analysis being undertaken, the oracle SCA-oracle maintains information associated with it in a database designated by database DSCA_DB, which is indexed by the value DSCA_index.

26 In a DSCA's analysis, the expression to analyze is characterized by only representing one function, the function to analyze.

27 In addition to the information described above for the requests of type ssca_request, in a request of type dsca_request, the information that characterizes the expression to analyze includes also the information regarding what is the value to infer (value tValue) and what is the known value (value kValue). In this way, if the expression to analyze is characterized by, for instance, the xor of the values a and b, resulting in the value c, where the value to infer is the value b and the known value is the value a, then the value expressionToAnalyze is given by expressionToAnalyze="c:=xor(a, b), with tValue=b and kValue=a". Moreover, in case the value that the attacker pretends to acquire is not represented by the total number of bits of the value that is unknown to him, then, in order to uniquely identify the expression to analyze, the part of the values (which are represented by the expression to analyze) that matter for the analysis must be identified. In this way, if the expression to analyze is characterized by, for instance, the xor of the values a and b, resulting in the value c,

- value kValue – this value represents the value that is used to correlate with the value to infer, at each one of the L0 analysis that characterize a DSCA.

In this model, the adversary does not know how to undertake SCAs, being assumed that this knowledge is embodied by the oracle SCA-oracle. In this way, this oracle is the one that selects what is the value that at each call to the function to analyze better correlates with the value to infer. Hence, for each one of the first L0-1 analysis associated with a DSCA, the output of the oracle SCA-oracle (value dsca_answer) is the value that this oracle considers to better correlate with the value to infer[28].

Depending on whether or not the adversary always uses as input to the next call to the function to analyze the value that is returned by the oracle SCA-oracle (in the previous call, associated with the same DSCA, made to this oracle), the value L0 is given by, respectively, L0=L1 or L0=L2.

The output of the oracle SCA-oracle for the requests starting with the request L0 (associated with each DSCA) is the inferred value (value iValue). In case the DSCA is accomplished with success, this value is equal to the target value (value tValue).

However, if the DSCA is not accomplished with success, the target value (value tValue) and the inferred value (value iValue) are different from each other. This happens in case the value kValue that is used in the request made to the oracle SCA-oracle and the value that is used (by the oracle to attack) as input to the function to analyze are different from each other, for at least one of the L0 requests associated with a DSCA.

Accordingly, in this case, to show that the values target value (value tValue) and inferred value (value iValue) are independent from each other, it is assumed here that the inferred value represents a value selected uniformly at random from the domain of the value tValue. This is done to prevent that (in this model) one tries to deduce the correct value from the inferred value without correctly undertaking the DSCA.

In case one of the requests of this type of request to the oracle SCA-oracle asks for the analysis of a function that this oracle knows not to be processed in the executed processing path, then the answer of this oracle is the flag notProcessed.

In addition, if one of the requests of this type of request asks the oracle SCA-oracle for the analysis of a function that this oracle is not sure if it was processed at the required point of the executed processing path, then this oracle analyzes this leakage as if it was sure that the processed function and the function to analyze are the same function. Once more, this happens in case the function that is executed is of the same type of the function to analyze and may be called at the required point of the executed processing path by one of (at least) two equal processing paths.

With reference to what has been said above, the third type of request to the oracle SCA-oracle is represented by:

dsca_answer:=SCA(dsca_request, recordingIndex, DSCA_index, expressionToAnalyze, kValue);

where the value to infer is the value b, the known value is the value a, and the attacker only wants to acquire the first 8 bits of the value b, then the value expressionToAnalyze is given by expressionToAnalyze="c:=xor(a, b), with c:c[0...7], a:a[0...7], b:b[0...7], tValue=b, and kValue=a".

28 Throughout the present description, it is assumed that the way in which the oracle SCA-oracle selects the values to correlate with the value to infer is deterministic. This is, it is considered that for the same function to analyze, for the same value to infer, and for the same already used set of values kValue (associated with both the latter function and the latter value to infer) the oracle SCA-oracle always selects the same value to use in the next call to the function to analyze.

The adversary defined in the SCA1 Model is an algorithm that is given the possibility to make requests to both the oracle SCA-oracle and the oracles that embody the instance of the cryptosystem to test[29]. These latter oracles, depending on whether they embody the default or the inverse algorithms of the cryptosystem to test, are designated by, respectively, default-oracles or inverse-oracles[30].

2.1.3 – Adversary's resources

The aim of the present description is obtaining results that are meaningful for practice. Hence, the approach chosen is a concrete security approach.

This approach is done via the constrain of the resources of the adversary (the latter is, hereinafter, represented by A). Accordingly, the resources of interest are defined as follows:

- value t – this value represents the running time (in seconds) that A has available to accomplish the attack (by convention, both the time to answer all the requests made by A to oracles defined in the experiment and the time to execute A's program are included in this value);
- value pp - this value represents the maximum processing power (in cycles per second) that A has available to accomplish the attack;
- value sc - this value represents the maximum storage capacity (in bits) that A has available to accomplish the attack;
- value q_d – this value represents the total number of queries that A (during the attack) may make to a default-oracle;
- value μ_d – this value represents the total number of bits of ciphertext that A (during the attack) may see in response to his queries to a default-oracle[31];
- value q_i – this value represents the total number of queries that A (during the attack) may make to an inverse-oracle;
- value μ_i – this value represents the total number of bits of plaintext that A (during the attack) may see in response to his queries to an inverse-oracle[32];
- value q_{SCA} – this value represents the total number of queries that A (during the attack) may make to the oracle SCA-oracle.

The use of the above described variables aims providing an accurate way of describing the maximum amount of the latter resources that an adversary has available to accomplish his attack. For this reason, one

29 Even if it is assumed that the adversary defined in the SCA1 Model is an algorithm, the adversary will be hereinafter referred to as if it was a person. Specifically, here, it is assumed that the adversary is a man.

30 For an authentication cryptosystem, the default and the inverse algorithms are, respectively, the tagging and the verification algorithms; for an encryption cryptosystem, the default and the inverse algorithms are, respectively, the encryption and the decryption algorithms.

31 For an encryption cryptosystem, the value μ_d represents the total number of ciphertext bits that the adversary A (during the attack) may see in response to his queries to a default-oracle; for an authentication cryptosystem, the value μ_d represents the total number of tag bits that the adversary A (during the attack) may see in response to his queries to a default-oracle.

32 For an encryption cryptosystem, the value μ_i represents the total number of plaintext bits that the adversary A (during the attack) may see in response to his queries to an inverse-oracle; for an authentication cryptosystem, the value μ_i is always equal to zero.

should predict what are the amounts of these resources that a real-world adversary may have available at some given moment in time. Indeed, this is what is done in this Section.

Commonly, of the above described resources, the ones that are most useful for the attacker are the processing power and the storage capacity he has available at some given moment in time. Hence, below, a way of predicting the amount of these two resources that the adversary may have available at some given moment in time is described.

In [27], the authors calculate the processing power that adversaries may have available at some given moment in time, depending on the amount of money they are willing to invest in this resource. In the present description, the most powerful adversary described in [27] (an adversary with a $300M budget) is considered. However, here, it is assumed that the adversary invests half of his budget on processing power and the other half on storage capacity.

In [42], the author predicts the price of magnetic disk storage till the year 2023. Accordingly, with base on this study, one is able to predict what is the amount of magnetic disk storage that an adversary, which is willing to invest $150M in this resource, may acquire in any of the following years.

Depending on the attack, an adversary may be more interested in acquiring either processing power or storage capacity. Hence, when describing an attack, one should be capable of defining how the $300M budget should be distributed between the latter two resources, in order to maximize the success of the attack. However, as the cost of these resources depends on both the type and the quantity of the hardware that is used and estimating this cost with accuracy is not an easy task, here, one simplifies these calculations by assuming that the $300M adversary invests half of his budget on processing power and the other half on storage capacity.

According to [27], in late 1995, an adversary that had a budget of $300M to invest in processing power could on average find a 40 bits key by brute-force each 0.0002 seconds. As so, this adversary would have been capable of searching $2^{\lceil \log2(2^{39}/0.0002) \rceil}$ keys per second. Assuming that the processing power that is necessary to test each key is 2^6 cycles, then the latter adversary would have had available a processing power of $2^{6+\lceil \log2(2^{39}/0.0002) \rceil}$ cycles per second[33,34]. Considering that the adversary at that moment in time had only a budget of $150M to invest in processing power, he would have had available half the above calculated processing power (this is, in this case, the adversary would have had available a processing power of $2^{6-1+\lceil \log2(2^{39}/0.0002) \rceil}$ cycles per second).

In accordance with what has been said above, if one assumes that Moore's Law[35] is going to be valid for the next years, one is able to predict the processing power capability that an adversary (with a $150M budget to invest in processing power) will have available in any of the years starting with the year 1995.

33 Depending on both the hardware and the type of the instruction to process, the number of instructions that may be processed per cycle varies. However, to facilitate the description, it is hereinafter assumed that one is capable of processing one machine instruction per cycle.

34 Here, it is assumed that the non-cryptographic functions that the attacker needs in order to undertake his attack are available on the hardware described in [27]. Otherwise, it would be necessary to calculate the price of hardware with such capabilities. Moreover, it is assumed that the processing power available to the attacker is given by the total number of cycles per second he has available, independently of the real hardware/software configuration that is present. In other words, it is assumed that for the attacker the hardware that is present appears as a single (very powerful) microprocessor.

In reality, due to the way the hardware described in [27] was defined, the adversary can only take profit of all the cycles per second available to him, in case his attack takes profit of a very high amount of parallel-processing. Nonetheless, to facilitate the description of the SCA1 Model, it is assumed here that the adversary can always take profit of all the cycles per second that are available to him, whether or not his "program" takes profit of a high amount of parallel-processing.

35 Moore's Law is an empiric law that states that the processing power available for a certain cost doubles (more or less) every 18 months.

Accordingly, the processing power capability that it is predicted that the latter adversary may have available in any of the years starting with the year 1995 may be calculated as follows:

```
ProcessingPowerForBudget150M(targetYear)
{
    initialYear:=1995;
    initialCapability:=2^57; // initialCapability=2^6-1+⌈log2(2^39/0.0002)⌉ cycles per second
    if (targetYear<initialYear) then return ⊥;
    targetProcessingPowerCapability:=initialCapability*2^⌈(targetYear-initialYear)*12/18⌉;
    return targetProcessingPowerCapability;
}
```

where: the value initialYear gives the year 0 of the calculations (initialYear=1995); the value initialCapability gives the amount of processing power (in cycles per second) that it is predicted that an adversary would have had available in the year initialYear, if (in that year) he had invested a budget of $150M in processing power; the value targetYear (with targetYear≥initialYear) gives the year for which the processing power is to be calculated; and the value targetProcessingPowerCapability gives the amount of processing power (in cycles per second) that it is predicted that an adversary can have available in the year targetYear, if (in that year) he invests a budget of $150M in processing power.

According to [42], in 1997, a Terabyte of Magnetic Disk Storage (not taking into account the hardware that is necessary to manage it) costed $50328.44. Moreover, it was said there that according to an October 1999 IBM's press release the increase of density in disk storage had doubled for the previous two years (years 1998 and 1999) and it was expected to continue to increase at that rate for the following years.

Accordingly, if one assumes that the increase rate of bit density in disk storage is maintained since the year 1998 and will continue to be maintained in the future, one is able to predict what is the storage capacity that an adversary (with a $150M budget to invest in this resource) will have available in any of the years starting with the year 1997.

Accordingly, the amount of storage capacity that it is predicted that the latter adversary may have available in any of the years starting with the year 1997 may be calculated as follows:

```
StorageCapacityForBudget150M(targetYear)
{
    initialYear:=1997;
    initialCapability:=2^55; // initialCapability=2^40*2^⌈log2(150*10^6/50328.44)⌉*2^3 bits
    if (targetYear<initialYear) then return ⊥;
    targetStorageCapacity:=initialCapability*2^targetYear-initialYear;
    return targetStorageCapacity;
}
```

where: the value initialYear gives the year 0 of the calculations (initialYear=1997); the value initialCapability gives the amount of storage capacity (in bits) that it is predicted that an adversary would have had available in the year initialYear, if (in that year) he had invested a budget of $150M in storage capacity; the value targetYear (with targetYear≥initialYear) gives the year for which the storage capacity of the adversary is to be calculated; and the value targetStorageCapacity gives the amount of storage capacity (in bits) that it is

predicted that an adversary can have available in the year targetYear, if (in that year) he invests a budget of $150M in storage capacity.

When evaluating the security of a cryptosystem, one should take into account two intervals of time. The first interval of time refers to the period of time (in seconds) that a cryptosystem is to be in use (value nUseInterval) and the second interval of time refers to the period of time (in seconds) (value nEvaluationInterval) that a cryptosystem, even if not used to create new information, is in use to process the information created in the first period of time. In other words, in the second period of time only the inverse algorithm of the cryptosystem to test is in use, contrary to what happens in the first period of time, when both the inverse and the default algorithms of the cryptosystem to test are in use.

Depending on the goal of the cryptosystem to evaluate, the duration of the latter two intervals of time differs. Nonetheless, it is assumed here that for general purpose cryptosystems the first interval of time spans 20 years and the second interval of time spans 10 years (t=nUseInterval+nEvaluationInterval, with nUseInterval=20 years and nEvaluationInterval=10 years).

Accordingly, general purpose cryptosystems that are either created or evaluated today should still be secure against an adversary that attacks them in 30 years from now (year 2034). As so, these cryptosystems should be secure against an adversary that at that time has a budget of $150M to invest in processing power and an equal budget to invest in storage capacity. In other words, these cryptosystems should be secure against an adversary that in 30 years from now (year 2034) has available a processing power capacity of $2^{\wedge 77}$ cycles per second and a storage capacity of $2^{\wedge 92}$ bits.

To facilitate testing cryptosystems, it is hereinafter assumed that whether the adversary attacks the cryptosystem to test in the first or in the second intervals of time described above, the access to the cryptographic hardware that executes the instance of the cryptosystem to attack is always equal. As so, the attacker, in the second interval of time, is not forbidden to access the oracles that process the direct algorithm of the cryptosystem to test, contrary to the definition of the second interval of time.

Depending on the configuration in which the cryptographic hardware that executes the instance of the cryptosystem to attack is used, the total number of calls that the attacker may pose to this hardware differs. Nevertheless, to facilitate testing cryptosystems, it is hereinafter assumed (unless something is said on contrary) that the total number of requests that the adversary may make to the oracles which embody the instances of the cryptosystem to test is restricted only by both the time (value t) and the processing power capacity (value pp) that he has available to accomplish the attack.

For a real-world attacker to be able to undertake a SCA, it is assumed that he has both the knowledge and the acquiring equipment that are needed to undertake the attack. The equipment of the attacker may be composed by, for instance, a good oscilloscope and the necessary sensors to connect to it. This oscilloscope (if needed for the attack) must have both pulse interpolation and a real-time sampling rate of between five and ten times the rate of the signal to acquire[36] (see "Display Formats", Section 7 of [10]).

The calculations described above for the amount of processing power and for the amount of storage capacity that an adversary may have available at some given moment in time are based on public information. As so, nothing is claimed regarding the resources that an adversary that uses non-public technology (like, probably, military intelligence adversaries) may have available at some given moment in time. Hence, in what regards these adversaries, the calculations described above should only be used to calculate the lower-bound of the resources that these adversaries may have available at some given moment in time.

36 The rate of the signal to acquire is usually equal to the clock rate of the cryptographic hardware to attack.

2.2 – The main result

In this description, it is shown that one should not infer the security of a cryptosystem against SCAs by the security against these attacks of the functions that are used by it. In other words, it is shown that there is a gap of security, in what regards SCAs, between the security of the functions made available by the cryptographic hardware's APIs and the security of the cryptosystems that are implemented by use of the latter functions.

In addition, it is shown that an attacker, most of times, can take profit of this gap to break the security of the cryptosystems that do not take it into account (see Appendix C).

The above referred results are very important for practical implementations because they show that SCAs are not used only to acquire the secret key of the cryptosystem to attack, but indeed they may be used to acquire any intermediary variable that this cryptosystem leaks (due to the way the functions defined by it are joined together). As so, it was demonstrated here that to be assured that a cryptosystem is secure against SCAs, it is necessary to evaluate it as a whole and not as the sum of the parts.

One example of a cryptosystem that is secure in a model in which SCAs are not defined[37] but not in the SCA1 Model is the symmetric authentication scheme designated by PMAC [11].

Accordingly, in [11], PMAC was proved secure in the SCA0 Model because in this setup the adversary is not capable of acquiring intermediary results of the authentication. In the SCA1 Model, however, PMAC is completely insecure because the adversary is capable of acquiring intermediary results of the authentication, which he can use later to forge new authentication tags (see Section C.6).

Some years ago, cryptographic hardware was used only on restricted entry facilities. Nowadays, however, with the advent of both the internet and "on the move computing" more and more hardware with cryptographic capabilities (for instance, PDAs, notebooks, smartcards, and mobile phones) is used in hostile environments. In these environments, in which the adversary may have control, it is much more difficult to securely protect cryptosystems. This is even more difficult if the attacker undertakes non-invasive attacks (as, for instance, side-channel attacks).

Accordingly, if the type of SCAs undertaken by the attacker is (for instance) based on both, or either, electromagnetic-analysis (see [21] and [3]) and optical-analysis (see [17]), then these attacks may even be undertaken when the attacker and the hardware to attack are at a distance of some meters.

Side-channel attacks based on power-analysis, on electromagnetic-analysis, and on optical-analysis (in accordance with, respectively, [40], [21], and [17]) are easy and cheap to implement. Therefore, these attacks must be considered as posing a major threat to the security of cryptosystems.

In addition, as the latter three types of side-channel attacks require that one is capable of undertaking timing-analysis, one is led to the conclusion that SCAs based on this type of analysis are also cheap and easy to implement.

37 The model in which side-channel attacks are not defined is hereinafter designated by "No-SCA Concrete Security Model" or, simply, by SCA0 Model. To facilitate the comparison between the SCA0 Model and the SCA1 Model, it is hereinafter assumed that the SCA0 Model is equal to the SCA1 Model, except that the adversary defined in the SCA0 Model is strictly forbidden to make requests to the oracle SCA-oracle. In this way, it is automatically assumed that all the security notions that are defined in the SCA1 Model are also defined in the SCA0 Model, whether or not these security notions are moot in the latter model.

In light of both the facility and the low cost of performing side-channel attacks, the results presented here are all the more meaningful.

2.3 – Related work

The model introduced in this paper aims incorporating the notion of side-channel attacks into the security notions that are used to test the security of symmetric encryption schemes, symmetric authentication schemes, asymmetric encryption schemes, asymmetric authentication schemes, and *signcryption* schemes. As far as the author is aware, this is the first paper of this kind.

There are, at least, four types of side-channel attacks. The first type is characterized by the use of timing-information (see [39], [12], [9], and [4]); the second type is characterized by the use of power-information (see [41], [40], [29], and [2]); the third type is characterized by the use of electromagnetic-information (see [21] and [3]); and the fourth type is characterized by the use of optical-information (see [17]).

There are several papers that study the security (in a concrete security approach) of symmetric encryption schemes, of symmetric authentication schemes, of asymmetric encryption schemes, of asymmetric authentication schemes, and of *signcryption* schemes. Accordingly, the security of symmetric encryption schemes, in what regards its confidentiality, is studied in [31] and [18] and, in what regards its unforgeability, is studied in [19] and [45]; the security of symmetric authentication schemes, in what regards its unforgeability, is studied in [33] and [32]; the security of asymmetric encryption schemes, in what regards its confidentiality, is studied in [30] and, in what regards its key-privacy, is studied in [25]; and the security of *signcryption* schemes, in what regards both its unforgeability and its confidentiality, is studied in [48], [15], [20], and [47] (see also [14]).

2.4 – Discussion – understanding the main result

The main result of this paper is that there is a gap between the security against SCAs of the functions made available by the cryptographic hardware's APIs and the security against these attacks of the cryptosystems that are implemented by use of the latter functions.

To fill this gap, one must find out how a cryptosystem must be implemented (on hardware) to be secure against SCAs. As so, in this Section, three forms of implementing a cryptosystem in hardware are described and the preferred form is chosen, taking into account its advantages and disadvantages.

There are two main forms of securely coding cryptosystems against SCAs. The first form (form I) is characterized by customly coding the cryptosystems to implement in a secure way against side-channel attacks; the second form (form II) is characterized by, first, customly coding against side-channel attacks both the non-cryptographic functions and the cryptographic primitives to be used by the cryptosystems to implement, and, finally, verifying that the way in which these cryptosystems use the latter two types of functions does not give origin to leakages when these cryptosystems are under SCAs.

A different type of cryptographic hardware is associated with each one of the above described forms of securely coding cryptosystems against SCAs. These two types of cryptographic hardware are defined as follows:

- customized cryptographic hardware – (associated with the form I of securely coding cryptosystems against SCAs) is created in a customized way in order to fulfill the needs of the cryptosystems to implement on it;
- general purpose cryptographic hardware - (associated with the form II of securely coding cryptosystems against SCAs) provides the resources, in the form of APIs, to be used in the coding of the cryptosystems to be implemented on it.

The form II of securely implementing cryptosystems against SCAs may be subdivided in two other forms, depending on the extent to which the non-cryptographic functions are customly coded against SCAs. These two sub-forms of the form II differ because, in the first of these two forms (form II.1), the non-cryptographic functions are customly coded to be secure against both SSCAs and DSCAs and, in the second of these two forms (form II.2), the non-cryptographic functions are only customly coded to be secure against SSCAs.

The form I of securely implementing cryptosystems against SCAs has the following main advantage:

- only the parts of the cryptosystems to implement that need to be protected against SCAs are protected against this type of attacks. Therefore, there is no waste of processing power on the processing of software protections not necessary for the secure implementation of the required cryptosystems.

This form of securely implementing cryptosystems against SCAs has the following disadvantages:

- the customers of the cryptographic hardware associated with the form I are restricted to the cryptosystems that the cryptographic hardware designers implement on it;
- as the software implementation of the cryptosystems implemented in this form is customized, it is not possible to directly compare the performance of these cryptosystems from their definition;
- creating customized cryptographic hardware (most usually) is more expensive than buying and later customizing general purpose cryptographic hardware.

The form II.1 of securely implementing cryptosystems against SCAs has the following advantages:

- the customers of the cryptographic hardware associated with the form II are able to implement on it any cryptosystem that defines the use of functions made available by the cryptographic hardware's APIs;
- as there is no hidden software customization for any of the cryptosystems implemented in this form, the performance of these cryptosystems can be directly compared from their definition;
- buying and later customizing general purpose cryptographic hardware (most usually) is cheaper than creating customized hardware.

This form of securely implementing cryptosystems against SCAs has the following main disadvantages:

- there is a waste of processing power each time a non-cryptographic function that does not need to be protected against DSCAs (in order that the cryptosystem that defines its use is secure against DSCAs) is called;

- the responsibility of the creation of cryptosystems secure against SCAs is left to the people that create cryptosystems and not to the cryptographic hardware designers.

The form II.2 of securely implementing cryptosystems against SCAs has the following advantages:

- the customers of the cryptographic hardware associated with the form II are able to implement on it any cryptosystem that defines the use of functions made available by the cryptographic hardware's APIs;
- as there is no hidden software customization for any of the cryptosystems implemented in this form, the performance of these cryptosystems can be directly compared from their definition;
- the implementation of cryptosystems in this form gives origin to code almost as efficient as the code associated with the customized implementation of these cryptosystems;
- buying and later customizing general purpose cryptographic hardware (most usually) is cheaper than creating customized hardware.

This form of securely implementing cryptosystems against SCAs has the following disadvantage:

- the responsibility of the creation of cryptosystems secure against SCAs is left to the people that create cryptosystems and not to the cryptographic hardware designers.

Of the above described forms of securely implementing cryptosystems against SCAs, the preferred form, in case one has no monetary restrictions, is the form I. This is so because, as the implementation of cryptosystems in this form is customized, the code of the cryptosystems implemented in this form is presumably both very efficient and secure against SCAs.

Nevertheless, as one usually has monetary restrictions, in this case, the preferred form of securely implementing cryptosystems against SCAs is the form II.2. This is so because this form gives origin to cryptosystems' implementations that are both cheap and very efficient.

In addition, the form II.2 is the form that better resembles the way in which cryptosystems are usually implemented in hardware. Therefore, for the SCA1 Model to resemble real-world implementations only slightly changes are necessary to the way in which, commonly, general purpose cryptographic hardware is created (see Section 3).

The form II.2 has the main disadvantage of leaving the responsability of the creation of cryptosystems secure against SCAs to the people that create cryptosystems and not to the cryptographic hardware designers. Hence, the SCA1 Model should help the people that create cryptosystems to create cryptosystems secure against SCAs, when these cryptosystems are implemented in accordance with the form II.2.

For this reason, the creation of the SCA1 Model is highly relevant.

2.3 – Outline of the paper

In Section 3, the definition of some basic notions is given. In Section 4, the formal definition of the security notion designated by “Existential Unforgeability” is given and, additionally, the results of studying the security of the symmetric authentication schemes CBC-MAC (see [33]), DMAC (see [8]), EAX-MAC (see [26]), HMAC

(see [36]), OMAC (see [44]), PMAC (see [11]), RIPE-MAC (see [1]), RMAC (see [8]), TMAC (see [22]), XECB$-MAC (see [46]), and XMACR (see [34]) are presented, in accordance with the latter security notion. Sections 5, 6, and 7 are, respectively, the Conclusions, the Acknowledgments, and the Bibliography Sections. In Appendix A, the definition of both pseudorandom functions and pseudorandom permutations is given. In Appendix B, the proofs of both the Theorems and the Propositions introduced in this description are given. In Appendix C, the pseudo-code representation of the attacks undertaken against the above referred symmetric authentication schemes is presented. And, in addition, in Appendix D, a brief resume of the contents of the long version of the paper is presented.

3 - Preliminaries

This Section is subdivided into three Sections. The first Section (Section 3.1) describes general preliminaries; the second Section (Section 3.2) describes preliminaries that refer to the SCA1 Model in general; and the third Section (Section 3.3) describes preliminaries that refer specifically to the (SCA1 Model's) symmetric authentication setting.

This Section may seem a bit long at first reading but it is very important for the quest. This is so because here: information to include in the pseudo-code representation of attacks is presented; information regarding how it is assumed that hardware manages memory is presented; and some warnings regarding the compilers of high-level programming languages are given.

3.1 – General

A *family of functions* is a map $F: Keys(F)\times Dom(F)\rightarrow Ran(F)$, where: Keys(F) is the key space of F, Dom(F) is the domain of F, and Ran(F) is the range (or codomain) of F. The two-input function F takes as input both a key $K\in Keys(F)$ and a value $X\in Dom(F)$ and returns a point $F(K, X)\in Ran(F)$. If the key space of F, Keys(F), is defined by $Keys(F)=\{0,1\}^k$, where k is an integer, then k is referred as the key's length. If the domain of F, Dom(F), is defined by $Dom(F)=\{0,1\}^l$, where l is an integer, then l is referred as the input's length. If the range of F, Ran(F), is defined by $Ran(F)=\{0,1\}^L$, where L is an integer, then L is referred as the output's length. In the present paper, the sizes of k, l, and L are always considered finite.

In addition, for each key $a\in Keys(F)$, the map F_a: $Dom(F)\rightarrow Ran(F)$ is defined by $F_a(X)=F(a,X)$, for all $X\in Dom(F)$. In this way, F specifies a collection of maps from Dom(F) to Ran(F), being each map associated with a key. Hence, F_a is interchangeably referred to as an instance of F or as an element (or member) of the family F. It is often said that a key is selected uniformly at random from Keys(F), being this operation represented by $a\leftarrow R\text{—}Keys(F)$. As a result, $f\leftarrow R\text{—}F$ is written to denote the operation $\{a\leftarrow R\text{—}Keys(F);\ f\leftarrow Fa\}$. In other words, $f\leftarrow R\text{—}F$ is written to denote the operation of selecting uniformly at random a member of the family F.

Let $Rand^{vl\rightarrow vL}$ represent the family of all the functions with input's length, value vl, defined by $0<min\_vl\leq vl\leq max\_vl$ and with output's length, value vL, defined by $0<min\_vL\leq vL\leq max\_vL$. Moreover, let

$Rand^{vl \to vL}$ represent the family of all the functions that for the same value of input have the same length of output.

The cardinality of $Rand^{vl \to vL}$, even for moderately small values of both max_vl and max_vL, is too big to be practical to define an index that may be used to identify any of the members of this family of functions. Hence, from hereinafter, a subset of 2^k random members of $Rand^{vl \to vL}$ is considered (subset $Rand_k^{vl \to vL}$).

As the members of $Rand_k^{vl \to vL}$ are selected uniformly at random from a much bigger set of functions, it is assumed that a relation between the inputs and the outputs of any of these functions cannot be discovered by any probabilistic algorithm that runs in time polynomial in k. Therefore, to facilitate the description, it is assumed that for each input defined in the domain of each one of those functions the corresponding output is a random value (defined in the codomain of those functions) that is independently distributed of all the other outputs associated with the respective function.

Let $Rand^{l \to vL}$ and $Rand_k^{l \to vL}$ represent, respectively, the sub-families of $Rand^{vl \to vL}$ and $Rand_k^{vl \to vL}$ that are characterized by only differing from the parent-families because the length of their inputs is fixed. Let $Rand^{vl \to L}$ and $Rand_k^{vl \to L}$ represent, respectively, the sub-families of $Rand^{vl \to vL}$ and $Rand_k^{vl \to vL}$ that are characterized by only differing from the parent-families because the length of their outputs is fixed. Let $Rand^{l \to L}$ and $Rand_k^{l \to L}$ represent, respectively, the sub-families of $Rand^{vl \to vL}$ and $Rand_k^{vl \to vL}$ that are characterized by only differing from the parent-families because the length of both their inputs and their outputs is fixed.

Let $Perm^l$ represent the sub-family of $Rand^{vl \to vL}$ that is characterized by representing the family of all the permutations with block's length equal to l.

The cardinality of $Perm^l$, even for a moderately small l, is too big to be practical to define an index that may be used to identify any of the members of this family of permutations. Hence, from hereinafter, a subset of 2^k random members of $Perm^l$ is considered (subset $Perm_k^l$).

As the members of $Perm_k^l$ are selected uniformly at random from a much bigger set of permutations, it is assumed that a relation between the inputs and the outputs of any of those permutations cannot be discovered by any probabilistic algorithm that runs in time polynomial in k. Therefore, to facilitate the description, it is assumed that for each input defined in the domain of each one of those permutations the corresponding output is a random value (defined in the codomain of those permutations) that is different from any of the other outputs associated with the respective permutation.

An array (also designated by sequential list, see [5]) represents a space of memory that is allocated in the form of n sequential slots of length valuesLength bits. The access to the value inside each one of the slots of an array is done by accessing the area of memory associated with the respective slot. Hence, the address of the bit 0 of the value to access (value addressOfTargetValue) is given by addressOfTargetValue=baseAddress+(i-1)*valuesLength, where the value baseAddress gives the address of the first bit in the array, the value i (with i=1...n) identifies the slot to access, and the value valuesLength gives the length of each one of the n slots of the array.

To facilitate the description, it is considered here that the access to the values inside an array designated by, for instance, testArray, which is defined as containing n slots of length valuesLength bits, is represented by targetValue=testArray[i-1], where the value i (with i=1...n) identifies the slot that contains the value to access and the value targetValue (which has a length of valuesLength bits) represents it.

Computers, usually, do not allocate (nor process) memory bit by bit but in multiples of some number of bits. This smallest amount of memory that is allocated (or processed) by a computer is hereinafter designated by "word" of memory. The number of bits that compose each word of memory is dependent on the hardware

that is present. For instance, in the usual 8-bit smartcard, the word of memory is 8 bits in length, in the usual PC, the word of memory is 32 bits in length, in the recent servers, the word of memory is 64 bits in length, and, probably, in the recent "supercomputers", the word of memory is 128 bits in length.

Accordingly, the memory that is necessary to allocate to store an array testArray of n slots of length valuesLength bits is represented by n*valuesLength, in case (n*valuesLength mod "length of the word of memory") is equal to zero, or by n*valuesLength+("length of the word of memory"-((n*valuesLength) mod "length of the word of memory"))) bits, otherwise[38].

With reference to what has been said above, one can see that there is the need to maintain a value that informs how many bits associated with an array are meaningful for the "program" that created it. For this reason, it is necessary to consider that for each region of allocated memory there is a value, LINK value, that informs where this region of memory starts, how many bits of it are meaningful for the "program", and in how many equal parts (slots) the meaningful area of the allocated memory is divided.

Accordingly, each LINK value is composed of three values, values slotLength, numberSlots, and memLocation, which length is sufficient to store, respectively, the value that represents the maximum number of bits of memory that an array with only one slot can have, the maximum number of slots of one bit each that an array can have, and the value that identifies any of the words of memory that may be allocated[39].

In this model, all the variables (except for the LINK values) are defined as arrays.

In order that a region of memory is easily identified in the code of a "program", it is usual to associate it with a name chosen by the "programmer". This name is hereinafter designated by variable name and the memory associated with it is designated by variable's value. For the "run-time environment"[40], this association between a variable name, a LINK value, and a region of allocated memory is called alias.

An alias can be of two different types, these are the owner alias and the read-only alias. The owner alias informs the "run-time environment" that the region of memory associated with it can be both read and modified, when this region of memory is identified through the use of this alias; the read-only alias informs the "run-time environment" that the region of memory associated with it can be read but not modified, when this region of memory is identified through the use of this alias. The type of alias is configured automatically by the "run-time environment" and can only be modified by it.

Here, it is assumed that the memory allocated inside the code that represents a function's definition is local to this function. This is, it is assumed that the memory allocated inside a function's definition can only be accessed by this function. Moreover, it is considered that a function can only access memory that is not local to it, in case an alias to this memory is passed as input to it. In this case, however, the function can read but not modify the region of memory associated with the alias passed as input to it.

Accordingly, when a new region of memory is created, an owner alias is created and, when read-only access to a region of memory is given to a function other than (the instance of) the function that allocated this memory, a read-only alias is created.

In this description, the "programmer" cannot access the LINK values directly. Indeed, he can only access directly the meaningful bits of memory associated with the LINK values. Nonetheless, the "programmer" can

38 Here, it is considered that the creation of an array of 0 bits allocates one word of memory.

39 In the real-world, the length of the values slotLength, numberSlots, and memLocation would be both fixed and dependent on the type of the hardware that is present. However, here, to facilitate the description, the length of these values is assumed as being equal, irrespectively of the type of the hardware that is present. Specifically, to allow that the length of these values remains actual for the next years, it is assumed that these values have a length of 128 bits.

40 The "run-time environment" is the "environment" in which a program runs.

get the information regarding both how many meaningful bits and how many slots are associated with a variable. Accordingly, for a variable designated by, for instance, testArray, the number of meaningful bits associated with it is given by |testArray|[41] and the number of slots associated with it is given by getNumberSlots(testArray).

With reference to what has been said above, the creation of an array designated by testArray, which is defined as having n slots of length valuesLength bits, is represented by testArray:=Array(n, valuesLength). Here, it is assumed that neither the number of slots of an array nor the length of these slots can be modified after the array is created.

Sometimes, one needs to access a value (array subArray) that is characterized by representing m sequential slots of a main array (array testArray) that has n ($m \leq n$) slots of length valuesLength bits. In this case, this sub-array may be represented by subArray=testArray[a1-1...a2-1], where the values a1 and a2 are defined by $1 \leq a1 \leq a2 \leq n$ and the value subArray represents an array of m=(a2-a1) slots of length valuesLength bits[42].

When a new region of memory is allocated, the content that is stored on it is deleted. More precisely, in the process in which a new region of memory is allocated, the "run-time environment" sets each word of this region of memory to the value zero[43], before allowing the access to it.

In this model, all the memory that is allocated inside a function is local to this function, the only exception being the memory associated with the values returned by this function. Indeed, after these values are returned, the memory associated with them starts to be local to the function that "holds" these values. This is, this memory starts to be local to the function that calls the function that returns these values.

Accordingly, when a value is set as being equal to the result of executing some function, this value is not formally created as an array but it is simply set by the "run-time environment" as an owner alias to the region of memory associated with the value that is returned by the called function.

When an array a1 is set as being equal to an array a2, two things may happen. This is, if the array a2, after this operation, is not referred anymore in the code, then, first, the array a1 is set as an owner alias to the region of memory associated with the array a2 and, finally, the alias a2 is deleted. Otherwise, first, the array a1 is created with the same number of slots and the same number of meaningful bits of the array a2 and, finally, the words of memory associated with the array a2 are copied to the array a1[44,45].

When a value (array a1) is set as being equal to the result of evaluating some expression other than the one that represents either the result of executing a function or the copy of arrays, then, first, the expression is read to find out if all the values that are referred by it have the same slot's length, secondly, the expression is read to find out how many meaningful bits result from its evaluation, thirdly, the array a1 is created in

41 The number of meaningful bits of the variable testArray can also be given by getLength(testArray).

42 It is assumed here that if the value a2 is smaller than the value a1, then the value subArray=testArray[a1-1...a2-1] represents an array of zero meaningful bits. This array is hereinafter designated by empty_string.

43 In this case, the adversary can discover the length of the region of memory that is allocated by undertaking a SSCA to the process in which this region of memory is zeroed. Indeed, this type of analysis' goal is the one designated by allocatedMemory.

44 If the array a1 is already created with the same number of slots and the same number of meaningful bits of the array a2, then the operation that creates a new array is skipped and only the copy operation is undertaken.

45 In this case, the adversary can find the number of memory's words that are copied between arrays by undertaking a SSCA to the process in which the copy operation is undertaken. Indeed, this type of analysis' goal is the one designated by copiedMemory.

accordance with the information acquired in the first two steps[46] and, finally, the expression is evaluated and the result of its evaluation is copied to the new array[47].

In accordance with the definition of arrays, a string is an array of bits. Strings, however, depending on the context, may be understood as arrays of bits, as numbers, or as base 2 polynomials. In addition, the values inside the slots of arrays are always considered as strings of bits.

In this model, memory is represented in the big endian form.

Above, it is said that the memory that is allocated inside a function can only be accessed by this function. Moreover, it is said that this function can only access memory not local to it, in case an alias to this memory is provided as input to it. However, when this is said, it is assumed that the called functions are not represented inline inside the functions that call them.

Accordingly, the functions (inline functions) which description is inline to the functions that call them do not allocate memory local to them but local to the functions that call them. Moreover, the inline functions can access all the memory local to the functions that call them. As so, the inline functions are not characterized by an header that specifies both its inputs and its outputs but only by a header that specifies how these functions are used. Examples of inline functions are the for function, the while function, and the if function.

The for function represents a loop characterized by three parameters. The first parameter defines a variable that the for function uses as a counter; the second parameter defines a boolean expression (which is dependent on the first parameter) that defines whether or not the loop starts and, if it starts, when it stops; and the third parameter defines how the first parameter is incremented each time the loop completes a round.

The while function represents a loop characterized by one parameter. This parameter represents a boolean expression used to verify whether or not the loop starts and, if it starts, when it stops.

The functions for and while both have as parameter a boolean expression which aim is defining if the loop starts and, if it starts, when it stops. Even so, when one wants that the loop terminates immediately, irrespectively of the evaluation of the latter boolean expression, one uses the command “break”. This command resumes the operation of the function that called the loop at the instruction right after the loop's pseudo-code representation.

The if function represents a conditional clause that has as parameter a boolean expression. This function is characterized by evaluating the boolean expression and by, depending on the value that results from this evaluation, resuming the program's execution at one of two different points of the processing path. This is, if the boolean expression evaluates to true, then the program's execution jumps to the first point of the processing path associated with the if function; if the boolean expression evaluates to false and there is a string “else” associated with the if function, then the program's execution jumps to the second point of the processing path associated with the if function. Otherwise, the if function terminates.

The first point of the processing path associated with the if function corresponds to the point where the instructions associated with a positive evaluation of the boolean expression are represented; the second point of the processing path corresponds to the point of the processing path where the instructions associated with a negative evaluation of the boolean expression are represented.

The representation of a function F that is characterized by having n inputs (values a_i, with i=1...n) is F (a_1, ..., a_n). When the number of inputs of the function F is known, then, when referring to this function in a

46 If the values referred by the expression to evaluate have all the same slot's length, then the new array is created with this slot's length. Otherwise, the new array is created with a slot's length equal to one bit.

47 If the array a1 is already created with both the number of meaningful bits and the number of slots that are necessary to store the result of evaluating the expression, then the operation that creates a new array is skipped.

generic way, these inputs do not need to be referred by name but can be represented, simply, by a point. In addition, if the function F is characterized by the possibility of being instanced with a variable number of inputs, then, when this function is referred in a generic way, this function may be simply represented by F (...), being assumed that its number of inputs is understood by the context in which the function is referred.

The error value is represented by the symbol $\perp$[48,49]; the function msb(.) returns the most significant bit of its input; the function lsb(.) returns the least significant bit of its input; the function ntz(.) returns the number of trailling bits zero that characterizes its input; the function shift-right(.) shifts one bit to the right the bits of its input; the function shift-left(.) shifts one bit to the left the bits of its input; the function rotate-right(.) rotates one bit to the right the bits of its input; the function rotate-left(.) rotates one bit to the left the bits of its input; the function ceiling(.) rounds its input to the smallest integer greater than, or equal to, its input[50]; the function floor(.) rounds its input to the greatest integer lesser than, or equal to, its input[51]; the function log2(.) gives the logarithm in the base 2 of its input; and the operation a1||a2 represents the concatenation of an array a2 to an array a1[52].

The function a1:=a2 sets the value a1 to the result of evaluating the expression a2. When the evaluation of the expression a2 gives origin to more than one result, for instance, values $b_1,...,b_n$, with n>1, then this expression is represented by $(b_1,...,b_n):=a2$. In addition, when the expression a2 is not completely described, instead of the use of the string ":=", one uses the symbol "$\leftarrow$". In this description, this happens when one represents either the values returned by the adversary at the end of an experiment or the creation of cryptographic keys.

The operations or(.,.), xor(.,.), and(.,.), and not(.) represent, respectively, the bitwise-or of its inputs[53], the bitwise-exclusive-or of its inputs[54], the bitwise-and of its inputs[55], and the bitwise-not of its input[56]. It is said that these functions process its inputs bit by bit, however, as computers only process information in multiples of the word's length, when necessary, it is considered that these functions make use of masks with the word's length to accomplish the task at hand.

The operations $\wedge$, $\vee$, $\leq$, $\geq$, <, >, $\neq$, and = represent, respectively, the boolean operations and, or, less than or equal to, greater than or equal to, less than, greater than, not equal to, and equal to.

The operation a1+b1 gives the result of adding the values b1 to the value a1. Sometimes, this operation is also represented by sum(a1, b1). The operation a1-b1 gives the result of subtracting the value b1 from the

48 Here, the error value (value $\perp$) is associated with a LINK value that does not point to any region of allocated memory. Moreover, the error value is not created by a program but it is provided by the "run-time environment". Furthermore, the error value and the value empty_string are defined in different ways, so that they can be distinguished from each other. Specifically, both the values $\perp$ and empty_string are associated with 0 meaningful bits of memory but the latter value is associated with (one word of) allocated memory and the former is not.

49 When an array is set as being equal to the value that is returned by a function and this function returns the error value, then the "run-time environment", instead of setting the new array as an owner alias to the returned information, sets it as a read-only alias to the region of memory associated with the error value. If the creation of owner alias and read-only alias has the same side-channel fingerprint, then one is assured that undertaking SCAs against the instruction which returns information is not useful to distinguish whether or not a function returns the error symbol. This is assumed hereinafter.

50 This operation can also be represented by $\lceil a1 \rceil$ or by roundUp(a1).

51 This operation can also be represented by $\lfloor a1 \rfloor$ or by roundDown(a1).

52 If the arrays that are concatenated have the same slot's length, then the new array is created with a slot's length equal to the slot's length of the latter arrays. Otherwise, the resulting array is created with a slot's length equal to one bit.

53 The bitwise-or of two bits is equal to 0, if both the bits are equal to 0, and is equal to 1, otherwise.

54 The bitwise-xor of two bits is equal to 0, if the two bits are equal to each other, and is equal to 1, otherwise.

55 The bitwise-and of two bits is equal to 0, if any of the bits is equal to 0, and is equal to 1, otherwise.

56 The bitwise-not of the bit 0 is equal to 1 and of the bitwise-not of the bit 1 is equal to 0.

value a1. Sometimes, this operation is also represented by subtraction(a1, b1). The operation a1*b1 gives the result of multiplying the value a1 by the value b1. Sometimes, this operation is represented by either mult(a1, b1) or multiplication(a1, b1). The operation a1/b1 gives the result of dividing the value a1 by the value b1. Sometimes, this operation is represented by either division(a1, b1) or div(a1, b1). The operation a1 mod b1 gives the remainder of the division of the value a1 by the value b1. Sometimes, this operation is also represented by mod(a1, b1).

Sometimes, a string is characterized by representing the result of concatenating another string to itself a determined number of times. In this case, the resultant string is given by $a1^{a2}$, where the value a1 represents the string that is concatenated to itself and the value a2 (with $a2 \geq 0$) represents the integer that informs how many times the value a1 is concatenated to itself ($|a1^{a2}|=a2*|a1|$ bits). For instance, the string $(1010)^{32}$ is characterized by representing a string of length 128 bits, which results from concatenating 32 times the string 1010 to itself; and the string 1010^{125} (also represented by $101||0^{125}$) is characterized by representing a string of length 128 bits, which results from concatenating 125 times the one-bit string 0 to the string 101.

In order that the above described way of representing strings is not confused with the way in which numbers are raised to some power, here, the representation of powers is always preceded by the symbol "^". For instance, the value $1010^{\wedge 32}$ represents the number that results from raising the integer 1010 to the power 32.

The operation that represents the increment in one unit of a value a1 is represented by a1:=a1+1 mod |a1|. Sometimes, when the length of the value a1 is understood by the context, this operation is simply represented by either a1++ or a1:=a1+1. The operation that represents the decrement in one unit of a value a1 is represented by a1:=a1-1 mod |a1|. Sometimes, when the length of the value a1 is understood by the context, this operation is simply represented by either a1-- or a1:=a1-1.

The operation that represents the selection uniformly at random of a value r from the domain of the well distributed strings of length a1 bits is represented by $r \leftarrow R \text{—} \{0,1\}^{a1}$, with a1 defined as an integer greater than zero (|a1|>0 bits). In the pseudo-code representations, however, this operation is represented by r:=getRandomBits(a1), where a1 gives the number of random bits to be returned by this function and r represents the result of its execution.

The codification of a value w1 into a string whose length, given by the value b1, is bigger than, or equal to, the length of the value w1 ($b1 \geq |w1|$) is denoted by $<<w1>>_{b1}$. This codification is characterized by the prepending of b1-|w1| bits 0 to the value w1, being represented by $<<w1>>_{b1}:=0^{b1-|w1|}||w1$, where || denotes the concatenation operation.

The symbols "{" and "}" are used to delimitate sections of pseudo-code. Each line of code terminates with the symbol ";", except for (1) the line where only the symbol "{" is represented, (2) the line where only the symbol "}" is represented, or (3) the line before the line where only the symbol "{" is represented. In this description, it is assumed that the symbols "{" and "}" are always represented in its own line. Moreover, it is assumed that the lines of code inside the functions for and while are also always represented in its own line.

To add comments to a pseudo-code representation, one, first, adds the string "//" and, finally, follows it by the desired comment. These comments are restricted to the line where the string "//" appears. Moreover, comments can only be represented at the end of lines of code, not in the middle of them.

To show that a group of variables is closely associated with each other, usually, one uses the same prefix for all of these variables but a different ending. This ending is usually represented in subscript, so that one can easily understand it as an index.

AES is not a block-cipher but a NIST's Standard [37] that describes the way in which the block-cipher designated by Rijndael [16] must be used to be in accordance with the Standard. In the art, however, it is usual to refer to the block-cipher Rijndael with both a block's length and a key's length equal to 128 bits by "AES". As so, from hereinafter, if nothing is said on contrary, the designation AES is used to refer to the block-cipher Rijndael with both a block's length and a key's length equal to 128 bits.

To distinguish where a Definition, a Proposition, a Theorem, an Experiment, or a "snippet" of code terminates and the textual description resumes, the symbol "**|**" is used. This symbol, however, may not be used when the limits of a Definition, of a Proposition, of a Theorem, of an Experiment, or of a "snippet" of code are well understood. For instance, when this information is presented inside a Table.

3.2 – SCA1 Model

This Section is subdivided into four Sections. The first Section (Section 3.2.1) describes oracles; the second Section (Section 3.2.2) describes compilers; the third Section (Section 3.2.3) describes some particulars of side-channel attacks; and the fourth Section (Section 3.2.4) describes Experiments.

3.2.1 – Oracles

In the SCA1 Model, the access of the adversary to the instance of the cryptosystem to test is done through the use of oracles. Accordingly, if the cryptosystem to test is characterized by two algorithms, then it is considered that the oracles in charge of the execution of the instance of the default algorithm of the cryptosystem to test are the default-oracles and the oracles in charge of the execution of the instance of the inverse algorithm of the cryptosystem to test are the inverse oracles. If the cryptosystem to test is characterized by one algorithm only, then it is assumed that the oracles in charge of its execution are the default-oracles.

In case the cryptosystem to test is an authentication cryptosystem, the default-oracles and the inverse-oracles represent, respectively, the oracles in charge of the creation of the authentication's tags and the oracles in charge of the verification of the integrity of the authentication's tags; and in case the cryptosystem to test is an encryption cryptosystem, the default-oracles and the inverse-oracles represent, respectively, the oracles in charge of the creation of the ciphertexts and the oracles in charge of both the decryption and the confirmation of the ciphertexts' integrity.

A *mute-oracle* is a modified version of either a default-oracle or an inverse-oracle. This oracle is characterized by differing from the original version of the latter oracles by its output. Accordingly, this modified oracle, instead of outputting the results of the requests made to it, (after accomplishing the processing necessary to accomplish these requests) just outputs a value (value requestAccomplished) informing that the requests have been accomplished.

The aim of mute oracles is modelling cryptographic hardware that is used in a setting in which this hardware can be led to process the calls made to it by the adversary but in which the latter is not capable of seeing the answers for his requests to this hardware. Hence, in this case, the aim of the adversary is not

seeing the answers for the requests made by him to mute oracles but it is undertaking SCAs against the processing that is undertaken by these oracles.

As so, in a model in which side-channel attacks are not defined, the mute-oracles are moot. This is, in a model in which side-channel attacks are not defined, the adversary, instead of having access to a mute version of some oracle, is simply forbidden to access the default version of the latter oracle.

Here, it is assumed that all the functions (which are defined by a cryptosystem) that may be attacked by the use of SCAs must be represented in its own line.

To verify whether or not the resources of an adversary are enough for him to accomplish an attack, it is assumed that the pseudo-code representation of his attack must be unambiguous. This is, all the steps that are accomplished by the adversary must be unambiguously represented. Accordingly, it is considered that the adversary is not allowed to use any "pseudo-code program" to help him into accomplishing his attack.

By the contrary, the oracles can use "pseudo-code programs" to help them into accomplishing their tasks, in case these tasks do not represent the execution of either the default or the inverse algorithms of the instance of the cryptosystem to test. Accordingly, here, whenever an oracle needs to maintain large amounts of information, it is considered that it uses an external database "program", which commands are textual commands.

3.2.2 – Compilers

In the SCA1 Model, it is assumed that the compilers associated with low-level programming languages compile the source code of programs as it is, whether or not they find instructions (dummy instructions) or variables (dummy variables) that never affect the output of the program.

By the contrary, it is assumed that the compilers associated with high-level programming languages, in order to make the compiled programs more efficient, first, create a temporary source code that is equal to the original one, secondly, delete both the dummy instructions and the dummy variables from the temporary source code, thirdly, include in the temporary source code the information regarding when variables are no longer needed by the program (so that the "run-time environment" frees up the memory associated with these variables, as soon as they are not needed anymore by the program) and, finally, compile the resultant temporary source code.

This deletion of both dummy instructions and dummy variables by the compilers of high-level programming languages is assumed as being dumb. This is, this deletion is nothing more than the deletion of lines of code that are superfluous for the correct execution of the program. As so, it is assumed that these compilers do not attempt to understand the purpose of the program in order to find out what is the best way to code it.

3.2.3 – Side-channel attacks

In this description, four types of leaked information are described (timing-information, power-information, electromagnetic-information, and optical information). However, as not all cryptographic hardware has optical components, SCAs based on optical-information cannot be undertaken against all cryptographic hardware.

Hence, this type of SCAs will not be referred again from hereinafter. Except to warn one that cryptographic hardware with optical components should be created in such a way that the information displaced by its optical components is not correlated with the processing undertaken by it (see [17]).

The SCA1 Model is a theoretical model, as so, one is not able to measure what is the amount of information leaked by each function. For this reason, this model requires that the functions made available by the cryptographic hardware's APIs are securely implemented against SCAs. More precisely, this model requires that the cryptographic primitives are securely implemented against both DSCAs and SSCAs and the non-cryptographic functions are securely implemented against SSCAs.

Accordingly, here, as one is not capable of verifying whether or not a function leaks information through side-channels, one can only base his SCAs on the leaked information perceived in a theoretical model.

As so, here, one can only verify whether or not the processing paths of an algorithm take all the same time to process (timing-information) and whether or not the non-cryptographic functions defined by the latter algorithms are used in such a way that one does not profit from undertaking DSCAs (based on either power-information or electromagnetic-information) against these functions to acquire unknown information.

Attacks based on power-information are better studied in the art than attacks based on electromagnetic-information. For instance, currently, it is not yet known how to achieve code that is well protected against electromagnetic-analysis but it is already known how to create code that achieves a good protection against power-analysis. Accordingly, an example of an implementation that is secure against power-analysis but not against electromagnetic-analysis is described in [3].

For this reason, besides the software protections that may be implemented in the code, to protect cryptosystems against electromagnetic-analysis, one needs to both physically protect (by use of shielding) against SCAs the hardware on which these cryptosystems are implemented and use this hardware at some distance from a possible source of attack.

As the protection against electromagnetic-analysis cannot be solely based on software, from hereinafter, when one refers to DSCAs, one is assuming that these attacks are based on power-information.

Attacks based on power-information are characterized by the measurement of the power that is consumed when instructions are processed. This is due to the way the microprocessor "sees" data. Indeed, microprocessors distinguish the bits 0 and 1 because these are represented by a different power-consumption fingerprint. Commonly, the bit 0 is encoded as a level of 0 Volts and the bit 1 is encoded as a level of either 3 or 5 Volts.

In this model, it is assumed that the attacker is capable of distinguishing what is the instruction that is processed at each point of the processing path. This is so because, here, it is considered that each instruction has a different fingerprint in what regards power-information. In addition to the information that characterizes a function, it is considered that the attacker is capable of measuring the power that is consumed when data is put on the data bus.

The power measured when data is put on the data bus, however, is not accurate enough for one to directly infer the values that are put on it. This is so because these measurements depend on the position of the bits 1 inside the word of data that is put on the data bus and, to a lesser extent, on the data already stored on memory.

Accordingly, in a DSCA based on power-information, the adversary changes the value known to him (value kValue) in such a way that he can correlate this value with both the value to infer (value tValue) and the value that results from the execution of the operation to analyze (value resultantValue). The correlation between

the values kValue, tValue, and resultantValue is based on the leakages that the attacker acquires when these values are put on the data bus.

In conclusion, in the SCA1 Model, the adversary bases his SSCAs on timing-information and his DSCAs on power-information. Moreover, the DSCAs are based on the power-information that is measured when data is put on the data bus.

3.2.4 – Experiments

The name of an experiment is composed of three strings joined together by the symbol "-". The first string aims identifying the goal of the experiment (in a generic way, this string is represented by the string "GOAL"); the second string aims identifying the attack associated with the experiment (in a generic way, this string is represented by the string "ATK"); and the third string aims identifying the model (SCA0 Model or SCA1 Model) associated with the experiment (in a generic way, this string is represented by the string "MODEL"). Accordingly, in a generic way, the name of an experiment is represented by "GOAL-ATK-MODEL".

In the experiments, the attack that is undertaken by the adversary is represented as a function (attacker's attack-function). This is so to facilitate showing that the adversary, at the beggining of the attack, receives paramaters associated with the experiment and, at the end of it, returns information.

Besides the inputs and the outputs, the header that represents the attacker's attack-function also includes (in subscript) a string that identifies the attack and (in superscript) a string that identifies all the oracles to which the adversary has access during the attack.

The pseudo-code representation of the attacker's attack-function is represented inline in the pseudo-code representation of experiments. This is done just to facilitate the representation of attacks and, hence, should not be confused with the representation of inline functions. This is, all the memory that is allocated inside the attacker's attack-function is local to this function and this function can only access memory not local to it, in case an alias to this memory is passed as parameter to it.

Throughout the present description, it is considered that the adversary wins a game associated with both a goal and an attack when he breaks the security of the cryptosystem to test in what regards the experiment associated with both the latter goal and the latter attack. Moreover, whenever a way to break the security of the cryptosystem to test is discovered, in what regards a particular game, it is said that the cryptosystem to test is insecure in accordance with that particular game. Otherwise, it is said that the cryptosystem to test "appears" to be secure in accordance with that particular game.

The relation between each two experiments (whether or not these experiments are associated with the same security notion) may be an implication relation, a dominance relation, an equivalent relation, or an incomparable relation.

An implication relation between two experiments EXP1 and EXP2 is denoted by EXP1=>EXP2, when the experiment EXP1 implies the experiment EXP2. More precisely, the implication relation EXP1=>EXP2 means that any scheme that is secure in accordance with the experiment EXP1 is also secure in accordance with the experiment EXP2.

A dominance relation between the experiments EXP1 and EXP2 is denoted by EXP1>EXP2, when the experiment EXP1 dominates the experiment EXP2. More precisely, the dominance relation EXP1>EXP2 is defined as follows: EXP1>EXP2, if both EXP1=>EXP2 and EXP1<≠EXP2, where EXP1=>EXP2 means that any

scheme that is secure in accordance with the experiment EXP1 is also secure in accordance with the experiment EXP2; and EXP1<≠EXP2 means that there are schemes that are secure in accordance with the experiment EXP2 but not in accordance with the experiment EXP1.

An equivalent relation between the experiments EXP1 and EXP2 is denoted by EXP1<=>EXP2. This relation is defined as follows: EXP1<=>EXP2, if both EXP1=>EXP2 and EXP1<=EXP2, where EXP1=>EXP2 means that any scheme that is secure in accordance with the experiment EXP1 is also secure in accordance with the experiment EXP2; and EXP1<=EXP2 means that any scheme that is secure in accordance with the experiment EXP2 is also secure in accordance with the experiment EXP1.

An incomparable relation between the experiments EXP1 and EXP2 is denoted by (EXP1≠>EXP2 ∧ EXP1<≠EXP2). This relation is defined as follows: (EXP1≠>EXP2 ∧ EXP1<≠EXP2), where EXP1≠>EXP2 means that there are schemes that are secure in accordance with the experiment EXP1 but not in accordance with the experiment EXP2; and EXP1<≠EXP2 means that there are schemes that are secure in accordance with the experiment EXP2 but not in accordance with the experiment EXP1.

In the figures, a connection between two experiments represents either a dominance or an equivalent relation. A dominance relation is represented by a connection characterized by a connector with an arrow represented in only one of the connector's two extremities[57]; and an equivalent relation is represented by a connection characterized by a connector with an arrow represented in each one of the connector's two extremities. Moreover, in the figures, a path represents a chain of n experiments that are characterized by an implication relation between each two consecutive experiments. This means that for all the n experiments that constitute a path, the experiment i implies the experiment i+1, with i=1...n-1. Furthermore, lack of a connection or of a path between two experiments indicates incomparable experiments.

3.3 – Symmetric authentication setting

A *symmetric authentication cryptosystem* (also designated by *symmetric authentication scheme,* by *message authentication code, or, simply, by* MAC) - *SA=(K, T, V)* - consists of the following three algorithms:

- the randomized *key generation algorithm – K* -, which takes as input a security parameter, $k \in \mathbb{N}$, and returns a key, value K. This algorithm is represented by $K \leftarrow R \leftarrow K(k)$;
- the stateful or randomized *tagging algorithm – T* -, which takes as input the key, value K, and a message, value M, and returns a tag, value τ. This algorithm is represented by $\tau := T_K(M)$, where $M \in \{0,1\}^*$ and $\tau \in \{0,1\}^L$ [58];
- the deterministic and stateless *verification algorithm – V* -, which takes as input the key, value K, a message, value M, and a tag, value τ, and returns 1, if the tag is valid, or 0, otherwise. This algorithm is represented by $answer := V_K(M, \tau)$, where $answer \in \{0,1\}$.

57 In a dominance relation, the connector's extremity represented by an arrow connects to the experiment that is dominated.

58 In this definition, the length of the tag - given by the value L - is defined as being of an *a priori* fixed length, whatever is the value and the length of the message that is authenticated by the symmetric authentication scheme. In spite of this, nothing forbids that the length of the tag is dependent on either, or both, the length or the value of the message to authenticate.

Besides the three algorithms *K*, *T*, and *V* a symmetric authentication scheme is also characterized by a *correctness requirement*, which states that for each key K (such that K←R—*K*(k)) and any message M∈{0,1}*, if T_K(M) returns τ, then V_K(M, τ) returns 1.

In the symmetric authentication setting, it is assumed that the symmetric authentication schemes (in accordance with their definition) only authenticate messages whose length is greater than zero bits (|message|>0 bits).

A *symmetric authentication oracle* (*SA-oracle*) is a default-oracle that embodies the tagging algorithm of a symmetric authentication scheme. A *symmetric verification oracle* (*SV-oracle*) is an inverse-oracle that embodies the verification algorithm of a symmetric authentication scheme.

Throughout the present description, to facilitate the description of the experiments defined in this setting, it is considered that the (not mute) tagging oracles defined in an experiment always add the pair of values, tag and message, associated with the requests made to them, to a database (database S_DB) that is shared by all the tagging oracles and by all the verification oracles defined in the experiment. This database is used with the aim that the verification oracles defined in the experiment only return information regarding the authentication of a pair of values, tag and message, in case this pair of values is not in the shared database.

Accordingly, the only aim of the database S_DB is providing a way to represent in pseudo-code the rule that states that the default-verification-oracles only return information as answer, in case the pair of values provided as input to the requests made to them is not associated with a request made to one of the tagging-oracles defined in the experiment. In this way, it is assumed that this database is virtual and, therefore, no restriction is imposed to its length.

As the tags created by mute tagging oracles are not seen by the adversary, the pairs of values, tag and message, associated with these tags are not added to the database S_DB.

As there are experiments in which more than one instance of the scheme to test is concurrently tested, these experiments define a different database S_DB for each instance of the scheme to test. In these cases, the database S_DB is represented by S_DB_{Ki}, where the value Ki identifies the instance of the scheme to test with which the database is associated. Furthermore, the shared databases are created at the beggining of the experiments that define them.

A *default (symmetric) verification oracle (SV-oracle)* is a verification oracle that, at input, accepts a pair of values composed of a tag (value τ) and of a message (value M). The output of this oracle is one one-bit value (value answer) that informs whether or not the integrity of the pair of values τ and M is confirmed.

Accordingly, the default and mute versions of the oracle SV-oracle, when this oracle is associated with a symmetric key Ki, are represented (in pseudo-code) by, respectively:

$SV\text{-}oracle_{Ki}(M, \tau)$
{
answer:=$V_{Ki}(M, \tau)$;
if ((M, τ) is stored into the database S_DB_{Ki}) then
return requestAccomplished;
return answer;
}
	$mute\text{-}SV\text{-}oracle_{Ki}(M, \tau)$
	{
	answer:=$V_{Ki}(M, \tau)$;
	return requestAccomplished;
	}

The SCAs that may be undertaken against either of the oracles SV-oracle or mute-SV-oracle are restricted to the execution of the instance of the verification algorithm of the cryptosystem to test (this is, are restricted to the execution of the algorithm $V_{Ki}(M, \tau)$).

A *chosen (symmetric) tagging oracle (SA-oracle)* is a tagging oracle that, at input, accepts one value (value M) - the message to authenticate -. The output of this oracle is the tag (value τ) that results from the authentication of the value M.

Accordingly, the default and mute versions of the oracle SA-oracle, when this oracle is associated with a symmetric key Ki, are represented (in pseudo-code) by, respectively:

$SA\text{-}oracle_{Ki}(M)$	$mute\text{-}SA\text{-}oracle_{Ki}(M)$
{	{
$\tau := T_{Ki}(M)$;	$\tau := T_{Ki}(M)$;
append (M, τ) to database S_DB_{Ki};	return requestAccomplished;
return τ;	}
}	

The SCAs that may be undertaken against either of the oracles SA-oracle or mute-SA-oracle are restricted to the execution of the instance of the tagging algorithm of the cryptosystem to test (this is, are restricted to the execution of the algorithm $T_{Ki}(M)$).

The experiments that are defined in this paper for the symmetric authentication setting are defined under one of the following attacks: S*trong-ChosenMessage-Attack* (*SCMA*) or *ChosenMessage-Attack* (*CMA*).

A *Strong-ChosenMessage-Attack* (S*CMA*) is an attack in which the adversary is allowed to make requests to default-verification-oracles and to chosen-tagging-oracles; a *ChosenMessage-Attack* (*CMA*) is an attack in which the adversary is allowed to make requests to mute-default-verification-oracles and to chosen-tagging-oracles. In any of the above referred attacks, the adversary is always allowed to make requests to the SCA-oracle, in addition to the requests to the oracles defined for each attack.

4 – Symmetric Authentication Setting

This Section is divided into two Sections. The first Section (Section 4.1) is characterized by a formal description of the security notion designated by "Existential Unforgeability" and the second Section (Section 4.2) is characterized by the presentation of the results of studying the security of the symmetric authentication schemes designated by CBC-MAC (see [33]), DMAC (see [8]), EAX-MAC (see [26]), HMAC (see [36]), OMAC (see [44]), PMAC (see [11]), RIPE-MAC (see [1]), RMAC (see [8]), TMAC (see [22]), XECB$-MAC (see [46]), and XMACR (see [34]) in what regards the latter security notion.

4.1 – Security Notions

In this Section, a new security notion is introduced, which aim is testing the unforgeability of the tags of symmetric authentication schemes. More precisely, the aim of this security notion is testing if an adversary is capable of creating a new pair of values, tag and message, for which the integrity is confirmed. This security notion is designated by "Existential Unforgeability".

EXISTENTIAL UNFORGEABILITY. The aim of this security notion is testing whether or not an adversary is capable of creating a new pair of values, tag and message, for which the integrity is confirmed. This security notion defines experiments in which the adversary runs in one stage, at the end of which he comes up with a pair of values composed of a tag (value τ) and of a message (value M). The adversary wins the game if the latter pair of values is characterized by both having its integrity confirmed and being different from any pair of values, tag and message, associated with a request to a tagging-oracle defined in the experiment. The authentication scheme is "good" if "reasonable" adversaries cannot win the game with a probability that is not negligible.

Definition 1 [EU-ATK-SCA1, with ATK$\in${SCMA, CMA}] Let *SA=(K, T, V)* be a symmetric authentication scheme; let $k\in\mathbb{N}$; let the values M and τ be, respectively, the message and the tag returned by the adversary at the end of the experiment; let the value answer represent the bit that informs whether or not the integrity of the pair of values M and τ is confirmed; let the oracles SCA-oracle, chosen-tagging-oracle, default-verification-oracle, and mute-default-verification-oracle be represented by, respectively, the oracles SCA(...), $T0_K(.)$, $V0_K(.,.)$, and $V1_K(.,.)$; let $\boldsymbol{A_{scma}}$ be an adversary that is allowed to make requests to the oracles SCA (...), $T0_K(.)$, and $V0_K(.,.)$; and let $\boldsymbol{A_{cma}}$ be an adversary that is allowed to make requests to the oracles SCA (...), $T0_K(.)$, and $V1_K(.,.)$. Now, consider the following experiments:

```
Experiment Expt_{SA, Aatk}^{eu-atk-sca1}(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack
        if (atk=scma) then (M, τ)←A_scma^{T0_K(.), V0_K(.,.), SCA(...)}(k, create);
        else if (atk=cma) then (M, τ)←A_cma^{T0_K(.), V1_K(.,.), SCA(...)}(k, create);
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
    if ((M, τ) is stored into the database S_DB_K) then return ⊥;
    answer:=V_K(M, τ);
    if (answer=1) then return 1;
    return 0;
}
```

Here, it is assumed that the pair of values, tag (value τ) and message (value M), that is returned by the adversary at the end of the experiment is not stored into the database S_DB_K.

The advantages of the adversaries A_{atk} may be defined via:

$$\mathbf{Adv}_{SA,\ Aatk}^{\text{eu-atk-sca1}}(k)=\Pr[\mathbf{Expt}_{SA,\ Aatk}^{\text{eu-atk-sca1}}(k)=1].$$

The advantage functions of a symmetric authentication scheme, in what regards the experiments eu-scma-sca1 and eu-cma-sca1, may be defined, respectively, as follows:

$$\mathbf{Adv}_{SA}^{\text{eu-scma-sca1}}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V0}, q_{SCA})=\max_{Ascma}\{\mathbf{Adv}_{SA,\ Ascma}^{\text{eu-scma-sca1}}(k)\};\ \text{and}$$
$$\mathbf{Adv}_{SA}^{\text{eu-cma-sca1}}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V1}, q_{SCA})=\max_{Acma}\{\mathbf{Adv}_{SA,\ Acma}^{\text{eu-cma-sca1}}(k)\};$$

where: the value t represents the maximum running time (in seconds) that the adversary A_{atk} has available to accomplish the attack; the value pp represents the maximum processing power (in cycles per second) that the adversary A_{atk} has available to accomplish the attack; the value sc represents the maximum storage capacity (in bits) that the adversary A_{atk} has available to accomplish the attack; the values q_{SCA}, q_{T0}, q_{V0}, and q_{V1} represent the maximum number of queries that the adversary A_{atk} may make to, respectively, the oracles SCA (...), $T0_K(.)$, $V0_K(.,.)$, and $V1_K(.,.)$; and the value μ_{T0} represents the maximum number of tag-bits that the adversary A_{atk} may see in response to the requests he makes to the oracle $T0_K(.)$.

Whenever one of the values q_{SCA}, q_{T0}, q_{V0}, q_{V1}, or μ_{T0} is not defined in one of the above advantage functions, it is assumed that this value, in this advantage function, is equal to zero. **|**

This security notion is an adaptation of the security notion designated by "Strong Existential Unforgeability" introduced in [32]. The security notion introduced here differs from the security notion introduced in [32] because in the latter security notion the oracle SCA-oracle is not defined.

Theorem 1 [EU-SCMA-SCA1>EU-CMA-SCA1] If the symmetric authentication scheme $\prod=(K, T, V)$ is secure in accordance with the experiment EU-SCMA-SCA1, then $\prod$ is secure in accordance with the experiment EU-CMA-SCA1. But the security of $\prod$ in accordance with the experiment EU-CMA-SCA1 does not imply the security of $\prod$ in accordance with the experiment EU-SCMA-SCA1.

Proof: see Appendix B. **|**

In the fig. 2, the relations between the experiments associated with the security notion designated by "Existential Unforgeability" are represented.

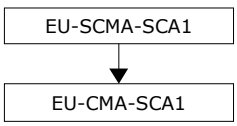

Fig. 2

At this point of the description, the reader is invited to read the Appendix C.

4.2 – Studied schemes

The security in the SCA1 Model of the following symmetric authentication schemes has already been studied: CBC-MAC (see [33]), DMAC (see [8]), EAX-MAC (see [26]), HMAC (see [36]), OMAC (see [44]), PMAC (see [11]), RIPE-MAC (see [1]), RMAC (see [8]), TMAC (see [22]), XECB$-MAC (see [46]), and XMACR (see [34]). For a pseudo-code representation of the attacks against the above referred schemes, see Appendix C.

The above referred schemes have been studied in a setting in which the messages to authenticate are both well distributed in their domain and may have a variable length (in accordance with both the security notion "Existential Unforgeability" and the definition of the studied scheme).

Two tables are used to resume the information regarding the security of the above referred schemes. The first table (Table 1) highlights the least "powerful" unforgeability experiment in accordance with which each one of the studied schemes is insecure; and the second table (Table 2) highlights the most "powerful" unforgeability experiment in accordance with which each one of the studied schemes "appears" to be secure.

Accordingly, a blank cell in Table 1 means that the scheme associated with this cell "appears" to be secure in accordance with the most "powerful" experiment relating to the security notion introduced in this paper; and a blank cell in Table 2 means that the scheme associated with this cell is insecure in accordance with the least "powerful" experiment relating to the security notion introduced in this paper.

Scheme	Unforgeability
CBC-MAC	EU-CMA-SCA1
DMAC	EU-CMA-SCA1
EAX-MAC	EU-CMA-SCA1
HMAC	
OMAC	EU-CMA-SCA1
PMAC	EU-CMA-SCA1
RIPE-MAC	
UMAC	EU-CMA-SCA1
TMAC	EU-CMA-SCA1
XECB$-MAC	EU-CMA-SCA1
XMACR	EU-CMA-SCA1

Table 1

Scheme	Unforgeability
CBC-MAC	
DMAC	
EAX-MAC	
HMAC	EU-SCMA-SCA1

OMAC	
PMAC	
RIPE-MAC	EU-SCMA-SCA1
UMAC	
TMAC	
XECB$-MAC	
XMACR	

Table 2

With reference to the Tables 1 and 2, one can see that of the eleven studied schemes only HMAC and RIPE-MAC "appear" to be secure in accordance with the most "powerful" experiment (experiment EU-SCMA-SCA1) associated with the security notion designated by "Existential Unforgeability". Moreover, one can see that the other nine schemes are completely insecure in accordance with the latter security notion.

5 – Conclusions and open problems

The implementation of cryptosystems in hardware is usually done on general purpose cryptographic hardware (for instance, on general purpose cryptographic smartcards). This type of hardware is characterized by providing the resources, in the form of APIs, to be used in the implementation of cryptosystems. These APIs are usually coded to be secure against side-channel attacks.

Nonetheless, it was shown here that there is a gap of security, in what regards SCAs, between the security of the functions made available by the cryptographic hardware's APIs and the security of the cryptosystems that use these functions. Moreover, it was shown here that an attacker can easily take profit of this gap to break the security of cryptosystems. Specifically, he can use this gap to break the security of most part of the cryptosystems that do not take it into account (see Appendix C).

Accordingly, one is lead to the conclusion that SCAs are not exclusively used to acquire the key of the instance of the cryptosystem to attack but indeed these attacks may be used to acquire any intermediary variable that the cryptosystem leaks. As so, one concludes that the security of cryptosystems against SCAs has to be tested as a whole and not as the sum of the parts. For this reason, a new model (SCA1 Model) was introduced in this description, in order to help the people that create cryptosystems to test them as a whole against SCAs.

The SCA1 Model aims being the base for the creation of security notions that take side-channel attacks into account. According to it, cryptosystems can be tested against an attacker that is able to undertake both differential side-channel attacks (DSCAs) and simple side-channel attacks (SSCAs).

In addition, this model tries to resemble as much as possible the implementation of cryptosystems in general purpose cryptographic hardware, which, in the real-world, is the type of hardware commonly used to implement cryptosystems.

Accordingly, the SCA1 Model aims diminishing the gap between the people who design algorithms, schemes, and software and the people who create the hardware in which the latter algorithms, schemes, and software are implemented. As so, only the cryptosystems that are proved secure in this model should be used in real-world implementations.

For this reason, the security of cryptosystems in accordance with the SCA1 Model should be required by the entities that create cryptographic standards, as, for instance, NIST. Indeed, the future NIST's "Block-Ciphers' Modes of Operation" Standard[59] should require that the cryptosystems that are chosen to be included in the Standard are secure in the SCA1 Model.

In this description, a new security notion designated by "Existential Unforgeability" was introduced, being its aim testing (in the SCA1 Model) the unforgeability of the tags of symmetric authentication schemes.

During the creation of this paper, some symmetric authentication schemes were studied, being the results of this study very disappointing (see Section 4.2). This is so because of the 11 symmetric authentication schemes studied, only two of them, HMAC and RIPE-MAC, "appear" to be secure in accordance with the most "powerful" experiment associated with the security notion introduced in this description. However, as none of the latter two schemes is randomized, neither of them creates tags that can be used to conceal the authenticated message, in case the latter is not appended to the tag.

Additionally, RIPE-MAC leaks enough information through side-channels for the attacker to acquire intermediary values of the authentication, which, later, may be used by him to break the security of RIPE-MAC when this scheme is tested in accordance with other security notions defined in the SCA1 Model (see the long version of the paper).

Accordingly, in what regards the symmetric authentication setting, one is lead to the conclusion that there is the need to create new cryptosystems, which not only are proved secure in accordance with the security notion introduced in this paper but are also both randomized and leakage-immune.

With reference to what has been said above, one concludes that the model introduced here should be very valuable in the protection of cryptosystems against SCAs.

6 - Acknowledgments

I would like to thank Mads Rasmussen, first, for reading my early informal drafts, secondly, for challenging me to rewrite them formally and, finally, for pointing me to some good examples of well written formal papers.

7 - Bibliography

[1] Alfred J. Menezes, Paul C. van Oorschot, "Handbook of Applied Cryptography", CRC Press, 1997, ISBN 0-8493-8523-7.

Available at http://www.cacr.math.uwaterloo.ca/hac/

59 See http://csrc.nist.gov/CryptoToolkit/modes/ .

[2] Christophe Clavier, Jean-Sébastien Coron, Nora Dabbous, "Differential Power Analysis in the Presence of Hardware Countermeasures", In Ç.K. Koç and C. Paar, Eds., Cryptographic Hardware and Embedded Systems - CHES 2000, vol. 1965 of Lecture Notes in Computer Science, pp. 252-263, Springer-Verlag, 2000.

Available at http://www.gemplus.com/smart/r_d/publications/pdf/CCD00dpa.pdf

[3] Dakshi Agrawal, Bruce Archambeault, Suresh Chari, Josyula R. Rao, "Advances in Side-Channel Cryptanalysis, Electromagnetic Analysis and Template Attacks", CryptoBytes Technical Newsletter, Volume 6, No. 1 - Spring 2003.

Available at http://www.rsasecurity.com/rsalabs/cryptobytes/CryptoBytes_March_2003_lowres.pdf

[4] David Brumley, Dan Boneh, "Remote Timing Attacks are Practical", In Proceedings of the 12th Usenix Security Symposium, 2003.

Available at http://www.usenix.org/publications/library/proceedings/sec03/tech/brumley/brumley.pdf

[5] Donald E. Knuth, "The Art of Computer Programming, Volume 1 (Fundamental Algorithms), Third Edition", Addison Wesley, 2002, ISBN 0201896834.

Available at http://www.amazon.com

[6] Donald E. Knuth, "The Art of Computer Programming, Volume 3 (Sorting and Searching), Second Edition", Addison Wesley, 2002, ISBN 0201896850.

Available at http://www.amazon.com

[7] Eli Biham, "How to Forge DES-Encrypted Messages in 2^28 Steps", Technical Report CS 885, August 1996.

Available at http://www.cs.technion.ac.il/users/wwwb/cgi-bin/tr-get.cgi/1996/CS/CS0884.ps

[8] Éliane Jaulmes, Antoine Joux and Frédéric Valette, "On the Security of Randomized CBC-MAC Beyond the Birthday Paradox Limit - A new Construction".

Available at http://eprint.iacr.org/2001/074.ps

[9] Gael Hachez, Franþois Kouene, Jean-Jacques Quisquater, "Timing Attack: What can be achieved by a powerful adversary", In Proceedings of the 20th Symposium on Information Theory in the Benelux, May 1999.

Available at http://www.dice.ucl.ac.be/crypto/publications/Timing.pdf

[10] Ian Hickman, "Oscilloscopes, Fifth Edition", Newnes, 2001, ISBN 0750647574.

Available at http://www.amazon.com

[11] J. Black, P. Rogaway, "A Block-Cipher Mode of Operation for Parallelizable Message Authentication", In Advances in Cryptology - EUROCRYPT 2002, International Conf. on the Theory and Applications of Cryptographic Techniques, 2002, Proc. LNCS 2332 Springer 2002.

Available at http://link.springer.de/link/service/series/0558/bibs/2332/23320384.htm

[12] J.-F. Dhem, F. Koeune, P.-A Leroux, P. Mestré, J.-J. Quisquater, J.-L. Willems, "A practical implementation of the timing attack", In Proceedings of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998), LNCS 1820.

Available at http://www.dice.ucl.ac.be/crypto/publications/1998/TimeAttack.pdf

[13] Jean-Sébastien Coron, Louis Goubin, "On Boolean and Arithmetic Masking Against Differential Power Analysis", In Ç.K. Koç and C. Paar, Eds., Cryptographic Hardware and Embedded Systems - CHES 2000, vol. 1965 of Lecture Notes in Computer Science, pp. 231-237, Springer-Verlag, 2000.

Available at http://www.gemplus.com/smart/r_d/publications/pdf/CG00mask.pdf

[14] Jee Hea, "Authenticated Encryption in the Public-Key Setting: Security Notions and Analyses", 2001.

Available at http://eprint.iacr.org/2001/079/

[15] Jee Hea An, Yevgeniy Dodis, Tal Rabin, "On the Security of Joint Signature and Encryption", In Advances in Cryptology -- EUROCRYPT 2002 Proceedings, Lecture Notes in Computer Science Vol. 2332 L. Knudsen ed, Springer-Verlag, 2002.

Available at http://link.springer.de/link/service/series/0558/bibs/2332/23320083.htm

[16] Joan Daemen, Vincent Rijmen, AES Proposal: Rijndael, NIST Submission, September 3, 1999.

Available at http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf

[17] Joe Loughry, David A. Umphress, "Information Leakage from Optical Emanations", In ACM Transactions on Information and System Security (TISSEC), Volume 5, Issue 3, August 2002.

Available at http://www.applied-math.org/optical_tempest.pdf

[18] Jonathan Katz, Moti Yung, "Complete characterization of Security Notions for Probabilistic Private-Key Encryption", In Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, ACM, 2000.

Available at http://doi.acm.org/10.1145/335305.335335

[19] Jonathan Katz, Moti Yung, "Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation", In Fast Software Encryption, 7th International Workshop, FSE 2000, 2000, Proceedings. Lecture Notes in Computer Science 1978 Springer 2001.

Available at http://link.springer.de/link/service/series/0558/bibs/1978/19780284.htm

[20] Joonsang Baek, Ron Steinfels, Yuliang Zheng, "Formal Proofs for the Security of Signcryption", In Public Key Cryptography, 5th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2002, 2002, Proceedings. Lecture Notes in Computer Science 2274 Springer 2002.

Available at http://link.springer.de/link/service/series/0558/bibs/2274/22740080.htm

[21] Josyula R. Rao, Pankaj Rohatgi, "EMpowering Side-Channel Attacks", May 11, 2001.

Available at http://eprint.iacr.org/2001/037.pdf

[22] Kaoru Kurosawa, Tetsu Iwata, "TMAC: Two-Key CBC MAC", Submission to NIST, 2002.

Available at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/tmac/tmac-spec.pdf

[23] Louis Goubin, "A Sound Method for Switching between Boolean and Arithmetic Masking", In Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, 2001, Proceedings. Lecture Notes in Computer Science 2162, Springer 2001.

Available at http://link.springer.de/link/service/series/0558/bibs/2162/21620003.htm

[24] Louis Goubin, Jacques Patarin, "DES and Differential Power Analysis - The "Duplication" Method", Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, 1999, Proceedings. Lecture Notes in Computer Science 1717 Springer 1999.

Available at http://link.springer.de/link/service/series/0558/bibs/1717/17170158.htm

[25] M. Bellare, A. Boldyreva, A. Desay, D Pointcheval, "Key-Privacy in Public-Key Encryption", In Advances in Cryptology - Asiacrypt 2001 Proceedings, Lecture Notes in Computer Science Vol. 2248, C. Boyd ed, Springer-Verlag, 2001.

Available at http://www.cs.ucsd.edu/users/mihir/papers/anonenc.html

[26] M. Bellare, P. Rogaway, D. Wagner, "A conventional Authenticated-Encryption Mode", October 2003.

Available at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/eax/eax-spec.pdf

[27] Matt Blaze, Whitfield Diffie, Ronald L. Rivest, Bruce Schneier, Tsutomu Shimomura, Eric Thompson, Michael Wiener, "Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security", January 1996.

Available at http://www.schneier.com/paper-keylength.pdf

[28] Mehdi-Laurent Akkar, Christphe Giraud, "An implementation of DES and AES, secure against some attacks", Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, 2001, Proceedings. Lecture Notes in Computer Science 2162 Springer 2001.

Available at http://link.springer.de/link/service/series/0558/bibs/2162/21620309.htm

[29] Mehdi-Laurent Akkar, Régis Bevan, "Power Analysis, What Is Now Possible ...", In Advances in Cryptology - ASIACRYPT 2000, 6th International Conference on the Theory and Application of Cryptology and Information Security, Proceedings. Lecture Notes in Computer Science 1976 Springer 2000.

Available at http://link.springer.de/link/service/series/0558/bibs/1976/19760489.htm

[30] Mihir Bellare, A. Desai, David Pointcheval, Phillip Rogaway, "Relations Among Notions of Security for Public-Key Encryption Schemes", Advances in Cryptology- Crypto 98 Proceedings, Lecture Notes in Computer Science Vol. 1462, H. Krawczyk ed, Springer-Verlag, 1998.

Available at http://www.cse.ucsd.edu/users/mihir/papers/relations.html

[31] Mihir Bellare, A. Desai, E. Jokipii, Phillip Rogaway, "A concrete Security Treatment of Symmetric Encryption", In the Proceedings of 38th Annual Symposium on Foundations of Computer Science, IEEE, 1997.

Available at http://www.cse.ucsd.edu/users/mihir/papers/sym-enc.ps

[32] Mihir Bellare, Chanathip Namprempre, "Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm", In Advances in Cryptology - Asiacrypt 2000 Proceedings, Lecture Notes in Computer Science Vol. 1976, T. Okamoto ed, Springer-Verlag, 2000.

Available at http://www.cse.ucsd.edu/users/mihir/papers/oem.html

[33] Mihir Bellare, Joe Kilian, Phillip Rogaway, "The security of the cipher block chaining message authentication code", In Journal of Computer and System Sciences, Vol. 61, No. 3, Dec 2000, pp. 362-399.

Available at http://www.cs.ucsd.edu/users/mihir/papers/cbc.pdf

[34] Mihir Bellare, Roch GuÚrin, Phillip Rogaway, "XOR MACs: New Methods for Message Authentication Using Finite Pseudorandom Functions", Advances in Cryptology - Crypto 95 Proceedings, Lecture Notes in Computer Science Vol. 963, D. Coppersmith ed, Springer-Verlag, 1995.

Available at http://www.cs.ucsd.edu/users/mihir/papers/xormacs.html

[35] National Institute of Standards and Technology, US Department of Commerce, Washington D.C., FIPS 180-1, "Secure Hash Standard (SHA-1)", April 1995.

Available at http://www.itl.nist.gov/fipspubs/fip180-1.htm

[36] National Institute of Standards and Technology, US Department of Commerce, Washington D.C., FIPS 198," The Keyed-Hash Message Authentication Code (HMAC)", March 2002.

Available at http://csrc.nist.gov/publications/fips/fips198/fips-198a.pdf

[37] National Institute of Standards and Technology, US Department of Commerce, Washington D.C., FIPS-197, "Advanced Encryption Standard (AES)", November 26, 2001.

Available at http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[38] Oded Goldreich, Shafi Goldwasser, Silvio Micali, "How to Construct Random Functions", In Journal of the ACM (JACM), Volume 33, Issue 4, October 1986.

Available at http://www.wisdom.weizmann.ac.il/~oded/ggm-jacm.pdf

[39] Paul C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," Advances in Cryptology - Crypto 96 Proceedings, Lecture Notes In Computer Science Vol. 1109, N. Koblitz ed., Springer-Verlag.

Available at http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf

[40] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, "Differential Power Analysis", In Advances in Cryptology - Crypto 99 Proceedings, Lecture Notes In Computer Science Vol. 1666, M. Wiener ed., Springer-Verlag, 1999.

Available at http://www.cryptography.com/resources/whitepapers/DPA.pdf

[41] Paul C. Kocher, Joshua Jaffe, Benjamin Jun, "Introduction to Differential Power Analysis and Related Attacks", 1998.

Available at http://www.cryptography.com/resources/whitepapers/DPATechInfo.PDF

[42] Steve Gilheany, "The Decline of Magnetic Disk Storage Cost Over the Next 25 Years", Version 22004v036, January 12, 2003.

Available at http://www.archivebuilders.com/whitepapers/22004p.pdf

[43] Suresh Chari, Charantjit Jutla, Josyula R. Rao, Pankaj Rohatgi, "A cautionary Note Regarding Evaluation of AES Candidates on Smart-Cards", February 1, 1999.

Available at http://csrc.nist.gov/CryptoToolkit/aes/round1/conf2/papers/chari.pdf

[44] Tetsu Iwata, Kaoru Kurosawa, "OMAC: One-Key CBC MAC", Submission to NIST, December 20, 2002.

Available at http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/omac/omac-spec.pdf

[45] Virgil D. Gligor, Pompiliu Donescu, "On message Integrity in Symmetric Encryption", Presented at the 1nd NIST Workshop on AES Modes of Operation, October 20, 2000. Available at

http://csrc.nist.gov/CryptoToolkit/modes/workshop1/papers/gligor-integrity-11-10-2000.pdf

[46] Virgil D. Gligor, Pompilu Donescu, "Fast Encryption and Authentication: XCBC Encryption and XECB Authentication Modes", Presented at the 2nd NIST Workshop on AES Modes of Operation, August 24, 2001. Available at

http://www.glue.umd.edu/afs/glue.umd.edu/home/enee/faculty/gligor/pub/NIST-submissionRev.ps

[47] Yevgeniy Dodis, Michael J. Freedman, Stanislaw Jarecki, Shabsi Walfish, "Optimal Signcryption from Any Trapdoor Permutation", January 28, 2004.

Available at http://eprint.iacr.org/2004/020.pdf

[48] Yuliang Zheng, "Digital Signcryption or How to Achieve Cost (Signature & Encryption) << Cost (Signature)+Cost(Encryption)", In Advances in Cryptology - Crypto'97, Lecture Notes in Computer Science, Vol. 1294, pp. 165-179, Springer-Verlag, 1997.

Available at http://www.pscit.monash.edu.au/~yuliang/pubs/c97-fnl-rvs.ps

Appendix A – Pseudorandom functions and pseudorandom permutations

In this Section, the following two security notions are introduced:

- "Pseudorandom Function Indistinguishability" - this security notion aims testing the indistinguishability of the outputs of the function to test from the outputs of a random member of the family of all the functions with both the same domain and the same codomain of the function to test[60];
- "Pseudorandom Permutation Indistinguishability" - this security notion aims testing the indistinguishability of the outputs of the permutation to test from the outputs of a random member of the family of all the permutations with a block's length equal to the block's length of the permutation to test.

PSEUDORANDOM FUNCTION INDISTINGUISHABILITY. The aim of this security notion is testing the pseudorandomness of the outputs of the function to test. The adversary defined in the experiment associated with this security notion is given access to both an oracle (oracle prf-to-test-oracle) that executes the instance of the function to test and an oracle (oracle real-and-random-prf-oracle) that outputs two strings in random order, one representing the result of executing the instance of the function to test and the other representing the result of executing a function selected uniformly at random from the family of all the functions with both the same domain and the same codomain of the function to test[61]. This adversary runs in one stage, at the end of which he comes up with a bit (value b') characterized by representing his guess regarding whether or not the first string returned (at each request) by the oracle that creates the challenge strings represents the result of executing the function to test. The adversary wins the game if the value b' is equal to the random value (value b) used to decide if the first string returned (at each request) by the latter oracle corresponds to the execution of the function to test. The function to test is a "good" pseudorandom function if "reasonable" adversaries cannot win the game with an increment of probability that is not negligible, regarding the probability of guessing the bit b at random.

Definition A.1 [PRFI-CMA-SCA1] Let vl be an integer defined by $0<min\_vl \leq vl \leq max\_vl$, with min_vl and max_vl integers; let vL be an integer defined by $0<min\_vL \leq vL \leq max\_vL$, with min_vL and max_vL integers;

60 Here, it is assumed that, for the same input, the length of the output of any of the members of the latter family of functions is always equal to the length of the output of the function to test.

61 Here, it is assumed that, for the same input, the length of the output of any of the members of the latter family of functions is always equal to the length of the output of the function to test.

let $F:Keys(F)x\{0,1\}^{vl} \rightarrow \{0,1\}^{vL}$ be the family of functions from which the instance of the function to test is chosen; let $k \in \mathbb{N}$; let $b \in \{0,1\}$; let the value b' (which is returned by the adversary at the end of the experiment) represent the guess of the adversary regarding whether or not the first string returned (at each request) by the oracle that creates the challenges strings corresponds to the execution of the function to test; let ρ_K represent the member of the family of functions $Rand_k^{vl \rightarrow vL}$ that is identified by the index K; let the oracles SCA-oracle, prf-to-test-oracle (see pseudo-code definition below), and real-and-random-prf-oracle (see pseudo-code definition below) be represented by, respectively, the oracles SCA(...), $F0_K(.)$, and $RARF_{K,b}(.)$; and let $\boldsymbol{A_{cma}}$ be an adversary that is allowed to make requests to the oracles SCA(...), $F0_K(.)$, and $RARF_{K,b}(.)$. Now, consider the following experiment:

```
Experiment Expt_{PRF, Acma}^{prfi-cma-sca1}(k)
{
    K←R—K(k);
    create database M_DB;
    ρK:=member K of Rand_k^{vl→vL};
    b:=getRandomBits(1);
    // begin of adversary's attack
        b'←A_cma^{RARFK,b(.), F0K(.), SCA(...)}(k, guess);
    // end of adversary's attack
    if (b=b') then return 1;
    return 0;
}
```

In the above pseudo-code representation, it is stated that ρ_K is a member of $Rand_k^{vl \rightarrow vL}$. However, depending on both the domain and the codomain of the function to test, ρ_K may be defined as a member of any of the families $Rand_k^{vl \rightarrow vL}$, $Rand_k^{vl \rightarrow L}$, $Rand_k^{l \rightarrow vL}$, or $Rand_k^{l \rightarrow L}$.

Here, the oracles real-and-random-prf-oracle (oracle $RARF_{K,b}(.)$) and prf-to-test-oracle (oracle $F0_K(.)$) are defined (in pseudo-code) by, respectively:

```
RARFK,b(m)
{
    str0:=FK(m);
    str1:=ρK(m);
    if (m is stored into the database M_DB) then return requestAccomplished;
    append m to the database M_DB;
    if (b=1) then return str0, str1;
    return str1, str0 ;
}
```

```
F0K(m)
{
    str:=FK(m);
    if (m is stored into the database M_DB) then return requestAccomplished;
    append m to the database M_DB;
    return str ;
}
```

Here, it is assumed that the adversary, for the same request, cannot see the answers of both the oracles $RARF_{K,b}(.)$ and $F0_K(.)$. For this to be represented in pseudo-code, a database (database M_DB) with information regarding all the values that have already been requested to any of the oracles $RARF_{K,b}(.)$ or $F0_K(.)$ is maintained by these oracles[62]. Accordingly, it is assumed that if a request to either of the oracles $RARF_{K,b}(.)$ or $F0_K(.)$ is already into the database M_DB, then these oracles, instead of returning the result of the request, only return the value requestAccomplished.

Above, it is assumed that the SCAs that may be undertaken against either of the oracles $RARF_{K,b}(.)$ or $F0_K(.)$ are restricted to the execution of the instance of the function to test (this is, are restricted to the execution of the algorithm $F_K(m)$). Moreover, it is assumed that the adversary makes at least one request to the oracle $RARF_{K,b}(.)$.

The advantage of the adversary A_{cma} may be defined via:

$$\mathbf{Adv}_{PRF,\,A_{cma}}^{\text{prfi-cma-sca1}}(k)=2*\Pr[\mathbf{Expt}_{PRF,\,A_{cma}}^{\text{prfi-cma-sca1}}(k)=1]-1.$$

The advantage function of a function, in what regards the experiment prfi-cma-sca1, may be defined as follows:

$$\mathbf{Adv}_{PRF}^{\text{prfi-cma-sca1}}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA})=\max_{A_{cma}}\{\mathbf{Adv}_{PRF,\,A_{cma}}^{\text{prfi-cma-sca1}}(k)\};$$

where: the value t represents the maximum running time (in seconds) that the adversary A_{cma} has available to accomplish the attack; the value pp represents the maximum processing power (in cycles per second) that the adversary A_{cma} has available to accomplish the attack; the value sc represents the maximum storage capacity (in bits) that the adversary A_{cma} has available to accomplish the attack; the values q_{SCA}, q_{F0}, and q_{RARF} represent the maximum number of queries that the adversary A_{cma} may make to, respectively, the oracles SCA(...), $F0_K(.)$, and $RARF_{K,b}(.)$; and the values μ_{F0} and μ_{RARF} represent the maximum number of pseudorandom-bits that the adversary A_{cma} may see in response to the requests that he makes to, respectively, the oracles $F0_K(.)$ and $RARF_{K,b}(.)$. ❙

This security notion is an adaptation of the security notion designated by "Polynomial-Time Statistical Tests For Functions" introduced in [38]. The security notion introduced here differs from the security notion "Polynomial-Time Statistical Tests For Functions" introduced in [38] because in the latter security notion: the

62 The only aim of the database M_DB is providing a way of representing in pseudo-code the rule that states that the adversary, for the same request, never sees the answers of both the oracles $RARF_{K,b}(.)$ and $F0_K(.)$. For this reason, it is assumed that this database is virtual and, therefore, no restriction is imposed to its length.

adversary (there called a distinguisher) is only given access to an oracle that creates challenge strings (oracle challenge-function-oracle) and not to an oracle that processes the function to test[63,64]; the outputs of the oracle challenge-function-oracle are the results of processing either the function to test or a function selected uniformly at random from the family of all the functions with both the same domain and the same codomain of the function to test, instead of being the random order output of the results of the execution of the latter two functions; and the oracle SCA-oracle is not defined.

Proposition A.1 [Any function with a sufficiently large minimum-output-length that is secure in the sense of PRFI-CMA-SCA1 may be transformed into a symmetric authentication scheme that is secure in the sense of EU-SCMA-SCA1] Let vl be an integer defined by $0<min\_vl\leq vl\leq max\_vl$, where the values min_vl and max_vl are integers; let vL be an integer defined by $0<min\_vL\leq vL\leq max\_vL$, where the values min_vL and max_vL are integers; let $F:Keys(F)x\{0,1\}^{vl}\rightarrow\{0,1\}^{vL}$ be a family of functions that is secure in the sense of PRFI-CMA-SCA1; and let the length of the smallest output of the function F (value min_vL) be sufficiently large (for instance, $min\_vL\geq 128$). Now, let the function F be used to create the symmetric authentication scheme $\Pi=(K, T, V)$ that is represented (in pseudo-code) by:

$T_K(M)$	$V_K(M, \tau)$
{	{
$\tau:=F_K(M)$;	$\tau\_:=F_K(M)$;
return τ;	if ($\tau=\tau\_$) then return 1;
}	return 0;
	}

Then, for $F_K(.)$ and Π as defined above,

$$\mathbf{Adv}_{SA,\,\Pi}^{\text{eu-scma-sca1}}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V0}, q_{SCA})\leq\mathbf{Adv}_{PRF,\,FK}^{\text{prfi-cma-sca1}}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA})+2^{\hat{}q-min\_vL}, \text{ with } q_{V0}=2^{\hat{}q}.$$

Proof: see Appendix B. **|**

PSEUDORANDOM PERMUTATION INDISTINGUISHABILITY. The aim of this security notion is testing the pseudorandomness of the outputs of the permutation to test. The adversary defined in the experiment associated with this security notion is given access to both an oracle (oracle prp-to-test-oracle) that executes the instance of the permutation to test and an oracle (oracle real-and-random-prp-oracle) that outputs two strings in random order, one representing the result of executing the instance of the permutation to test and the other representing the result of executing a permutation selected uniformly at random from the set of all the permutations with the same block's length of the permutation to test. This adversary runs in one stage, at the end of which he comes up with a bit (value b') characterized by representing his guess regarding whether

63 Notice that the access to an oracle that executes the instance of the function to test is required to assure that this function has the property of poly-random collections described in the Section 1.2 of [38].

64 Even if the version of the security notion introduced here appears to define an adversary that is more powerful than the one defined in [38], the author of this paper is not aware of any attack that, with a non-negligible probability, enables the adversary to win the version of the security notion introduced here but not the direct adaptation of the security notion introduced in [38].

or not the first string returned (at each request) by the oracle that creates the challenge strings represents the result of executing the permutation to test. The adversary wins the game if the value b' is equal to the random value (value b) used to decide if the first string returned (at each request) by the latter oracle corresponds to the execution of the permutation to test. The permutation to test is a "good" pseudorandom permutation if "reasonable" adversaries cannot win the game with an increase of probability that is not negligible, regarding the probability of guessing the bit b at random.

Definition A.2 [PRPI-CMA-SCA1] Let $P:Keys(P)x\{0,1\}^l \rightarrow \{0,1\}^l$ be the family of permutations from which the instance of the permutation to test is chosen; let $k \in \mathbb{N}$; let $b \in \{0,1\}$; let the value b' (which is returned by the adversary at the end of the experiment) represent the guess of the adversary regarding whether or not the first string returned (at each request) by the oracle that creates the challenges strings corresponds to the execution of the instance of the permutation to test; let the value l give the length of the block of the permutation to test; let ρ_K represent the member of the family of permutations $Perm_k^l$ that is identified by the index K; let the oracles SCA-oracle, prp-to-test-oracle (see pseudo-code definition below), and real-and-random-prp-oracle (see pseudo-code definition below) be represented by, respectively, the oracles SCA(...), $P0_K(.)$, and $RARP_{K,b}(.)$; and let $\boldsymbol{A_{cma}}$ be an adversary that is allowed to make requests to the oracles SCA(...), $P0_K(.)$, and $RARP_{K,b}(.)$. Now, consider the following experiment:

```
Experiment Expt_{PRP, Acma}^{prpi-cma-sca1}(k)
{
    K←R—K(k);
    create database M_DB;
    b:=getRandomBits(1);
    ρK:=member K of Perm_k^l;
    // begin of adversary's attack
        b'←A_cma^{RARPK,b(.), P0K(.), SCA(...)}(k, guess);
    // end of adversary's attack
    if (b=b') then return 1;
    return 0;
}
```

Here, the oracles real-and-random-prp-oracle (oracle $RARP_{K,b}(.)$) and prp-to-test-oracle (oracle $P0_K(.)$) are defined (in pseudo-code) by, respectively:

```
RARPK,b(m)
{
    str0:=PK(m);
    str1:=ρK(m);
    if (m is stored into the database M_DB) then return requestAccomplished;
    append m to the database M_DB;
    if (b=1) then return str0, str1;
    return str1, str0 ;
}
```

```
P0K(m)
{
    str:=PK(m);
    if (m is stored into the database M_DB) then return requestAccomplished;
    append m to the database M_DB;
    return str ;
}
```

Here, it is assumed that the adversary, for the same request, cannot see the answers of both the oracles $RARP_{K,b}(.)$ and $P0_K(.)$. For this to be represented in pseudo-code, a database (database M_DB) with information regarding all the values that have already been requested to any of the oracles $RARP_{K,b}(.)$ or $P0_K(.)$ is maintained by these oracles[65]. Accordingly, it is assumed that if a request to either of the oracles $RARP_{K,b}(.)$ or $P0_K(.)$ is already into the database M_DB, then these oracles, instead of returning the results of the request, only return the value requestAccomplished.

Above, it is assumed that the SCAs that may be undertaken against either of the oracles $RARP_{K,b}(.)$ or $P0_K(.)$ are restricted to the execution of the instance of the permutation to test (this is, are restricted to the execution of the algorithm $P_K(m)$). Moreover, it is assumed that the adversary makes at least one request to the oracle $RARP_{K,b}(.)$.

The advantage of the adversary A_{cma} may be defined via:

$$\mathbf{Adv}_{PRP,\,A_{cma}}^{\text{prpi-cma-sca1}}(k)=2*Pr[\mathbf{Expt}_{PRP,\,A_{cma}}^{\text{prpi-cma-sca1}}(k)=1]-1.$$

The advantage function of a permutation, in what regards the experiment prpi-cma-sca1, may be defined as follows:

$$\mathbf{Adv}_{PRP}^{\text{prpi-cma-sca1}}(k, t, pp, sc, q_{P0}, \mu_{P0}, q_{RARP}, \mu_{RARP}, q_{SCA})=\max_{A_{cma}}\{\mathbf{Adv}_{PRP,\,A_{cma}}^{\text{prpi-cma-sca1}}(k)\};$$

where: the value t represents the maximum running time (in seconds) that the adversary A_{cma} has available to accomplish the attack; the value pp represents the maximum processing power (in cycles per second) that the adversary A_{cma} has available to accomplish the attack; the value sc represents the maximum storage capacity (in bits) that the adversary A_{cma} has available to accomplish the attack; the values q_{SCA}, q_{P0}, and q_{RARP} represent the maximum number of queries that the adversary A_{cma} may make to, respectively, the oracles SCA(...), $P0_K(.)$, and $RARP_{K,b}(.)$; and the values μ_{P0} and μ_{RARP} represent the maximum number of pseudorandom-bits that the adversary A_{cma} may see in response to the requests he makes to, respectively, the oracles $P0_K(.)$ and $RARP_{K,b}(.)$. ❙

This security notion is an adaptation of the security notion designated by "Pseudorandom Function Indistinguishability" introduced in this paper. The adaptation of the security notion "Pseudorandom Function Indistinguishability" into the security notion "Pseudorandom Permutation Indistinguishability" follows the

65 The only aim of the database M_DB is providing a way to represent in pseudo-code the rule that states that the adversary, for the same request, never sees the answers of both the oracles $RARP_{K,b}(.)$ and $P0_K(.)$. For this reason, it is assumed that this database is virtual and, therefore, no restriction is imposed to its length.

adaptation described in [33] for the case in which a security notion that tests the pseudorandomness of functions is adapted into a security notion that tests the pseudorandomness of permutations.

Pseudorandom permutations with a block's length equal to l have an inherent limit on their quality as soon as the adversary sees a total of $2^{\text{ceiling}((l+1)/2)}$ results of their use. This limit is seen when the results of the permutation to test are compared with the results of a function selected uniformly at random from the set of all the functions with both an input's length and an output's length equal to the length of the block of the permutation to test. This result is shown as Proposition A.2.

Proposition A.2 [$\mathbf{Adv}_{PRF,P}^{\text{prfi-cma-sca1}}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA}) \geq 0.316*(q*(q-1)/2^{l})$, for any permutation P with a block's length equal to l] Let $P:Keys(P)\times\{0,1\}^{l}\rightarrow\{0,1\}^{l}$ be any family of permutations; let the family of functions $Rand^{l\rightarrow L}$ be restricted to the family of 2^{k} functions with both an input's length and an output's length equal to l ($Rand_{k}^{l\rightarrow l}$); and let q (with $1\leq q\leq 2^{\text{ceiling}((l+1)/2)}$) be an integer that represents the total number of queries that the adversary may make to the oracle $RARF_{K,b}(.)$. Then, there is an adversary A such that:

$$\mathbf{Adv}_{PRF,P}^{\text{prfi-cma-sca1}}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA}) \geq 0.316*(q*(q-1)/2^{l}).$$

Proof: see Appendix B. **|**

This Proposition is an adaptation of the Proposition described in [33] to show the inherent limit that a pseudorandom permutation has when it is used as a pseudorandom function. Accordingly, with reference to the Proposition A.2, one can see that a "good" pseudorandom permutation (with a block's length equal to l) can be used instead of a "good" pseudorandom function (with both an input's length and an output's length equal to l), in case the value q is such that the quantity $0.316*(q*(q-1)/2^{l})$ is negligible.

Appendix B – Theorems' and Propositions' proofs

Proof of Theorem 1 [EU-SCMA-SCA1>EU-CMA-SCA1]:

(1) [EU-SCMA-SCA1>EU-CMA-SCA1] If the symmetric authentication scheme $\prod=(K, T, V)$ is secure in accordance with the experiment EU-SCMA-SCA1, then $\prod$ is secure in accordance with the experiment EU-CMA-SCA1. But the security of $\prod$ in accordance with the experiment EU-CMA-SCA1 does not imply the security of $\prod$ in accordance with the experiment EU-SCMA-SCA1.

To prove that EU-SCMA-SCA1>EU-CMA-SCA1, one must prove that EU-SCMA-SCA1=>EU-CMA-SCA1 and EU-SCMA-SCA1<≠EU-CMA-SCA1. Accordingly, EU-SCMA-SCA1=>EU-CMA-SCA1 is proved in part (2) of this proof and EU-SCMA-SCA1<≠EU-CMA-SCA1 is proved in part (3) of this proof.

(2) [EU-SCMA-SCA1=>EU-CMA-SCA1] If the symmetric authentication scheme $\prod=(K, T, V)$ is secure in accordance with the experiment EU-SCMA-SCA1, then $\prod$ is secure in accordance with the experiment EU-CMA-SCA1.

Part (2) of the proof follows directly from the definition of the experiments EU-SCMA-SCA1 and EU-CMA-SCA1.

Assume that A1 is an adversary attacking $\prod$ in the EU-CMA-SCA1 sense. A new adversary (adversary A2) that attacks $\prod$ in the EU-SCMA-SCA1 sense is contructed by use of the adversary A1[66].

Let the oracles $T0_K(.)$, $V0_K(.,.)$, and SCA(...) be the oracles to which the adversary A2 has access and let the oracles $T0_K(.)$, $V1_K(.,.)$, and SCA(...) be the oracles to which the adversary A1 has access. As one can see, both the adversaries A1 and A2 can access the oracles $T0_K(.)$ and SCA(...). As so, in order to show that the adversary A2 is capable of using his oracles to simulate adversary A1's oracles, let the oracle $V1_K(.,.)$ be defined as a version of the oracle $V0_K(.,.)$ that is characterized by not returning the results of the calls made to it (this is, let the oracle $V1_K(.,.)$ be defined by $V1_K(a1,b1)$\{ answer:=$V0_K(a1,b1)$; return requestAccomplished;\}) and let the adversary A2 ignore the answers of the oracle $V0_K(.,.)$. In this way, it is clear that the adversary A2 is capable of simulating the oracles of the adversary A1. Therefore, it is clear that A2's advantage is given by:

$$\mathbf{Adv}_{SA,\,A1}^{\text{eu-cma-sca1}}(k)=\mathbf{Adv}_{SA,\,A2}^{\text{eu-scma-sca1}}(k).$$

Since A1 is an arbitrary adversary,

$$\mathbf{Adv}_{SA}^{\text{eu-cma-sca1}}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V1}, q_{SCA})\leq\mathbf{Adv}_{SA}^{\text{eu-scma-sca1}}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V0}, q_{SCA}).$$

As a result, the security of $\prod$ in accordance with the experiment EU-SCMA-SCA1 implies the security of $\prod$ in accordance with the experiment EU-CMA-SCA1, as claimed.

(3) [EU-SCMA-SCA1<≠EU-CMA-SCA1] The security of $\prod$ in accordance with the experiment EU-CMA-SCA1 does not imply the security of $\prod$ in accordance with the experiment EU-SCMA-SCA1.

In Part (3) of the proof it is shown that there is a scheme that is secure in the sense of EU-CMA-SCA1 but not in the sense of EU-SCMA-SCA1.

Assume that $\prod=(K, T, V)$ is a deterministic symmetric authentication scheme that is secure in accordance with the experiment EU-SCMA-SCA1 (for instance, SHA1-HMAC [36]). Moreover, assume that the tagging algorithm of $\prod$ (algorithm $T_K(.)$) represents a "good" pseudorandom function (PRF) in the SCA1 Model[67]. Now,

66 This means that it is possible to use adversary A2's oracles to simulate adversary A1's oracles.

67 Informally, being a "good" pseudorandom function (PRF) in the SCA1 Model means that an adversary, even if he is allowed to undertake SCAs against the oracle that executes the instance of the function to test, is not capable of discerning if the output of this oracle is the result of executing the function to test or the result of executing a function selected uniformly at random from the set of all the functions with both the same domain and the same codomain of the function to test. For a formal definition of the security notion that tests the

consider that a new scheme (scheme $\prod$'=(K, T', V')) that is secure in the sense of EU-CMA-SCA1 but not in the sense of EU-SCMA-SCA1 is constructed by use of the tagging algorithm of $\prod$.

Accordingly, let the tagging algorithm (algorithm T_K'(.)) and the verification algorithm (algorithm V_K'(.,.)) associated with $\prod$' be defined, respectively, as follows:

T_K'(M)	V_K'(M, τ)
{	{
if (|M|=0) then return $\perp$;	if (|M|=0) then return $\perp$;
K1:=B_K((10101010)^{16});	// the value tagLength is defined by $\prod$
K2:=B_K((01010101)^{16});	if (|τ|$\neq$tagLength+128+l) then return $\perp$; // with l=32
τ1:=T_{K1}(M);	K1:=B_K((10101010)^{16});
r:=getRandomBits(128);	K2:=B_K((01010101)^{16});
pad:=r||τ1;	r:=τ[0...127];
original_τ2:=T_{K2}(pad);	τ2:=τ[128...128+l-1];
τ2:=original_τ2[0...l-1]; // with l=32	τ1:=τ[128+l...\|τ|-1];
τ:=r||τ2\|\|τ1;	τ1_:=T_{K1}(M);
return τ;	pad:=r||τ1_;
}	original_τ2:=T_{K2}(pad);
	τ2_:=original_τ2[0...l-1];
	τ_:=r||τ2_\|\|τ1_;
	if (τ=τ_) then return 1;
	return 0;
	}

In the above pseudo-code representation, the function T_{Ki}(.) (with i$\in${1,2}) represents the tagging algorithm of $\prod$. Moreover, the function B_K(.) represents a "good" pseudorandom permutation (PRP) in the SCA1 Model[68] (for instance, AES [37]).

In this proof, it is assumed that forging messages with a probability of less than 2^{-20} is negligible. Furthemore, it is assumed that the length of the tag τ2, given by the value l, is 32 bits.

Now, to prove that the security of $\prod$' in accordance with the experiment EU-CMA-SCA1 does not imply the security of $\prod$' in accordance with the experiment EU-SCMA-SCA1, it is shown that $\prod$' is not secure in accordance with the experiment EU-SCMA-SCA1 (see part 3.1 of the proof) but it is secure in accordance with the experiment EU-CMA-SCA1 (see part 3.2 of the proof).

(3.1) [$\prod$' is not secure in accordance with the experiment EU-SCMA-SCA1]

$\prod$' is clearly not secure in accordance with the experiment EU-SCMA-SCA1.

pseudorandomness of a function ("Pseudorandom Function Indistinguishability"), see Appendix A.

68 Informally, being a "good" pseudorandom permutation (PRP) in the SCA1 Model means that an adversary, even if he is allowed to undertake SCAs against the oracle that executes the instance of the permutation to test, is not capable of discerning if the output of this oracle is the result of executing the permutation to test or the result of executing a permutation selected uniformly at random from the set of all the permutations with the same block's length of the permutation to test. For a formal definition of the security notion that tests the pseudorandomness of a permutation ("Pseudorandom Permutation Indistinguishability"), see Appendix A.

Let the oracles $T0_K(.)$, $V0_K(.,.)$, and SCA(...) be the oracles to which the adversary defined in the experiment EU-SCMA-SCA1 (adversary A_{scma}) has access. Now, consider that the adversary A_{scma} to forge a new tag associated with $\prod'$ proceeds as follows:

```
Experiment Expt_{SA, Ascma}^{eu-scma-sca1}(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack: (M, τ)←A_scma^{T0_K(.), V0_K(.,.), SCA(...)}(k, create)
    A_scma^{T0_K(.), V0_K(.,.), SCA(...)}(k, create)
    {
        l:=32;
        M:=getRandomBits(128);
        τ:=T0_K(M);
        r:=τ[0...127];
        τ2:=τ[128...128+l-1];
        τ1:=τ[128+l...|τ|-1];
        r':=xor(r, 10^127);
        counter1:=-1;
        continue1:=true;
        while(continue1=true)
        {
            counter1++;
            τ2':=<<counter1>>_l;
            τ':=r'||τ2'||τ1;
            if (counter<2^32-1) then
            {
                answer:=V0_K(M, τ');
            }
            else
            {
                answer:=1;
            }
            if (answer=1) then continue1:=false;
        }
        τ:=τ';
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
    if ((M, τ) is stored into the database S_DB_K) then return ⊥;
    answer:=V_K(M, τ);
    if (answer=1) then return 1;
    return 0;
} |
```

With reference to the above pseudo-code representation, one can see that if the adversary A_{scma} is allowed to make one request to the oracle $T0_K(.)$ and (2^l-1) requests to the oracle $V0_K(.)$, then he can forge one tag associated with $\prod'$. Hence, $\prod'$ is not secure in the sense of EU-SCMA-SCA1, as claimed.

(3.2) [$\prod$' is secure in accordance with the experiment EU-CMA-SCA1]

With reference to the tagging and the verification algorithms associated with $\prod$', one can see that the adversary defined in the experiment EU-CMA-SCA1 (adversary A_{cma}) to forge a tag τ has to forge at least one of the two partial tags (values $\tau 1$ or $\tau 2$) that compose it. As so, to prove that $\prod$' is secure in the sense of EU-CMA-SCA1, one has to prove that the adversary A_{cma} is not capable of forging any of the two partial tags $\tau 1$ or $\tau 2$ with a probability that is not negligible.

The creation of the partial tag $\tau 1$ represents the direct use of the tagging algorithm of $\prod$. Therefore, as $\prod$ is secure in the sense of EU-SCMA-SCA1, one is assured that the adversary A_{cma} is not capable of forging $\tau 1$.

The creation of the partial tag $\tau 2$, if one considers that the tag $\tau 2$ is not restricted to the first l bits that result from the use of the tagging algorithm of $\prod$, represents the direct use of the tagging algorithm of $\prod$. Therefore, in this case, as $\prod$ is secure in the sense of EU-SCMA-SCA1, one is assured that the adversary A_{cma} is not capable of forging the tag $\tau 2$. However, as it is assumed that the tag $\tau 2$ represents the most important l (with l=32) bits of the result of using the tagging algorithm of $\prod$, the security of $\prod$ in accordance with the experiment EU-SCMA-SCA1 is not helpful to prove that the adversary A_{cma} cannot forge the tag $\tau 2$ with a non-negligible probability.

As so, instead of proving that the tag $\tau 2$ is unforgeable (by the adversary A_{cma}) because $\prod$ is secure in the sense of EU-SCMA-SCA1, one has to prove that the tag $\tau 2$ is unforgeable (by the adversary A_{cma}) due to the way it is defined in $\prod$'.

Accordingly, if one proves that the adversary A_{cma} cannot guess any of the l bits of the tag $\tau 2$ with a non-negligible increase of probability, regarding the probability of guessing any of these bits at random, then one is assured that the adversary A_{cma} cannot forge the tag $\tau 2$ with a probability better than 2^{-l}. Hence, in this case, as $2^{-l}<2^{-20}$ (with l=32), one is assured that the adversary A_{cma} cannot forge the tag $\tau 2$ with a probability that is not negligible.

Accordingly, to prove this, one has to prove that the adversary A_{cma} cannot take profit of interacting with any of the oracles provided to him (tagging and verification oracles) to increase the probability he has of guessing at random any of the bits of the tag $\tau 2$.

Taking into account that the tagging algorithm of $\prod$ is a "good" pseudorandom function in the SCA1 Model, one is assured that 1) the execution of the tagging algorithm of $\prod$ does not leak any information through SCAs and 2) the adversary A_{cma} is not capable of guessing any of the bits that result from the execution of the tagging algorithm of $\prod$ with an increase of probability that is not negligible, regarding the probability of guessing these bits at random.

In this way, one has just to require that the tagging and the verification oracles provided to the adversary A_{cma} do not help him into guessing any of the bits of the tag $\tau 2$. With reference to the representation of both the tagging and the verification algorithms associated with $\prod$', one can see that, in the experiment EU-CMA-SCA1, (as the verification oracle provided to the adversary A_{cma} is mute) the latter requirement is verified.

Above, the forgeability of the partial tags $\tau 1$ and $\tau 2$ is studied independently of each other. This is so because the creation of the partial tags $\tau 1$ and $\tau 2$ is independent of each other, except in what regards the creation of the keys K1 and K2.

In addition, as it is considered that the function $B_K(.)$ is a "good" pseudorandom permutation in the SCA1 Model, one is assured that the adversary does not take profit from undertaking SCAs against this function to

guess bits of any of the keys K1 or K2. Thus, the adversary A_{cma} does not take profit of undertaking SCAs against the function $B_K(.)$ to forge any of the partial tags $\tau 1$ or $\tau 2$.

In conclusion, as the adversary A_{cma} is not capable of forging any of the partial tags $\tau 1$ or $\tau 2$ with a probability that is not negligible, it is clear that $\prod'$ is secure in the sense of EU-CMA-SCA1, as claimed. **|**

Proof of Proposition A.1 [Any function with a sufficiently large minimum-output-length that is secure in the sense of PRFI-CMA-SCA1 may be transformed into a symmetric authentication scheme that is secure in the sense of EU-SCMA-SCA1]:

(1) Any function with a sufficiently large minimum-output-length that is secure in the sense of PRFI-CMA-SCA1 may be transformed into a symmetric authentication scheme that is secure in the sense of EU-SCMA-SCA1.

Taking into account that EU-SCMA-SCA1>EU-CMA-SCA1, one is assured that if a function with a sufficiently large minimum-output-length that is secure in the sense of PRFI-CMA-SCA1 may be transformed into a symmetric authentication scheme that is secure in the sense of EU-SCMA-SCA1 (see part (3) of the proof), then this function may also be transformed into a symmetric authentication scheme that is secure in the sense of EU-CMA-SCA1 (see part (2) of the proof).

Part (3) of the proof makes use of part (2) of the proof.

(2) Any function with a sufficiently large minimum-output-length that is secure in the sense of PRFI-CMA-SCA1 may be transformed into a symmetric authentication scheme that is secure in the sense of EU-CMA-SCA1.

Part (2) of the proof follows directly from the definition of the experiments EU-CMA-SCA1 and PRFI-CMA-SCA1.

Assume that A1 is an adversary attacking $\prod$ in the EU-CMA-SCA1 sense. A new adversary (adversary A2) that attacks $F_K(.)$ in the PRFI-CMA-SCA1 sense is contructed by use of the adversary A1.

Let the oracles $F0_K(.)$, $RARF_{K,b}(.)$, and SCA(...) be the oracles to which the adversary A2 has access and let the oracles $T0_K(.)$, $V1_K(.,.)$, and SCA(...) be the oracles to which the adversary A1 has access. As one can see, both the adversaries A1 and A2 can access the oracle SCA(...). As so, in order to show that the adversary A2 is capable of using his oracles to simulate adversary A1's oracles, let the oracle $T0_K(.)$ be defined as the oracle $F0_K(.)$ (this is, let the oracle $T0_K(.)$ be defined by $T0_K(a1)$ {return $F0_K(a1)$;}[69]) and let the oracle $V1_K(.,.)$ be defined as a version of the oracle $F0_K(.)$ that is characterized by not returning the results of the requests that are made to it (this is, let the oracle $V1_K(.,.)$ be defined by $V1_K(a1, b1)$ {answer:=$F0_K(a1)$; return requestAccomplished;}).

69 Notice that, in this case, it is not necessary to define that the oracle $T0_K(.)$ uses the database S_DB because the verification oracle defined in the experiment EU-CMA-SCA1 is a mute verification oracle.

Accordingly, it is clear that the adversary A2 is capable of simulating the oracles of the adversary A1. As so, if the adversary A1 (by use of the oracles $T0_K(.)$, $V1_K(.,.)$, and SCA(...)) is capable of creating a new pair of values, tag and message, for which the integrity is confirmed, then the adversary A2 (by use of the oracles $F0_K(.)$ and SCA(...)) is capable of creating a new and valid pair of values, input and output, associated with the function to test.

The aim of the adversary A1 is just to create a new pair of values, message and tag, for which the integrity is confirmed. The aim of the adversary A2, however, is not to create a new and valid pair of values, input and output, associated with the function to test, but it is to distinguish whether or not the first string returned (at each request) by the oracle $RARF_{K,b}(.)$ is created by use of the function to test.

In this way, even if it is proved above that the adversary A2 is capable of simulating adversary A1's oracles, in order to prove the claim made in part (2) of the proof, one has to prove that the adversary A2 takes profit of simulating A1's oracles to win his game.

Indeed, to prove that the adversary A2 takes profit of simulating A1's oracles to win his game, one has just to see that the adversary A2 is capable of doing it by undertaking the following steps:

1. by use of the oracles $F0_K(.)$ and SCA(...), the adversary A2 creates a new and valid pair of values, input (value guessedInput) and output (value guessedOutput), associated with the function to test;
2. the adversary A2 makes a request to the oracle $RARF_{K,b}(.)$, for which the input is the value guessedInput;
3. the adversary A2 analyses the two strings returned by the oracle $RARF_{K,b}(.)$ in the step 2. and concludes whether or not the first string returned by the latter oracle is equal to the value guessedOutput.

With reference to what has been said above and considering that the only request that the adversary A2 (during the attack) makes to the oracle $RARF_{K,b}(.)$ is the one described above, it is clear that A2's advantage is given by:

$$\mathbf{Adv}_{SA,\,\Pi,\,A1}^{\text{eu-cma-sca1}}(k)=\mathbf{Adv}_{PRF,\,FK,\,A2}^{\text{prfi-cma-sca1}}(k).$$

Since A1 is an arbitrary adversary, then, for $F_K(.)$ a pseudorandom function and for Π a symmetric authentication scheme defined as represented above,

$$\mathbf{Adv}_{SA,\,\Pi}^{\text{eu-cma-sca1}}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V1}, q_{SCA})\leq\mathbf{Adv}_{PRF,\,Fk}^{\text{prfi-cma-sca1}}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA}).$$

As a result, it is proved that the security of $F_K(.)$ in accordance with the experiment PRFI-CMA-SCA1 implies the security of Π in accordance with the experiment EU-CMA-SCA1, as claimed.

(3) Any function with a sufficiently large minimum-output-length that is secure in the sense of PRFI-CMA-SCA1 may be transformed into a symmetric authentication scheme that is secure in the sense of EU-SCMA-SCA1.

Part (3) of the proof follows from both part (2) of the proof and the definition of the experiments EU-CMA-SCA1, EU-SCMA-SCA1, and PRFI-CMA-SCA1.

Let the oracles $F0_K(.)$, $RARF_{K,b}(.)$, and SCA(...) be the oracles to which the adversary defined in the experiment PRFI-CMA-SCA1 has access, let the oracles $T0_K(.)$, $V1_K(.,.)$, and SCA(...) be the oracles to which the adversary defined in the experiment EU-CMA-SCA1 has access, and let the oracles $T0_K(.)$, $V0_K(.,.)$, and SCA(...) be the oracles to which the adversary defined in the experiment EU-SCMA-SCA1 has access.

In part (2) of the proof, it is shown that it is possible to create an adversary (adversary A2) that attacks $F_K(.)$ in the PRFI-CMA-SCA1 sense by use of an adversary (adversary A1) that attacks $\prod$ in the EU-CMA-SCA1 sense.

In part (3) of the proof, however, it is not possible to show that an adversary (adversary A4) that attacks $F_K(.)$ in the PRFI-CMA-SCA1 sense may be created from an adversary (adversary A3) that attacks $\prod$ in the EU-SCMA-SCA1 sense. This is so because in the experiment PRFI-CMA-SCA1, contrary to what happens in the experiment EU-SCMA-SCA1, there is not an oracle that may be used to test the integrity of pairs of values, input and output, associated with the function to test.

In this way, to prove the claim that is made in this part of the proof, one, first, must find out what is the increase of probability that the adversary defined in the experiment EU-SCMA-SCA1 has in creating a new pair of values, tag and message, for which the integrity is confirmed, regarding the probability that the adversary defined in the experiment EU-CMA-SCA1 accomplishes the same task; and, finally, must add the quantity that represents this increase of probability to both sides of the last inequation introduced in part (2) of the proof.

Accordingly, as the only "resource" that is different between the adversaries defined in the experiments EU-SCMA-SCA1 (adversary A_{scma}) and EU-CMA-SCA1 (adversary A_{cma}) is the possibility that the adversary A_{scma} has of acessing the oracle $V0_K(.,.)$, instead of accessing the oracle $V1_K(.,.)$, one has to find out what is the profit that the adversary A_{scma} takes from this.

As it is assumed that the function $F_K(.)$ is a "good" pseudorandom function in the SCA1 Model, one is assured that 1) this function does not leak any information through SCAs and 2) the adversary can not guess any of the bits that result from executing the function $F_K(.)$ with a non-negligible increase of probability, regarding the probability of guessing any of these bits at random. Hence, the only profit that the adversary A_{scma} takes from making requests to the oracle $V0_K(.,.)$, instead of making requests to the oracle $V1_K(.,.)$, is analysing the answers for the requests he makes to the former oracle. Therefore, the best attack that the adversary A_{scma} can undertake to accomplish his intent is a brute-force attack to the oracle $V0_K(.,.)$.

The aim of performing a brute-force attack to the oracle $V0_K(.,.)$ is either confirming a tag for a message chosen by the adversary A_{scma} or confirming a message for a tag chosen by the adversary A_{scma}. Thus, in such an attack, the adversary A_{scma} aims trying as many pairs of values, tag and message, as possible. Hence, in order that the adversary A_{scma}'s attack is successful with only a negligible probability, it is necessary that the total number of pairs of values, tag and message, he can try is much smaller than the total number of possible tags.

Accordingly, if it is assumed that a probability of less than 2^{-20} is negligible and the adversary A_{cma} is only allowed to make 2^q requests to the oracle $V0_K(.,.)$, then, in order that the brute-force attack of the adversary A_{scma} succedds with only a negligible probability, the total number of possible tags must be given by 2^{min_vL}, where the value min_vL is such that $2^{q-min\_vL} < 2^{-20}$.

In this way, the adversary A_{scma} has an increase of probability in creating a new and valid pair of values, message and tag, regarding the probability that the adversary A_{cma} accomplishes the same task, that is given by 2^{q-min_vL}, with $q_{V0}=2^{q}$.

As so,

$$\mathbf{Adv}_{SA,\Pi}^{eu\text{-}cma\text{-}sca1}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V1}, q_{SCA})=\mathbf{Adv}_{SA,\Pi}^{eu\text{-}scma\text{-}sca1}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V0}, q_{SCA})-2^{q-min\_vL}, \text{ with } q_{V0}= 2^{q}$$

or, in another way,

$$\mathbf{Adv}_{SA,\Pi}^{eu\text{-}cma\text{-}sca1}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V1}, q_{SCA})+2^{q-min\_vL}=\mathbf{Adv}_{SA,\Pi}^{eu\text{-}scma\text{-}sca1}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V0}, q_{SCA}), \text{ with } q_{V0}= 2^{q}.$$

Hence, if one adds the quantity 2^{q-min_vL} to both sides of the last inequation introduced in part (2) of the proof, one gets:

$$\mathbf{Adv}_{SA,\Pi}^{eu\text{-}scma\text{-}sca1}(k, t, pp, sc, q_{T0}, \mu_{T0}, q_{V0}, q_{SCA})\leq\mathbf{Adv}_{PRF,FK}^{prfi\text{-}cma\text{-}sca1}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA})+2^{q-min\_vL}, \text{ with } q_{V0}= 2^{q},$$

what ends the proof of the claim that is made in this part of the proof. **|**

Proof of Proposition A.2 [$\mathbf{Adv}_{PRF,P}^{prfi\text{-}cma\text{-}sca1}(k, t, pp, sc, q_{F0}, \mu_{F0}, q_{RARF}, \mu_{RARF}, q_{SCA})\geq 0.316*(q*(q-1)/2^{l})$, for any permutation P with a block's length equal to l]:

As a permutation is a bijection, the adversary defined in the experiment PRFI-CMA-SCA1 (adversary A_{cma}) is assured that for each result of using a permutation there is only one input associated with it; as the function (function ρ_K) used to test the pseudorandomness of the permutation to test is one of the 2^{k} functions selected uniformly at random from the set of all the functions with both an input's length and an output's length equal to l, the results of this function appear to the adversary A_{cma} as if they were selected both uniformly at random and independently from any of the other results of using this function.

Accordingly, to distinguish whether or not the first string returned (at each request) by the oracle $RARF_{K,b}$ (.) represents the result of executing the instance of the permutation to test, the adversary A_{cma} has to find a collision between two strings associated with the function ρ_K.

After finding a collision, the adversary A_{cma} verifies whether or not this collision corresponds to one of the first strings returned (at each request) by the oracle $RARF_{K,b}(.)$ and, as a result of this verification, he is capable of distinguishing whether or not the first string returned (at each request) by the oracle $RARF_{K,b}(.)$ corresponds to the execution of the instance of the permutation to test.

Due to the birthday phenomenon (see [33], Appendix A), the adversary A_{cma} expects to start seeing collisions as soon as he sees the results of $2^{ceiling((l+1)/2)}$ requests to the oracle $RARF_{K,b}(.)$. Thus, as the probability that a collision is found in the experiment of throwing q balls randomly and independently into 2^{l} buckets is given by $0.316*(q*(q-1)/2^{l})$, the lower bound on the advantage of distinguishing the permutation

to test from a function selected uniformly at random from the family of all the functions with both the same input's length and the same output's length of the permutation to test is as claimed.

When describing attacks based on the birthday problem, it is usual not to describe the way in which the storage of the collected values is undertaken (see, for instance, [33] and [7]). However, this is necessary to verify if the attack may be undertaken by a real-world adversary.

In order that the probability of succeding in a birthday attack is near ½, one has to compare two sets of values (sets B1 and B2) for which the result of the multiplication of the number of their elements exceeds $2^{\wedge l}$ [70], where l is the length of the value to find by use of the birthday attack. Usually, at least one of those two sets of values is stored in (non-volatile) memory. For this reason, it is necessary that the adversary A_{cma} has the enough storage capacity available to accomplish this task.

Accordingly, if one assumes that the set of elements to be stored contains $2^{\wedge ceiling(l/2)}$ elements, then the adversary A_{cma} has to have the enough storage capacity available to store all those elements. Moreover, he has to have the enough processing power capacity to both manage the storage of those elements and the process that enables him to verify if all those elements are different from each other. To accomplish this, it is thus necessary that the $2^{\wedge ceiling(l/2)}$ elements to store in memory are stored in such a way that the resources of the adversary are not wasted.

To maximize the adversary's resources, it is proposed here that one creates a chained hash-table (for more on chained hash-tables, see Section 6.4 of [6]) by use of two arrays. The first array (which aim is serving as an hash-table for the second array) should have as much slots as possible[71] and the second array (which aim is growing the linked lists indexed by the values in the first array) must have $2^{\wedge ceiling((l+1)/2)}$ slots[72].

The first array aims serving as an hash-table for the linked lists created in the second array. Thus, to make the search of the values in the second array efficient, each value in the first array is indexed by an hash-key (value hashKey) that depends on the values to store in the second array. The way in which the value hashKey is created must give origin to as less collisions as possible. This means that this value should be uniformly distributed over all the values of length |hashKey|=log2("number of slots in the first array") bits[73].

One way of creating a value hashKey that is presumably well distributed over all the values of length log2 ("number of slots in the first array") bits is to use a collision-resistant hash function that has the value to store as input and has as output a value of length log2("number of slots in the first array") bits. Nonetheless, if the information to store is pseudo-random, as is the case, then, instead of using an additional function to create the value hashKey, one may use as hash-key the first log2("number of slots in the first array") bits of the information to store[74]. This has as an additional advantage that only the remaining l-log2("number of slots in the first array") bits of the values to store have to be stored in the linked lists grown in the second array.

As the values stored in the first array aim pointing to the linked lists created in the second array, their length is log2("number of slots in the second array") bits.

---

70 Actually, the number of elements in each one of the sets B1 and B2 must be such that $0.316*(q1*(q2-1)/2^{\wedge l})>1/2$, where the value q1 represents the number of elements inside the set B1, the value q2 represents the number of elements inside the set B2, and the value l represents the length (in bits) of the values to compare.

71 For the adversary to use the chained hash-table, it is assumed that he does not have the enough storage capacity to create an array of $2^{\wedge l}$ slots of one bit each. This will be explained below.

72 It is being assumed that, in this case, the adversary stores both the sets B1 and B2 in the second array.

73 It is being assumed that the number of slots in the first array is given by $2^{\wedge a1}$, where the value a1 is an integer greater than, or equal to, ceiling((l+1)/2).

74 It is assumed that the length of the values to store, given by the value l, is greater than log2("number of slots in the first array bits") bits.

The second array aims serving as the base for the creation of the linked lists that are indexed by the values hashKey associated with the first array. As so, the length of the value in each node of those linked lists must be such that it can store the remaining l-log2("number of slots in the first array") bits of the values to store and a value of length log2("number of slots in the second array") bits, which aim is pointing to the next node in the linked list.

For a very big first array, there is a very high probability that the number of elements in any of the linked lists in the second array is always small. As so, the sequential search of the values inside any of those linked lists ends up as always being very fast. However, to improve this search, one may store these values in a sequential order. Thus, here, these values will be stored in ascending order.

Above, the storage of the two sets of values B1 and B2 is described for the case in which the adversary does not have the enough storage capacity to create an array of $2^{\wedge l}$ slots of one bit. However, if the adversary has the enough storage capacity to create such an array, then this array must be created instead of the chained hash-table described above. In this case, the bit stored in each one of the $2^{\wedge l}$ slots of this new array serves as a boolean flag, which aim is informing whether or not the value that indexes the respective slot has already been returned by the oracle $RARF_{K,b}(.)$.

Below, the above described attack is represented. In this attack, it is assumed that the adversary has (1) at least $2^{\wedge ceiling((l+1)/2)}*(2*ceiling((l+1)/2)+(l-ceiling((l+1)/2)))$ bits of storage capacity available and (2) a processing power sufficient to accomplish the attack in a short period of time (for instance, a period of some days).

```
Experiment Expt_{PRF, Acma}^{prfi-cma-sca1}(k)
{
    K←R—K(k);
    create database M_DB;
    b:=getRandomBits(1);
    // begin of adversary's attack: b'←A_{cma}^{RARF_{K,b}(.), F0K(.), SCA(...)}(k, guess)
    A_{cma}^{RARF_{K,b}(.), F0K(.), SCA(...)}(k, guess)
    {
        // l="length of the block of the permutation to test"
        // nArray1Slots="number of slots of the array array1"
        // array1ValuesBits="length of the slots of the array array1"
        // array1PointerBits="number of bits needed to index any of the slots of the array array1"
        // nArray2Slots="number of slots of the array array2"
        // array2ValuesBits="length of the nodes of the linked lists that are grown in the array array2"
        // array2PointerBits="number of bits needed to index any of the slots of the array array2"
        b':=1;
        nValues:=2^ceiling((l+1)/2);
        if (sc>2^l) then
        {
            nArray1Slots:=2^l;
            array1ValuesBits:=1;
            remainingBits:=0;
            maximumValuesToStore:=nValues;
        }
        else
        {
            nArray2Slots:=nValues;
            array2PointerBits:=log2(nArray2Slots);
            nArray1Slots:=2^floor(log2(sc))-1-ceiling(log2(array2PointerBits));
```

```
    array1PointerBits:=log2(nArray1Slots);
    remainingBits:=l-array1PointerBits;
    array2ValuesBits:=remainingBits+array2PointerBits;
    array2:=Array(nArray2Slots, array2ValuesBits);
    array1ValuesBits:=array2PointerBits;
    maximumValuesToStore:=nValues-1; // this is so because the first slot of the array array2, per convention, is set to zero
}
array1:=Array(nArray1Slots, array1ValuesBits);
counter1:=-1;
while(counter1<maximumValuesToStore-1)
{
    counter1++;
    (a1, b1):=RARF_{K,b}(<<counter1>>_l);
    if (remainingBits=0) then
    {
        if (array1[a1]=<<0>>_1) then
        {
            array1[a1]:=<<1>>_1;
        }
        else
        {
            b':=0;
            return b';
        }
    }
    else
    {
        if (array1[a1[0...array1PointerBits-1]]=<<0>>_array2PointerBits) then
        {
            array1[a1[0...array1PointerBits-1]]:=<<counter1+1>>_array2PointerBits;
            array2[counter1+1]:=a1[l-array1PointerBits...l-1]||<<0>>_array2PointerBits;
        }
        else
        {
            oldPointer:=<<0>>_array2PointerBits;
            newPointer:=array1[a1[0...array1PointerBits-1]];
            continue0:=true;
            while(continue0=true)
            {
                if ((array2[newPointer])[0...remainingBits-1]>a1[l-array1PointerBits...l-1]) then
                {
                    array2[counter1+1]:=a1[l-array1PointerBits...l-1]||<<newPointer>>_array2PointerBits;
                    if (oldPointer=<<0>>_array2PointerBits) then
                    {
                        array1[a1[0...array1PointerBits-1]]:=<<counter1+1>>_array2PointerBits;
                    }
                    else
                    {
                        array2[oldPointer]:=(array2[oldPointer])[0...array2ValuesBits-array2PointerBits-1]||
                                                                                    <<counter1+1>>_array2PointerBits;
                    }
                    continue0:=false;
                }
                else
```

```
                    {
                        if ((array2[newPointer])[0...remainingBits-1]=a1[l-array1PointerBits...l-1]) then
                        {
                            b':=0;
                            return b';
                        }
                        else
                        {
                            if ((array2[newPointer])[array2ValuesBits-array2PointerBits...array2ValuesBits-1]=<<0>>array2PointerBits)
                                                                                                                    then
                            {
                                array2[counter1+1]:=a1[l-array1PointerBits...l-1]||<<0>>array2PointerBits;
                                array2[newPointer]:=(array2[newPointer])[0...array2ValuesBits-array2PointerBits-1]||
                                                                                    <<counter1+1>>array2PointerBits;
                                continue0:=false;
                            }
                            else
                            {
                                oldPointer:=newPointer;
                                newPointer:=(array2[oldPointer])[array2ValuesBits-array2PointerBits...array2ValuesBits-1];
                            }
                        }
                    }
                }
            }
        }
    }
    return b';
}
// end of adversary's attack
if (b=b') then return 1;
return 0;
}
```

With reference to what has been said above, one can see that the value l for which the adversary can win the above described game depends on his storage capacity. Moreover, it depends on how the adversary stores the values to compare. This is, in the proposed configurations, the value l for which the adversary can win the game depends on whether the values to compare are stored in a chained hash-table or in a table of $2^{\wedge l}$ slots of 1 bit.

Accordingly, by calculating the storage capacity that it is predicted that an adversary can have available at some given moment in time (in case he has a budget of $150M to invest in this resource at that moment in time), one can predict the value l for which at that moment in time the adversary will be capable of winning this game. Actually, this is what is shown in Table 3 for the years 2004, 2009, 2014, 2019, 2024, 2029, and 2034.

Years	**Chained Hash-Table**	**Table of $2^{\wedge l}$ slots of 1 bit**
2004	l=107	l=62
2009	l=117	l=67
2014	l=127	l=72
2019	l=137	l=77

Years	Chained Hash-Table	Table of $2^{\wedge l}$ slots of 1 bit
2024	l=147	l=82
2029	l=157	l=87
2034	l=167	l=92

Table 3

With reference to both the Table 3 and the above pseudo-code representation, one is lead to the conclusion that for the same available storage capacity the chained hash-table enables one to undertake the attack for bigger values of l. Nonetheless, for the same value of l, the table of $2^{\wedge l}$ slots of 1 bit enables one to undertake the attack in a shorter period of time.

Appendix C – Pseudo-code representation of the attacks undertaken against the already studied schemes

In this Section, the pseudo-code representation of attacks against eleven symmetric authentication schemes is presented. These schemes are the following: CBC-MAC (see [33]), DMAC (see [8]), EAX-MAC (see [26]), HMAC (see [36]), OMAC (see [44]), PMAC (see [11]), RIPE-MAC (see [1]), RMAC (see [8]), TMAC (see [22]), XECB$-MAC (see [46]), and XMACR (see [34]).

The definition of the schemes described here is almost restricted to a pseudo-code representation. As so, before seeing the definition presented here, one is invited to see the definition presented in the paper associated with each one of the schemes described here.

The pseudo-code representation of the attacks described in this Section does not aim showing the most efficient attack or the attack with the smallest pseudo-code representation. Indeed, the only aim of these representations is showing a possible attack against the schemes studied here.

In all the attacks represented in this Section, it is assumed that the attacker is as powerful as an attacker that, in the year 2034, invests $150M on processing power and $150M on storage capacity. Nonetheless, most of times, the described attacks can be accomplished with success by an attacker that, at the moment of the attack, besides his PC, has invested a short amount of money (for instance, less than $2000) in acquiring equipment (for instance, an oscilloscope[75] and the necessary sensors).

C.1 – CBC-MAC

In this Section, the security of the symmetric authentication scheme designated by CBC-MAC (see [33]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the scheme only accepts messages whose length is equal to 384

75 If one pretends to undertake SCAs to slow microprocessors, as, for instance, the microprocessors that are used by smartcards, then one can buy potent-enough oscilloscopes at, for instance, http://www.picotech.com.

bits (|message|=384 bits); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.1) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
T_K(M)
{
    if (|M|≠384) then return ⊥; // |message|=384
    n_m:=|M|/128;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*128...j*128-1];
    }
    y_0:=<<0>>_128;
    for(i:=1; i≤n_m; i++)
    {
        x1_i:=xor(m_i, y_i-1);
        y_i:=AES_K(x1_i);
    }
    τ:=y_n_m;
    return τ;
}
```

```
V_K(M, τ)
{
    if (|M|≠384) then return ⊥; // |message|=384
    if (|τ|≠128) then return ⊥; // |tag|=128
    n_m:=|M|/128;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*128...j*128-1];
    }
    y_0:=<<0>>_128;
    for(i:=1; i≤n_m; i++)
    {
        x1_i:=xor(m_i, y_i-1);
        y_i:=AES_K(x1_i);
     }
     τ_:=y_n_m;
     if (τ_=τ) then return 1;
     return 0;
}
```

Fig. C.1

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt^{eu-cma-sca1}_{SA, Acma}(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack : (M, τ)←A_cma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    A_cma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    {
        dscaIndex:=0;
        τ:=getRandomBits(128); // |tag|=128
        M:=getRandomBits(384); // |message|=3*128
        n_m:=|M|/128;
        for(j:=1; j≤n_m; j++)
        {
            m_j:=M[(j-1)*128...j*128-1];
        }
        temp_m2_1:=m_2;
        for(i:=1; i≤L1; i++)
        {
```

```
            Mi:=m1||temp_m2i||m3;
            SCA(acquiring_request, i)
            {
                V1K(Mi, τ);
            }
            temp_m2i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(mi, yi-1), with i=2, tValue=y1, and kValue=m2", temp_m2i);
        }
        dscaIndex++;
        y1:=temp_m2L1+1;
        temp_m31:=m3;
        for(i:=1; i≤L1; i++)
        {
            Mi:=m1||m2||temp_m3i;
            SCA(acquiring_request, i)
            {
                V1K(Mi, τ);
            }
            temp_m3i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(mi, yi-1), with i=3, tValue=y2, and kValue=m3", temp_m3i);
        }
        dscaIndex++;
        y2:=temp_m3L1+1;
        current_m2:=xor(m3, y2); // x13:=xor(m3, y2)
        current_m2:=xor(current_m2, y1);
        for(i:=1; i≤L1; i++)
        {
            Mi:=m1||current_m2||temp_m3i;
            SCA(acquiring_request, i)
            {
                V1K(Mi, τ);
            }
            temp_m3i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(mi, yi-1), with i=3, tValue=y2, and kValue=m3", temp_m3i);
        }
        dscaIndex++;
        current_y2:=temp_m3L1+1;
        future_y3:=current_y2;
        τ:=future_y3;
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DBK
    if ((M, τ) is stored into the database S_DBK) then return ⊥;
    answer:=VK(M, τ);
    if (answer=1) then return 1;
    return 0;
} |
```

C.2 – DMAC

In this Section, the security of the symmetric authentication scheme designated by DMAC (see [8]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the scheme only accepts messages whose length is a non-zero

multiple of the length of the block of the block-cipher used (|message|=a1*128 bits, where a1 is an integer greater than zero); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.2) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

$T_K(M)$	$V_K(M, \tau)$
{	{
if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return ⊥;	if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return ⊥;
K1:=K[0...127];	if (|τ|≠128) then return ⊥; // |tag|=128
K2:=K[128...255];	K1:=K[0...127];
n_m:=|M|/128;	K2:=K[128...255];
for(j:=1; j≤n_m; j++)	n_m:=|M|/128;
{	for(j:=1; j≤n_m; j++)
m_j:=M[(j-1)*128...j*128-1];	{
}	m_j:=M[(j-1)*128...j*128-1];
y_0:=<<0>>$_{128}$;	}
for(i:=1; i≤n_m; i++)	y_0:=<<0>>$_{128}$;
{	for(i:=1; i≤n_m; i++)
$x1_i$:=xor(m_i, y_{i-1});	{
y_i:=AES_{K1}($x1_i$);	$x1_i$:=xor(m_i, y_{i-1});
}	y_i:=AES_{K1}($x1_i$);
τ:=AES_{K2}(y_{n_m});	}
return τ;	τ_:=AES_{K2}(y_{n_m});
}	if (τ_=τ) then return 1;
	return 0;
	}

Fig. C.2

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

Experiment **Expt**$^{eu\text{-}cma\text{-}sca1}_{SA,\ A_{cma}}$(k)
{
K←R—*K*(k);
create database S_DB_K;
// begin of adversary's attack : (M, τ)←$A_{cma}^{T0K(.),\ V1K(.,.),\ SCA(...)}$(k, create)
$A_{cma}^{T0K(.),\ V1K(.,.),\ SCA(...)}$(k, create)
{
dscaIndex:=0;
M:=getRandomBits(384); // |message|=3*128
n_m:=|M|/128;
for(j:=1; j≤n_m; j++)
{
m_j:=M[(j-1)*128...j*128-1];
}
τ:=$T0_K$(M);
$temp\_m3_1$:=m_3;

```
        for(i:=1; i≤L1; i++)
        {
            Mi:=m1||m2||temp_m3i;
            SCA(acquiring_request, i,)
            {
                V1K(Mi, τ);
            }
            temp_m3i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(mi, yi-1), with i=3, tValue=y2, and kValue=m3", temp_m3i);
        }
        dscaIndex++;
        y2:=temp_m3L1+1;
        m2':=getRandomBits(128);
        for(i:=1; i≤L1; i++)
        {
            Mi:=m1||m2'||temp_m3i;
            SCA(acquiring_request, i,)
            {
                V1K(Mi, τ);
            }
            temp_m3i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(mi, yi-1), with i=3, tValue=y2, and kValue=m3", temp_m3i);
        }
        dscaIndex++;
        current_y2:=temp_m3L1+1;
        m3':=xor(m3, y2); // x13=xor(m3, y2)
        m3':=xor(m3', current_y2);
        M:=m1||m2'||m3';
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DBK
    if ((M, τ) is stored into the database S_DBK) then return ⊥;
    answer:=VK(M, τ);
    if (answer=1) then return 1;
    return 0;
}
```

C.3 – EAX-MAC

In this Section, the security of the symmetric authentication scheme designated by EAX-MAC (see [26]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the messages to authenticate have a length of at least one bit (|message|>0 bits); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.3) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
T_K(M)
{
    if (|M|=0) then return ⊥;
    r0:=<<0>>_128;
    L:=AES_K(<<0>>_128);
    B:=shift_left(L);
    if (msb(L)≠0) then
    {
      B:=xor(B, 0^120 10000111);
    }
    P:=shift_left(B);
    if (msb(B)≠0) then
    {
      P:=xor(P, 0^120 10000111);
    }
    a1:=<<0>>_128||r0;
    // if ((|a1| mod 128)=0) then
    // {
        a1p1:=a1[0...|a1|-|B|-1];
        a1p2:=a1[|a1|-|B|...|a1|-1];
        w1_temp:=xor(a1p2, B);
        w1:=a1p1||w1_temp;
    // }
    n_w1:=2;
    w1_1:=w1[0...127];
    w1_2:=w1[128...255];
    w1_y_0:=<<0>>_128;
    for(i:=1; i≤n_w1; i++)
    {
        w1_a1_i:=xor(w1_i, w1_y_{i-1});
        w1_y_i:=AES_K(w1_a1_i);
    }
    v1:=w1_y_{n_w1}; // n_w1=2
    a2:=<<1>>_128||M;
    if ((|a2| mod 128)=0) then
    {
        a2p1:=a2[0...|a2|-|B|-1];
        a2p2:=a2[|a2|-|B|...|a2|-1];
        w2_temp:=xor(a2p2, B);
        w2:=a2p1||w2_temp;
    }
    else
    {
        a2':=a2||10^{128-1-(|a2| mod 128)};
        a2'p1:=a2'[0...|a2'|-|P|-1];
        a2'p2:=a2'[|a2'|-|P|...|a2'|-1];
        w2_temp:=xor(a2'p2, P);
        w2:=a2'p1||w2_temp;
    }
    n_w2:=|w2|/128;
    for(j:=1; j≤n_w2; j++)
    {
       w2_j:=w2[(j-1)*128...j*128-1];
    }
```

```
V_K(M, τ)
{
    if (|M|=0) then return ⊥;
    if (|τ|≠128) then return ⊥; // |tag|=128
    r0:=<<0>>_128;
    L:=AES_K(<<0>>_128);
    B:=shift_left(L);
    if (msb(L)≠0) then
    {
      B:=xor(B, 0^120 10000111);
    }
    P:=shift_left(B);
    if (msb(B)≠0) then
    {
      P:=xor(P, 0^120 10000111);
    }
    a1:=<<0>>_128||r0;
    // if ((|a1|/128)=0) then
    // {
        a1p1:=a1[0...|a1|-|B|-1];
        a1p2:=a1[|a1|-|B|...|a1|-1];
        w1_temp:=xor(a1p2, B);
        w1:=a1p1||w1_temp;
    // }
    n_w1:=2;
    w1_1:=w1[0...127];
    w1_2:=w1[128...255];
    w1_y_0:=<<0>>_128;
    for(i:=1; i≤n_w1; i++)
    {
        w1_a1_i:=xor(w1_i, w1_y_{i-1});
        w1_y_i:=AES_K(w1_a1_i);
    }
    v1:=w1_y_{n_w1}; // n_w1=2
    a2:=<<1>>_128||M;
    if ((|a2| mod 128)=0) then
    {
        a2p1:=a2[0...|a2|-|B|-1];
        a2p2:=a2[|a2|-|B|...|a2|-1];
        w2_temp:=xor(a2p2, B);
        w2:=a2p1||w2_temp;
    }
    else
    {
        a2':=a2||10^{128-1-(|a2| mod 128)};
        a2'p1:=a2'[0...|a2'|-|P|-1];
        a2'p2:=a2'[|a2'|-|P|...|a2'|-1];
        w2_temp:=xor(a2'p2, P);
        w2:=a2'p1||w2_temp;
    }
    n_w2:=|w2|/128;
    for(j:=1; j≤n_w2; j++)
    {
       w2_j:=w2[(j-1)*128...j*128-1];
```

```
    w2_y0:=<<0>>128;
    for(i:=1; i≤n_w2; i++)
    {
        w2_a2i:=xor(w2i, w2_yi-1);
        w2_yi:=AESK(w2_a2i);
    }
    v2:=w2_yn_w2;
    a3:=<<2>>128;
    // if ((|a3| mod 128)=0) then
    //{
        a3p1:=a3[0...|a3|-|B|-1]; // a3p1=empty_string
        a3p2:=a3[|a3|-|B|...|a3|-1];
        w3_temp:=xor(a3p2, B);
        w3:=a3p1||w3_temp;
    // }
    n_w3:=1;
    w31:=w3;
    w3_y0:=<<0>>128;
    for(i:=1; i≤n_w3; i++)
    {
        w3_a3i:=xor(w3i, w3_yi-1);
        w3_yi:=AESK(w3_a3i);
    }
    v3:=w3_yn_w3;
    a4:=xor(v1, v2);
    τ:=xor(a4, v3);
    return τ;
}
```

```
    }
    w2_y0:=<0>>128;
    for(i:=1; i≤n_w2; i++)
    {
        w2_a2i:=xor(w2i, w2_yi-1);
        w2_yi:=AESK(w2_a2i);
    }
    v2:=w2_yn_w2;
    a3:=<<2>>128;
    //if ((|a3| mod 128)=0) then
    //{
        a3p1:=a3[0...|a3|-|B|-1]; // a3p1=empty_string
        a3p2:=a3[|a3|-|B|...|a3|-1];
        w3_temp:=xor(a3p2, B);
        w3:=a3p1||w3_temp;
    //}
    n_w3:=1;
    w31:=w3;
    w3_y0:=<<0>>128;
    for(i:=1; i≤n_w3; i++)
    {
        w3_a3i:=xor(w3i, w3_yi-1);
        w3_yi:=AESK(w3_a3i);
    }
    v3:=w3_yn_w3;
    a4:=xor(v1, v2);
    τ_:=xor(a4, v3);
    if (τ_=τ) then return 1;
    return 0;
}
```

Fig. C.3

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt_{SA, Acma}^{eu-cma-sca1}(k)
{
    K←R—K(k);
    create database S_DBK;
    // begin of adversary's attack : (M, τ)←Acma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    Acma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    {
        dscaIndex:=0;
        r0:=<<0>>128;
        τ:=getRandomBits(128); // |tag|=128
        M:=getRandomBits(384); // |message|=3*128
        n_m:=|M|/128;
        for(j:=1; j≤n_m; j++)
        {
```

```
    mj:=M[(j-1)*128...j*128-1];
}
//  w2=a2p1||w2_temp, with a2p1=<<1>>128||m1||m2 and w2_temp=xor(m3, B)
//  w21=<<1>>128, w22=m1, w23=m2, w24=w2_temp;
temp_w22_1:=m1;
for(i:=1; i≤L1; i++)
{
    temp_m1i:=temp_w22_i;
    M_i:=temp_m1i||m2||m3;
    SCA(acquiring_request, i,)
    {
        V1K(M_i, τ);
    }
    temp_w22_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=2, tValue=w2_y1, and kValue=w22",
                                                                                                    temp_w22_i);
}
dscaIndex++;
w2_y1:=temp_w22_L1+1;
current_m1:=w2_y1; // current_w22=current_m1
temp_w23_1:=m2;
for(i:=1; i≤L1; i++)
{
    temp_m2i:=temp_w23_i;
    M_i:=current_m1||temp_m2i||m3;
    SCA(acquiring_request, i,)
    {
        V1K(M_i, τ);
    }
    temp_w23_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=3, tValue=w2_y2, and kValue=w23",
                                                                                                    temp_w23_i);
}
dscaIndex++;
// w2_a22=xor(w22, w2_y1)=<<0>>128, with w22=current_m1, w2_y1=current_m1, w2_y2=AESK(w2_a22)=L
current_w2_y2:=temp_w23_L1+1;
L:=current_w2_y2;
B:=shift_left(L);
if (msb(L)≠0) then
{
    B:=xor(B, 0120 10000111);
}
// w1_a11=<<0>>128, w1_y1=L, w11=<<0>>128, w1_y0=<<0>>128, w12=B, w1_a12=xor(B, L), w1_y2=AESK(w1_a12)
w12:=xor(<<0>>128, B);
w1_a12:=xor(B, L);
current_m1:=xor(w2_y1, w1_a12);
for(i:=1; i≤L1; i++)
{
    temp_m2i:=temp_w23_i;
    M_i:=current_m1||temp_m2i||m3;
    SCA(acquiring_request, i,)
    {
        V1K(M_i, τ);
    }
    temp_w23_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=3, tValue=w2_y2, and kValue=w23",
                                                                                                    temp_w23_i);
```

```
}
dscaIndex++;
current_w2_y2:=temp_w23_L1+1;
w1_y2:=current_w2_y2;
v1:=w1_y2;
for(i:=1; i≤L1; i++)
{
    temp_m2i:=temp_w23_i;
    M_i:=m1||temp_m2i||m3;
    SCA(acquiring_request, i,)
    {
        V1K(M_i, τ);
    }
    temp_w23_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=3, tValue=w2_y2, and kValue=w23",
                                                                                                    temp_w23_i);
}
dscaIndex++;
w2_y2:=temp_w23_L1+1;
temp_w24_1:=xor(m3, B);
for(i:=1; i≤L1; i++)
{
    temp_m3i:=xor(temp_w24_i, B);
    M_i:=m1||m2||temp_m3i;
    SCA(acquiring_request, i,)
    {
        V1K(M_i, τ);
    }
    temp_w24_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=4, tValue=w2_y3, and kValue=w24",
                                                                                                    temp_w24_i);
}
dscaIndex++;
w2_y3:=temp_w24_L1+1;
current_m2:=xor(m3, B); // w24=w2_temp, with w2_temp=xor(a2p2, B) and a2p2=m3
current_m2:=xor(current_m2, w2_y3); // w2_a24=xor(w24, w2_y3)
current_m2:=xor(current_m2, w2_y2);
for(i:=1; i≤L1; i++)
{
    temp_m3i:=xor(temp_w24_i, B);
    M_i:=m1||current_m2||temp_m3i;
    SCA(acquiring_request, i,)
    {
        V1K(M_i, τ);
    }
    temp_w24_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=4, tValue=w2_y3, and kValue=w24",
                                                                                                    temp_w24_i);
}
dscaIndex++;
current_w2_y3:=temp_w24_L1+1;
w2_y4:=current_w2_y3;
v2:=w2_y4;
a3p2:=<<2>>128;
w3_temp:=xor(a3p2, B);
w31:=w3_temp;
w3_a31:=xor(w31, w3_y0); // w3_y0=<<0>>128
```

```
        current_m1:=xor(w2_y1, w3_a31); // w2_a22=w3_a31
        for(i:=1; i≤L1; i++)
        {
            temp_m2i:=temp_w23_i;
            M_i:=m1||temp_m2i||m3;
            SCA(acquiring_request, i,)
            {
                V1K(M_i, τ);
            }
            temp_w23_i+1:=SCA(dsca_request, i, dscaIndex, "w2_a2i:=xor(w2i, w2_yi-1), with i=3, tValue=w2_y2, and kValue=w23",
                                                                                                    temp_w23_i);
        }
        dscaIndex++;
        current_w2_y2:=temp_w23_L1+1;
        v3:=current_w2_y2;
        a4:=xor(v1, v2);
        τ:=xor(a4, v3);
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DBK
    if ((M, τ) is stored into the database S_DBK) then return ⊥;
    answer:=VK(M, τ);
    if (answer=1) then return 1;
    return 0;
} |
```

C.4 – HMAC

In this Section, the security of the symmetric authentication scheme designated by HMAC (see [36]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying hash-compression primitive is SHA-1 (see [35]); the scheme only accepts messages whose length is a non-zero multiple of the length of the block of the compression-function used (|message|=a1*512 bits, where a1 is an integer greater than zero); the scheme only accepts tags whose length is equal to 160 bits (|tag|=160 bits); and the value λ represents the initial hash-key defined in [35].

The scheme (see fig. C.4) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

$T_K(M)$	$V_K(M, \tau)$
{	{
if ((|M|=0) ∨ ((|M| mod 512)≠0)) then return ⊥;	if ((|M|=0) ∨ ((|M| mod 512)≠0)) then return ⊥;
K0:=K||$0^{512-\|K\|}$;	if (|τ|≠160) then return ⊥; // |tag|=160
ipad=$(00110110)^{512/8}$;	K0:=K||$0^{512-\|K\|}$;
opad=$(01011100)^{512/8}$;	ipad=$(00110110)^{512/8}$;
K1:=xor(K0, ipad);	opad=$(01011100)^{512/8}$;
K2:=xor(K0, opad);	K1:=xor(K0, ipad);
n_m:=|M|/512;	K2:=xor(K0, opad);
for(j:=1; j≤n_m; j++)	n_m:=|M|/512;

```
    {
        mj:=M[(j-1)*512...j*512-1];
    }
    hKey0:=λ;
    m0:=K1;
    for(i:=0; i≤n_m; i++)
    {
      hKeyi+1:=SHA1hKeyi (mi);
    }
    x0:=K2;
    x1:=<<hKeyn_m+1>>512;
    for(i:=0; i≤1; i++)
    {
      hKeyi+1:=SHA1hKeyi (xi);
    }
    τ:=hKey2;
    return τ;
}
```

```
    for(j:=1; j≤n_m; j++)
    {
        mj:=M[(j-1)*512...j*512-1];
    }
    hKey0:=λ;
    m0:=K1;
    for(i:=0; i≤n_m; i++)
    {
      hKeyi+1:=SHA1hKeyi (mi);
    }
    x0:=K2;
    x1:=<<hKeyn_m+1>>512;
    for(i:=0; i≤1; i++)
    {
      hKeyi+1:=SHA1hKeyi (xi);
    }
    τ_:=hKey2;
    if (τ_=τ) then return 1;
    return 0;
}
```

Fig. C.4

a) Unforgeability

i) Secure in the sense of EU-SCMA-SCA1

The studied scheme "appears" to be secure in the sense of EU-SCMA-SCA1.

C.5 – OMAC

In this Section, the security of the symmetric authentication scheme designated by OMAC (see [44]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the messages to authenticate have a length of at least one bit (|message|>0 bits); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.5) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
TK(M)
{
    if (|M|=0) then return ⊥;
    L:=AESK(<<0>>128);
    tRemainder:=|M| mod 128;
    if (tRemainder=0) then p_m:=M;
```

```
VK(M, τ)
{
    if (|M|=0) then return ⊥;
    if (|τ|≠128) then return ⊥; // |tag|=128
    L:=AESK(<<0>>128);
    tRemainder:=|M| mod 128;
```

```
    else p_m:=M||10^(128-tRemainder-1);
    n_p_m:=|p_m|/128;
    for(j:=1; j≤n_p_m; j++)
    {
        p_m_j:=p_m[(j-1)*128...j*128-1];
    }
    if (tRemainder=0) then
    {
        B:=shift_left(L);
        if (lsb(L)≠0) then
        {
          B:=xor(B, 0^120 10000111);
        }
    }
    else
    {
        B:=shift_right(L);
        if (msb(L)≠0) then
        {
          B:=xor(B, 10^120 1000011);
        }
    }
    p_m_(n_p_m)':=xor(p_m_(n_p_m), B);
    p_m_(n_p_m):=p_m_(n_p_m)';
    y_0:=<<0>>_128;
    for(i:=1; i≤n_p_m; i++)
    {
        x1_i:=xor(p_m_i, y_(i-1));
        y_i:=AES_K(x1_i);
    }
    τ:=y_(n_p_m);
    return τ;
}
```

```
    if (tRemainder=0) then p_m:=M;
    else p_m:=M||10^(128-tRemainder-1);
    n_p_m:=|p_m|/128;
    for(j:=1; j≤n_p_m; j++)
    {
        p_m_j:=p_m[(j-1)*128...j*128-1];
    }
    if (tRemainder=0) then
    {
        B:=shift_left(L);
        if (lsb(L)≠0) then
        {
           B:=xor(B, 0^120 10000111);
        }
    }
    else
    {
        B:=shift_right(L);
        if (msb(L)≠0) then
        {
          B:=xor(B, 10^120 1000011);
         }
    }
    p_m_(n_p_m)':=xor(p_m_(n_p_m), B);
    p_m_(n_p_m):=p_m_(n_p_m)';
    y_0:=<<0>>_128;
    for(i:=1; i≤n_p_m; i++)
    {
        x1_i:=xor(p_m_i, y_(i-1));
        y_i:=AES_K(x1_i);
    }
    τ_:=y_(n_p_m);
    if (τ_=τ) then return 1;
    return 0;
}
```

Fig. C.5

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt_(SA, Acma)^(eu-cma-sca1)(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack : (M, τ)←A_cma^(T0K(.), V1K(.,.), SCA(...))(k, create)
    A_cma^(T0K(.), V1K(.,.), SCA(...))(k, create)
    {
        dscaIndex:=0;
        τ:=getRandomBits(128); // |tag|=128
```

M:=getRandomBits(384); // |message|=3*128
n_m:=|M|/128;
for(j:=1; j≤n_m; j++)
{
    m_j:=M[(j-1)*128...j*128-1];
}
$p\_m_{2\_1}$:=m_2;
$p\_m_{3\_1}$:=m_3;
for(i:=1; i≤L1; i++)
{
    $temp\_m2_i$:=$p\_m_{2\_i}$;
    $temp\_m3_i$:=$p\_m_{3\_i}$;
    M_i:=m_1||$temp\_m2_i$||$temp\_m3_i$;
    SCA(acquiring_request, i,)
    {
        $V1_K(M_i, \tau)$;
    }
    $p\_m_{2\_i+1}$:=SCA(dsca_request, i, dscaIndex, "$x1_i$:=xor(p_m_i, y_{i-1}), with i=2, tValue=y_1, and kValue=p_m_2", $p\_m_{2\_i}$);
    $p\_m_{3\_i+1}$:=SCA(dsca_request, i, dscaIndex+1, "$p\_m_{n\_p\_m}$':=xor($p\_m_{n\_p\_m}$, B), with n_p_m=3, tValue=B, and kValue=p_m_3", $p\_m_{3\_i}$);
}
dscaIndex:=dscaIndex+2;
y_1:=$p\_m_{2\_L1+1}$;
B:=$p\_m_{3\_L1+1}$;
$t\_p\_m3_1$:=xor(m_3, B);
for(i:=1; i≤L1; i++)
{
    $temp\_m3_i$:=xor($t\_p\_m3_i$, B);
    M_i:=m_1||m_2||$temp\_m3_i$;
    SCA(acquiring_request, i,)
    {
        $V1_K(M_i, \tau)$;
    }
    $t\_p\_m3_{i+1}$:=SCA(dsca_request, i, dscaIndex, "$x1_i$:=xor(p_m_i, y_{i-1}), with i=3, tValue=y_2, and kValue=p_m_3", $t\_p\_m3_i$);
}
dscaIndex++;
y_2:=$p\_m_{3\_L1+1}$;
current_m2:=xor(m_3, B); // p_m_3=xor(m_3, B)
current_m2:=xor(current_m2, y_2); // $x1_3$=xor(p_m_3, y_2)
current_m2:=xor(current_m2, y_1);
for(i:=1; i≤L1; i++)
{
    $temp\_m3_i$:=xor($t\_p\_m3_i$, B);
    M_i:=m_1||current_m2||$temp\_m3_i$;
    SCA(acquiring_request, i,)
    {
        $V1_K(M_i, \tau)$;
    }
    $t\_p\_m3_{i+1}$:=SCA(dsca_request, i, dscaIndex, "$x1_i$:=xor(p_m_i, y_{i-1}), with i=3, tValue=y_2, and kValue:=p_m_3", $t\_p\_m3_i$);
}
dscaIndex++;
current_y2:=$t\_p\_m3_{L1+1}$;
future_y3:=current_y2;
τ:=future_y3;

```
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
    if ((M, τ) is stored into the database S_DB_K) then return ⊥;
    answer:=V_K(M, τ);
    if (answer=1) then return 1;
    return 0;
} |
```

C.6 – PMAC

In this Section, the security of the symmetric authentication scheme designated by PMAC (see [11]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the messages to authenticate have a length of at least one bit (|message|>0 bits); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.6) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
T_K(M)
{
    if ((|M|=0) ∨ (|M|>128*2^128)) then return ⊥;
    L:=AES_K(<<0>>_128);
    tRemainder:=|M| mod 128;
    if (tRemainder=0) then p_m:=M;
    else p_m:=M||10^(128-tRemainder-1);
    n_p_m:=|p_m|/128;
    for(j:=1; j≤n_p_m; j++)
    {
        p_m_j:=p_m[(j-1)*128...j*128-1];
    }
    checksum_0:=<<0>>_128;
    z_1:=L;
    for(i:=2; i≤n_p_m-1; i++)
    {
        ctr1:=1;
        t_x1:=L;
        while (ctr1≤ntz(i))
        {
            tt_x1:=shift_left(t_x1);
            if (msb(t_x1)≠0) then
            {
                tt_x1:=xor(tt_x1, 0^120 10000111);
            }
            ctr1:=(ctr1+1) mod 2^128;
            t_x1:=tt_x1;
        }
        z_i:=xor(z_i-1, t_x1);
```

```
V_K(M, τ)
{
    if ((|M|=0) ∨ (|M|>128*2^128)) then return ⊥;
    if (|τ|≠128) then return ⊥; // |tag|=128
    L:=AES_K(<<0>>_128);
    tRemainder:=|M| mod 128;
    if (tRemainder=0) then p_m:=M;
    else p_m:=M||10^(128-tRemainder-1);
    n_p_m:=|p_m|/128;
    for(j:=1; j≤n_p_m; j++)
    {
        p_m_j:=p_m[(j-1)*128...j*128-1];
    }
    checksum_0:=<<0>>_128;
    z_1:=L;
    for(i:=2; i≤n_p_m-1; i++)
    {
        ctr1:=1;
        t_x1:=L;
        while (ctr1≤ntz(i))
        {
            tt_x1:=shift_left(t_x1);
            if (msb(t_x1)≠0) then
            {
              tt_x1:=xor(tt_x1, 0^120 10000111);
            }
            ctr1:=(ctr1+1) mod 2^128;
            t_x1:=tt_x1;
```

```
    }
    for(i:=1; i≤n_p_m-1; i++)
    {
        x1_i:=xor(p_m_i, z_i);
        y_i:=AES_K(x1_i);
        checksum_i:=xor(checksum_{i-1}, y_i);
    }
    checksum_{n_p_m}:=xor(checksum_{n_p_m-1}, p_m_{n_p_m});
    checksum:=checksum_{n_p_m};
    if (tRemainder=0) then
    {
        x2:=shift_right(L);
        if (lsb(L)≠0) then
        {
           x2:=xor(x2, 10^120 1000011);
        }
        x3:=xor(checksum, x2);
    }
    else x3:=checksum;
    y_{n_p_m}:=AES_K(x3);
    τ:=y_{n_p_m};
    return τ;
}
```

```
        }
        z_i:=xor(z_{i-1}, t_x1);
    }
    for(i:=1; i≤n_p_m-1; i++)
    {
        x1_i:=xor(p_m_i, z_i);
        y_i:=AES_K(x1_i);
        checksum_i:=xor(checksum_{i-1}, y_i);
    }
    checksum_{n_p_m}:=xor(checksum_{n_p_m-1}, p_m_{n_p_m});
    checksum:=checksum_{n_p_m};
    if (tRemainder=0) then
    {
        x2:=shift_right(L);
        if (lsb(L)≠0) then
        {
          x2:=xor(x2, 10^120 1000011);
        }
        x3:=xor(checksum, x2);
    }
    else x3:=checksum;
    y_{n_p_m}:=AES_K(x3);
    τ_:=y_{n_p_m};
    if (τ_=τ) then return 1;
    return 0;
}
```

Fig. C.6

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt_{SA, Acma}^{eu-cma-sca1}(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack : (M, τ)←A_cma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    A_cma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    {
        dscaIndex:=0;
        τ:=getRandomBits(128); // |tag|=128
        M:=getRandomBits(512); // |message|=4*128
        n_m:=|M|/128;
        for(j:=1; j≤n_m; j++)
        {
            m_j:=M[(j-1)*128...j*128-1];
        }
        p_m_{4_1}:=m_4;
        for(i:=1; i≤L1; i++)
        {
```

```
    temp_m4_i:=p_m_{4_i};
    M_i:=m_1||m_2||m_3||temp_m4_i;
    SCA(acquiring_request, i,)
    {
        V1_K(M_i, τ);
    }
    p_m_{4_i+1}:=SCA(dsca_request, i, dscaIndex, "checksum_{n_p_m}:=xor(checksum_{n_p_m-1}, p_m_{n_p_m}), with n_p_m=4,
                                                                    tValue=checksum_3, and kValue=p_m_4", p_m_{4_i});
}
dscaIndex++;
checksum_3:=p_m_{4_L1+1};
checksum_4:=xor(checksum_3, m_4);
checksum:=checksum_4;
p_m_{1_1}:=m_1;
for(i:=1; i≤L1; i++)
{
    temp_m1_i:=p_m_{1_i};
    M_i:=temp_m1_i||m_2||m_3||m_4;
    SCA(acquiring_request, i,)
    {
        V1_K(M_i, τ);
    }
    p_m_{1_i+1}:=SCA(dsca_request, i, dscaIndex, "x1_i:=xor(p_m_i, z_i), with i=1, tValue=z_1, and kValue=p_m_1", p_m_{1_i});
}
dscaIndex++;
z_1:=p_m_{1_L1+1};
L:=z_1;
for(i:=2; i≤3; i++)
{
    ctr1:=1;
    t_x1:=L;
    while (ctr1≤ntz(i))
    {
        tt_x1:=shift_left(t_x1);
        if (msb(t_x1)≠0) then
        {
            tt_x1:=xor(tt_x1, 0^{120}10000111);
        }
        ctr1:=(ctr1+ 1) mod 2^{128};
        t_x1:=tt_x1;
    }
    z_i:=xor(z_{i-1}, t_x1);
}
x2:=shift_right(L);
if (lsb(L)≠0) then
{
    x2:=xor(x2, 10^{120}1000011);
}
x3:=xor(checksum, x2);
for(i:=1; i≤3; i++)
{
    current_m_i:=xor(x3, z_i);
}
for(i:=1; i≤L1; i++)
```

```
        {
            temp_m4_i:=p_m4_i;
            M_i:=current_m_1||current_m_2||current_m_3||temp_m4_i;
            SCA(acquiring_request, i,)
            {
                V1_K(M_i, τ);
            }
            p_m4_i+1:=SCA(dsca_request, i, dscaIndex, "checksum_n_p_m:=xor(checksum_n_p_m-1, p_m_n_p_m), with n_p_m=4,
                                                                    tValue=checksum_3, and kValue=p_m_4", p_m4_i);
        }
        dscaIndex++;
        current_checksum_3:=p_m4_L1+1; // current_checksum_3=xor(xor(AES_K(x3), AES_K(x3)), AES_K(x3))=AES_K(x3)
        y_4:=current_checksum_3;
        τ:=y_4;
        return M, τ;
    }
    // end of adversary's attack
   // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
   if ((M, τ) is stored into the database S_DB_K) then return ⊥;
   answer:=V_K(M, τ);
   if (answer=1) then return 1;
   return 0;
}
```

C.7 – RIPE-MAC

In this Section, the security of the symmetric authentication scheme designated by RIPE-MAC (see [1]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the messages to authenticate have a length of at least one bit (|message|>0 bits); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.7) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

$T_K(M)$	$V_K(M, \tau)$
{	{
if (|M|=0) then return ⊥;	if (|M|=0) then return ⊥;
K1:=K;	if (|τ|≠128) then return ⊥; // |tag|=128
K2:=xor(K, $(11110000)^{16}$);	K1:=K;
tRemainder:=|M| mod 128;	K2:=xor(K, $(11110000)^{16}$);
p_m:=M||$10^{128\text{-tRemainder-}1}$;	tRemainder:=|M| mod 128;
n_p_m:=|p_m|/128;	p_m:=M||$10^{128\text{-tRemainder-}1}$;
for(j:=1; j≤n_p_m; j++)	n_p_m:=|p_m|/128;
{	for(j:=1; j≤n_p_m; j++)
p_m_j:=p_m[(j-1)*128...j*128-1];	{
}	p_m_j:=p_m[(j-1)*128...j*128-1];
$p\_m_{n\_p\_m+1}$:=<<|M|>>$_{128}$;	}
y_0:=<<0>>$_{128}$;	$p\_m_{n\_p\_m+1}$:=<<|M|>>$_{128}$;
for(i:=1; i≤n_p_m+1; i++)	y_0:=<<0>>$_{128}$;

{
$x1_i$:=xor(m_i, y_{i-1});
$x2_i$:=AES_{K1}($x1_i$);
y_i:=xor($x2_i$, m_i);
}
x3:=AES_{K2}^{-1}($y_{n\_p\_m+1}$);
τ:=AES_{K1}(x3);
return τ;
}
	for(i:=1; i≤n_p_m+1; i++)
	{
	$x1_i$:=xor(m_i, y_{i-1});
	$x2_i$:=AES_{K1}($x1_i$);
	y_i:=xor($x2_i$, m_i);
	}
	x3:=AES_{K2}^{-1}($y_{n\_p\_m+1}$);
	$\tau\_$:=AES_{K1}(x3);
	if ($\tau\_$=τ) then return 1;
	return 0;
	}

Fig. C.7

a) Unforgeability

i) Secure in the sense of EU-SCMA-SCA1

The studied scheme "appears" to be secure in the sense of EU-SCMA-SCA1.

C.8 – RMAC

In this Section, the security of the symmetric authentication scheme designated by RMAC (see [8]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the scheme only accepts messages whose length is a non-zero multiple of the length of the block of the block-cipher used (|message|=a1*128 bits, where a1 is an integer greater than zero); and the scheme only accepts tags whose length is 256 bits (|tag|=256 bits).

The scheme (see fig. C.8) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

T_K(M)	V_K(M, τ)

{
if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return ⊥;
K1:=K[0...127];
K2:=K[128...383];
n_m:=|M|/128;
for(j:=1; j≤n_m; j++)
{
m_j:=M[(j-1)*128...j*128-1];
}
R:=getRandomBits(128);
p_R:=R||0^{128};
xored_K2:=xor(K2, p_R);
y_0:=<<0>>$_{128}$;
	{
	if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return ⊥;
	if (|τ|≠256) then return ⊥; // |tag|=256
	K1:=K[0...127];
	K2:=K[128...383];
	n_m:=|M|/128;
	for(j:=1; j≤n_m; j++)
	{
	m_j:=M[(j-1)*128...j*128-1];
	}
	R:=τ[0...127];
	x2:=τ[128...255];
	p_R:=R||0^{128};

```
    for(i:=1; i≤n_m; i++)
    {
        x1_i:=xor(m_i, y_{i-1});
        y_i:=AES_{K1}(x1_i);
    }
    x2:=AES_{xored_K2}(y_{n_m});
    τ:=R||x2;
    return τ;
}
```

```
    xored_K2:=xor(K2, p_R);
    y_0:=<<0>>_128;
    for(i:=1; i≤n_m; i++)
    {
        x1_i:=xor(m_i, y_{i-1});
        y_i:=AES_{K1}(x1_i);
    }
    x2_:=AES_{xored_K2}(y_{n_m});
    if (x2_=x2) then return 1;
    return 0;
}
```

Fig. C.8

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt_{SA, Acma}^{eu-cma-sca1}(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack : (M, τ)←A_cma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    A_cma^{T0K(.), V1K(.,.), SCA(...)}(k, create)
    {
        dscaIndex:=0;
        M:=getRandomBits(384); // |message|=3*128
        τ:=T0_K(M);
        n_m:=|M|/128;
        for(j:=1; j≤n_m; j++)
        {
            m_j:=M[(j-1)*128...j*128-1];
        }
        temp_m3_1:=m_3;
        for(i:=1; i≤L1; i++)
        {
            M_i:=m_1||m_2||temp_m3_i;
            SCA(acquiring_request, i,)
            {
                V1_K(M_i, τ);
            }
            temp_m3_{i+1}:=SCA(dsca_request, i, dscaIndex, "x1_i:=xor(m_i, y_{i-1}), with i=3, tValue=y_2, and kValue=m_3", temp_m3_i);
        }
        dscaIndex++;
        y_2:=temp_m3_{L1+1};
        future_m_2:=getRandomBits(128);
        for(i:=1; i≤L1; i++)
        {
            M_i:=m_1||future_m_2||temp_m3_i;
            SCA(acquiring_request, i,)
            {
```

```
            V1_K(M_i, τ);
        }
        temp_m3_{i+1}:=SCA(dsca_request, i, dscaIndex, "x1_i:=xor(m_i, y_{i-1}), with i=3, tValue=y_2, and kValue=m_3", temp_m3_i);
    }
    dscaIndex++;
    future_y_2:=temp_m3_{L1+1};
    future_m_3:=xor(m_3, y_2); // x1_3=(m_3, y_2)
    future_m_3:=xor(future_m_3, future_y_2);
    M:=m_1||future_m_2||future_m_3;
    return M, τ;
  }
 // end of adversary's attack
 // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
 if ((M, τ) is stored into the database S_DB_K) then return ⊥;
 answer:=V_K(M, τ);
 if (answer=1) then return 1;
 return 0;
} |
```

C.9 – TMAC

In this Section, the security of the symmetric authentication scheme designated by TMAC (see [22]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the messages to authenticate have a length of at least one bit (|message|>0 bits); and the scheme only accepts tags whose length is equal to 128 bits (|tag|=128 bits).

The scheme (see fig. C.9) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
T_K(M)
{
   if (|M|=0) then return ⊥;
   K1:=K[0...127];
   K2:=K[128...255];
   tRemainder:=|M| mod 128;
   if (tRemainder=0) then
   {
      p_m:=M;
      xor_k:=shift_left(K2);
      if (lsb(K2)≠0) then
      {
        xor_k:=xor(xor_k, 0^120 10000111);
      }
   }
   else
   {
      p_m:=M||10^{128-tRemainder-1};
      xor_k:=K2;
   }
```

```
V_K(M, τ)
{
   if (|M|=0) then return ⊥;
   if (|τ|≠128) then return ⊥; // |tag|=128
   K1:=K[0...127];
   K2:=K[128...255];
   tRemainder:=|M| mod 128;
   if (tRemainder=0) then
   {
      p_m:=M;
      xor_k:=shift_left(K2);
      if (lsb(K2)≠0) then
      {
         xor_k:=xor(xor_k, 0^120 10000111);
      }
   }
   else
   {
      p_m:=M||10^{128-tRemainder-1};
      xor_k:=K2;
```

```
    n_p_m:=|p_m|/128;
    for(j:=1; j≤n_p_m; j++)
    {
        p_mj:=p_m[(j-1)*128...j*128-1];
    }
    p_mn_p_m':=xor(p_mn_p_m, xor_k);
    p_mn_p_m:=p_mn_p_m';
    y0:=<<0>>128;
    for(i:=1; i≤n_p_m; i++)
    {
        x1i:=xor(p_mi, yi-1);
        yi:=AESK1(x1i);
    }
    τ:=yn_p_m;
    return τ;
}
```

```
    }
    n_p_m:=|p_m|/128;
    for(j:=1; j≤n_p_m; j++)
    {
        p_mj:=p_m[(j-1)*128...j*128-1];
    }
    p_mn_p_m':=xor(p_mn_p_m, xor_k);
    p_mn_p_m:=p_mn_p_m';
    y0:=<<0>>128;
    for(i:=1; i≤n_p_m; i++)
    {
        x1i:=xor(p_mi, yi-1);
        yi:=AESK1(x1i);
    }
    τ_:=yn_p_m;
    if (τ_=τ) then return 1;
    return 0;
}
```

Fig. C.9

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt_SA, Acma^eu-cma-sca1(k)
{
    K←R—K(k);
    create database S_DBK;
    // begin of adversary's attack : (M, τ)←Acma^T0K(.), V1K(.,.), SCA(...)(k, create)
    Acma^T0K(.), V1K(.,.), SCA(...)(k, create)
    {
        dscaIndex:=0;
        τ:=getRandomBits(128); // |tag|=128
        M:=getRandomBits(384); // |message|=3*128
        n_m:=|M|/128;
        for(j:=1; j≤n_m; j++)
        {
            mj:=M[(j-1)*128...j*128-1];
        }
        p_m3_1:=m3;
        for(i:=1; i≤L1; i++)
        {
            temp_m3i:=p_m3_i;
            Mi:=m1||m2||temp_m3i;
            SCA(acquiring_request, i,)
            {
                V1K(Mi, τ);
            }
            p_m3_i+1:=SCA(dsca_request, i, dscaIndex, "p_mn_p_m':=xor(p_mn_p_m, xor_k), with n_p_m=3, tValue=xor_k, and
                                                                                    kValue=p_m3", p_m3_i);
```

```
        }
        dscaIndex++;
        xor_k:=p_m3_L1+1;
        p_m2_1:=m2;
        for(i:=1; i≤L1; i++)
        {
            Mi:=m1||temp_m2i||m3;
            SCA(acquiring_request, i,)
            {
                V1K(Mi, τ);
            }
            p_m2_i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(p_mi, yi-1), with i=2, tValue=y1, and kValue=p_m2", p_m2_i);
        }
        dscaIndex++;
        y1:=p_m2_L1+1;
        t_p_m31:=xor(m3, xor_k);
        for(i:=1; i≤L1; i++)
        {
            temp_m3i:=xor(t_p_m3i, xor_k);
            Mi:=m1||m2||temp_m3i;
            SCA(acquiring_request, i,)
            {
                V1K(Mi, τ);
            }
            t_p_m3i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(p_mi, yi-1), with i=3, tValue=y2, and kValue=p_m3", t_p_m3i);
        }
        dscaIndex++;
        y2:=p_m3_L1+1;
        current_m2:=xor(m3, xor_k);  // p_m3=xor(p_m3, xor_k), with initial p_m3=m3
        current_m2:=xor(current_m2, y2); // x13=xor(p_m3, y2);
        current_m2:=xor(current_m2, y1);
        for(i:=1; i≤L1; i++)
        {
            temp_m3i:=xor(t_p_m3i, xor_k);
            Mi:=m1||current_m2||temp_m3i;
            SCA(acquiring_request, i,)
            {
                V1K(Mi, τ);
            }
            t_p_m3i+1:=SCA(dsca_request, i, dscaIndex, "x1i:=xor(p_mi, yi-1), with i=3, tValue=y2, and kValue=p_m3", t_p_m3i);
        }
        dscaIndex++;
        current_y2:=temp_m3L1+1;
        future_y3:=current_y2;
        τ:=future_y3;
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DBK
    if ((M, τ) is stored into the database S_DBK) then return ⊥;
    answer:=VK(M, τ);
    if (answer=1) then return 1;
    return 0;
}
```

C.10 – XECB$-MAC

In this Section, the security of the symmetric authentication scheme designated by XECB$-MAC (see [46]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the scheme only accepts messages whose length is a non-zero multiple of the length of the block of the block-cipher used (|message|=a1*128 bits, where a1 is an integer greater than zero); and the scheme only accepts tags whose length is equal to 256 bits (|tag|=256 bits).

The scheme (see fig. C.10) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
T_K(M)
{
    if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return ⊥;
    K1:=K[0...127];
    K2:=K[128...255];
    r_0:=getRandomBits(128);
    y_0:=AES_K1(r_0);
    z_0:=AES_K2(r_0);
    n_m:=|M|/128;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*128...j*128-1];
    }
    m_n_m+1:=z_0;
    s_0:=<<0>>_128;
    checksum_0:=<<0>>_128;
    for(i:=1; i≤n_m+1; i++)
    {
        s_i:=(s_i-1+ y_0) mod 2^128;
        w_i:=(m_i+ s_i) mod 2^128;
        y_i:=AES_K1(w_i);
        checksum_i:=xor(checksum_i-1, y_i);
    }
    checksum:=checksum_n_m+1;
    τ:=r_0||checksum;
    return τ;
}
```

```
V_K(M, τ)
{
    if ((|M|=0) ∨ ((|M| mod 128)≠0)) then return ⊥;
    if (|τ|≠256) then return ⊥; // |tag|=256
    K1:=K[0...127];
    K2:=K[128...255];
    r0:=τ[0...127];
    checksum:=τ[128...255];
    y_0:=AES_K1(r_0);
    z_0:=AES_K2(r_0);
    n_m:=|M|/128;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*128...j*128-1];
    }
    m_n_m+1:=z_0;
    s_0:=<<0>>_128;
    checksum_0:=<<0>>_128;
    for(i:=1; i≤n_m+1; i++)
    {
        s_i:=(s_i-1+ y_0) mod 2^128;
        w_i:=(m_i+ s_i) mod 2^128;
        y_i:=AES_K1(w_i);
        checksum_i:=xor(checksum_i-1, y_i);
    }
    checksum_:=checksum_n_m+1;
    if (checksum_=checksum) then return 1;
    return 0;
}
```

Fig. C.10

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

Experiment **Expt**$_{SA,\ Acma}^{eu\text{-}cma\text{-}sca1}(k)$

```
{
    K←R—K(k);
   create database S_DBK;
    // begin of adversary's attack : (M, τ)←Acma^T0K(.), V1K(.,.), SCA(...)(k, create)
    Acma^T0K(.), V1K(.,.), SCA(...)(k, create)
    {
        dscaIndex:=0;
        τ:=getRandomBits(256); // |tag|=256
        M:=getRandomBits(384); // |message|=3*128
        n_m:=|M|/128;
        for(j:=1; j≤n_m; j++)
        {
            mj:=M[(j-1)*128...j*128-1];
        }
        r0:=τ[0...127];
        checksum:=τ[128...255];
       temp_m11:=m1;
        s1:=<<0>>128;
        s2:=<<0>>128;
        s3:=<<0>>128;
        dscaIndex2:=dscaIndex;
        dscaIndex++;
        for(i:=1; i≤L2; i++)
        {
            if (i=1) then r01:=r0;
            else
            {
                if (i=2) then r02:=(m1+s1) mod 2^128;
                else
                {
                    if (i=3) then r03:=(m2+s2) mod 2^128;
                    else
                    {
                        if (i=4) then r04:=(m3+s3) mod 2^128;
                        else r0i:=getRandomBits(128);
                    }
                }
            }
            τi:=r0i||checksum;
            for(j:=1; j≤L1; j++)
            {
                M_j:=temp_m1j||m2||m3;
                SCA(acquiring_request, j,)
                {
                    V1K(M_j, τi);
                }
                temp_m1j+1:=SCA(dsca_request, j, dscaIndex, "wi:=(mi+ si) mod 2^128, with i=1, tValue=s1, and kValue=m1",
                                                                                                    temp_m1j);
```

```
            }
            dscaIndex++;
            s1_i:=temp_m1_{L1+1};
            y0_i:=s1_i; // s1=xor(s_0, y_0), with s_0=<<0>>_128
            if (i=1) then
            {
                for(ctr:=1; ctr≤3; ctr++)
                {
                    if (ctr=1) then s_1:=s1_1;
                    else s_ctr:=(s_{ctr-1}+ y0_1) mod 2^128;
                }
            }
            for (ctr:=1; ctr≤3; ctr++)
            {
                t_m_{ctr_j}:=(r0_i+ s_ctr) mod 2^128;
            }
            M_i:=t_m_{1_j}||t_m_{2_j}||t_m_{3_j};
            t_checksum3_i:=y0_i; // t_checksum3_i=xor(xor(y0_i,y0_i), y0_i)=y0_i=s1_i=AES_{K1}(r0_i)
            SCA(acquiring_request, L1+i,)
            {
                V1_K(M_i, τ);
            }
            t_checksum3_{i+1}:=SCA(dsca_request, L1+i, dscaIndex2, "checksum_i:=xor(checksum_{i-1}, y_i), with i=4, tValue=y_4, and
                                                                    kValue=checksum_3", t_checksum3_i);
        }
        y_4:=t_checksum3_{L2+1};
        checksum_0:=<<0>>_128;
        // y_i=AES_{K1}(w_i)=y0_j, with w_i=(m_i+s_i) mod 2^128, i=1...3, and j=i+1
        for(ctr:=1; ctr≤3; ctr++)
        {
            checksum_ctr:=xor(checksum_{ctr-1}, y0_{ctr+1});
        }
        checksum_4:=xor(checksum_3, y_4);
        checksum:=checksum_4;
        τ:=r_0||checksum;
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
    if ((M, τ) is stored into the database S_DB_K) then return ⊥;
    answer:=V_K(M, τ);
    if (answer=1) then return 1;
    return 0;
}
```

C.11 – XMACR

In this Section, the security of the symmetric authentication scheme designated by XMACR (see [34]) is studied through the use of pseudo-code. To facilitate this study, the following particulars are considered: the underlying block-cipher is AES (see [37]); the scheme only accepts messages whose length is a non-zero

multiple of half the length of the block of the block-cipher used (|message|=a1*128/2 bits, where a1 is an integer greater than zero); and the scheme only accepts tags whose length is equal to 255 bits (|tag|=255 bits).

The scheme (see fig. C.11) and the attack that is undertaken against it are represented (in pseudo-code) by, respectively:

```
T_K(M)
{
    if ((|M|=0) ∨ ((|M| mod 64)≠0)) then return ⊥;
    r_0:=getRandomBits(127);
    n_m:=|M|/64;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*64...j*64-1];
    }
    w_0:=<<0>>_1||r_0;
    y_0:=AES_K(w_0);
    checksum_0:=y_0;
    for(i:=1; i≤n_m; i++)
    {
        w_i:=<<1>>_1||<<i>>_63||m_i;
        y_i:=AES_K(w_i);
        checksum_i:=xor(checksum_i-1, y_i);
    }
    checksum:=checksum_n_m;
    τ:=r_0||checksum;
    return τ;
}
```

```
V_K(M, τ)
{
    if ((|M|=0) ∨ ((|M| mod 64)≠0)) then return ⊥;
    if (|τ|≠255) then return ⊥; // |tag|=255
    r_0:=τ[0...126];
    checksum:=τ[127...254];
    n_m:=|M|/64;
    for(j:=1; j≤n_m; j++)
    {
        m_j:=M[(j-1)*64...j*64-1];
    }
    w_0:=<<0>>_1||r_0;
    y_0:=AES_K(w_0);
    checksum_0:=y_0;
    for(i:=1; i≤n_m; i++)
    {
        w_i:=<<1>>_1||<<i>>_63||m_i;
        y_i:=AES_K(w_i);
        checksum_i:=xor(checksum_i-1, y_i);
    }
    checksum_:=checksum_n_m;
    if (checksum_=checksum) then return 1;
    return 0;
}
```

Fig. C.11

a) Unforgeability

i) Insecure in the sense of EU-CMA-SCA1

```
Experiment Expt_SA, Acma^eu-cma-sca1(k)
{
    K←R—K(k);
    create database S_DB_K;
    // begin of adversary's attack : (M, τ)←A_cma^T0K(.), V1K(.,.), SCA(...)(k, create)
    A_cma^T0K(.), V1K(.,.), SCA(...)(k, create)
    {
        dscaIndex:=0;
        M0:=getRandomBits(192); // |message|=3*64
        n_m:=|M0|/64;
        for(j:=1; j≤n_m; j++)
        {
```

```
            m_j:=M0[(j-1)*64...j*64-1];
        }
        m_4:=getRandomBits(64);
        for(i:=1; i≤L2; i++)
        {
            τ_i:=T0_K(M0);
            r0_i:=τ_i[0...126];
            checksum_i:=τ_i[127...254];
            checksum3_i:=checksum_i;
            t_M_i:=m_1||m_2||m_3||m_4;
            SCA(acquiring_request, i,)
            {
                V1_K(t_M_i, τ_i);
            }
            checksum3_{i+1}:=SCA(dsca_request, i, dscaIndex, "checksum_i:=xor(checksum_{i-1}, y_i), with i=4, tValue=y_4, and
                                                                                    kValue=checksum_3", checksum3_i);
        }
        dscaIndex++;
        y_4:=checksum3_{L2+1};
        checksum:=xor(checksum_1, y_4);
        M:=m_1||m_2||m_3||m_4;
        τ:=r0_1||checksum;
        return M, τ;
    }
    // end of adversary's attack
    // by definition, the pair of values (M, τ) is not stored into the database S_DB_K
    if ((M, τ) is stored into the database S_DB_K) then return ⊥;
    answer:=V_K(M, τ);
    if (answer=1) then return 1;
    return 0;
}
```

Appendix D – Contents of the long version of the paper

The long version of this paper, besides describing the SCA1 Model, is characterized for studying the symmetric encryption setting, the asymmetric encryption setting, the symmetric authentication setting, the asymmetric authentication setting; and the *signcryption* setting.

In the symmetric encryption setting, the schemes are studied in accordance with the following goals:

- "Confidentiality of Encryptions" goal;
- "Unforgeability of Ciphertexts" goal;
- "Offline Entity-Privacy" goal;
- "Online Entity-Privacy" goal;
- "Pseudorandomness of Encryptions" goal;
- "Recovery of Symmetric-Key's Bit" goal;
- "Key-Collisions" goal.

In the asymmetric encryption setting, the schemes are studied in accordance with the following goals:

- "Confidentiality of Encryptions" goal;
- "Special Unforgeability of Ciphertexts" goal;
- "Trapdoor One-Wayness" goal;
- "Offline Entity-Privacy" goal;
- "Online Entity-Privacy" goal;
- "Pseudorandomness of Encryptions" goal.

In the symmetric authentication setting, the schemes are studied in accordance with the following goals:

- "Unforgeability of Tags" goal;
- "Offline Entity-Privacy" goal;
- "Online Entity-Privacy" goal;
- "Pseudorandomness of Tags" goal;
- "Recovery of Symmetric-Key's Bit" goal;
- "Key-Collisions" goal.

In the asymmetric authentication setting, the schemes are studied in accordance with the following goals:

- "Unforgeability of Tags" goal;
- "Offline Entity-Privacy" goal;
- "Online Entity-Privacy" goal;
- "Pseudorandomness of Tags" goal.

In the *signcryption* setting, the schemes are studied in accordance with the following goals:

- "Confidentiality of Encryptions" goal;
- "Special Unforgeability of Ciphertexts" goal;
- "Trapdoor One-Wayness" goal;
- "Offline Entity-Privacy" goal;
- "Online Entity-Privacy" goal;
- "Pseudorandomness of Encryptions" goal.

The goal designated by "Confidentiality of Encryptions" aims testing if an adversary that is given challenge ciphertexts is capable of discovering anything about the corresponding plaintexts.

The goal designated by "Unforgeability of Ciphertexts" aims testing if an adversary is capable of creating a new pair of values, ciphertext and associated data value, for which the integrity is confirmed. The goal designated by "Special Unforgeability of Ciphertexts" is a variation of this goal that is adapted to both the asymmetric encryption setting and the *signcryption* setting.

The goal designated by "Pseudorandomness of Encryptions" aims testing if an adversary is capable of distinguishing the ciphertexts created by the scheme to test, from strings (of the same length of the latter ciphertexts) selected uniformly at random from the domain of these ciphertexts.

The goal designated by "Recovery of Symmetric-Key's Bit" aims testing if an adversary is capable of recovering any of the bits of the symmetric key used by the instance of the scheme to test.

The goal designated by "Key-Collisions" aims testing if an adversary is capable of recovering any of the keys corresponding to one of the different instances of the scheme to test that are in use.

The goal designated by "Unforgeability of Tags" aims testing if an adversary is capable of creating a new pair of values, tag and message, for which the integrity is confirmed. This goal corresponds to the security notion designated by "Existential Unforgeability", which was introduced in this paper.

The goal designated by "Pseudorandomness of Tags" aims testing if an adversary is capable of distinguishing the tags created by the scheme to test, from strings (of the same length of the latter tags) selected uniformly at random from the domain of these tags.

The goal designated by "Trapdoor One-Wayness" aims testing if the way in which the scheme to test defines the use of the asymmetric encryption trapdoor allows the adversary to invert the latter.

The goal designated by "Offline Entity-Privacy" aims testing if an adversary that is given ciphertexts (or tags) created by one of two possible entities is capable of discovering to which of those two entities correspond the latter ciphertexts (or tags)[76].

The goal designated by "Online Entity-Privacy" aims testing if an adversary that is allowed to undertake SCAs against the processing executed by one of two possible entities is capable of discovering to which of those two entities corresponds the latter processing[77].

76 This goal was introduced in [25] for the asymmetric encryption setting (of the SCA0 Model).

77 An introduction to this goal was presented at the Rump Session of CHES 2004 (see http://mind-security.com/papers/2).