Authentication with RIPEMD-160 and other alternatives: A Hardware Design Perspective

H. Michail(1) , A. Gregoriades (2), V. Kelefouras(1),
G. Athanasiou(1) , A. Kritikakou(1) and C. Goutis(1),
(1)Electrical & Computer Engineering Department, University of Patras, Greece
(2) Computer Science & Engineering Department, European University, Cyprus

1. Abstract

Taking into consideration the rapid evolution of communication standards that include message authentication and integrity verification, it is realized that constructions like MAC and HMAC, are widely used in the most popular cryptographic schemes since provision of a way to check the integrity of information transmitted over or stored in an unreliable medium is a prime necessity in the world of open computing and communications.
MACs are used so as to protect both a message's integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content. In every modern cryptographic scheme that is used to secure a crucial application that calls for security, a keyed-hash message authentication code, or HMAC, is incorporated. Beyond HMAC, a block cipher algorithm is also incorporated (i.e like AES), thus resulting to the whole security scheme. The proposed hardware design invokes a number of optimizing techniques like pipeline, evaluation-based partial unrolling, certain algorithmic transformations in space and time and computational re-ordering, leading to a high-throughput and low-power design for the whole HMAC construction. Finally, a new algorithm, CMAC, for producing message authenticating codes (MACs) which was recently proposed by NIST, is also described. The proposed security scheme incorporates a FIPS approved and a secure block cipher algorithm (that might have already been deployed in the security scheme) and was standardized by NIST in May, 2005. This work concludes with an efficient hardware implementation of the CMAC standard.

2. Introduction

Security issues are crucial as long as the further spread of internet applications is concerned. This is true particularly today considering the fact that sensitive data are stored in networked computers and transferred via several types of networks. It is critical that this sensitive data are protected from eavesdroppers, thieves or generally anyone who tries to illegally acquire the sensitive data. Thus it is important to store the data safely and in such way that will discourage anyone from getting into them. Beliefs and claims for lack of security in our internet transactions could lead someone to assume that is the main reason for many people to avoid transactions via internet.

Cryptography is the scientific field that among other has to do with offering security over internet. Cryptography is a significant weapon in our quest to protect our treasure: the sensitive data. Seen as a process, we can assume that cryptography is nothing more than a simple mathematic function that transforms one set of numbers (or characters, objects etc) to another. This means that it is used to change comprehensible data (messages) to a seemingly corrupted group of numbers (characters). Cryptography services can be conducted by a wide range of applications offering specific security requirements like authentication, integrity, non-repudiation, etc. Cryptographic features are incorporated in almost every application that has to do with transactions that handle personal and sensitive data.

Applications like the Public Key Infrastracture (PKI) (NIST, 2001 b), IPSec (NIST, 2005 a), Secure Electronic Transactions (SET) (Loeb, 1998), and the 802.16 (Johnston & Walker, 2004) standard for Local and Metropolitan Area Networks incorporate authenticating and other services. These other services often include encryption but in this chapter we will focus on ways to achieve authentication efficiently through different mechanisms and on how to achieve high performance implementations.

All the pre-reffered applications presuppose that an authenticating module that includes a certain cryptographic function called “hash function” is nested in the implementation of the application. Hash function H(M) is a transformation that takes an input message M and returns a fixed-size string, which is called the hash value h (that is, h = H(M)). Hash functions with just this property have a variety of general computational uses, but when employed in cryptography, the hash functions are usually chosen to have some additional properties. The basic requirements for a cryptographic hash function are as follows.

- The input can be of any length.
- The output has a fixed length.
- H(x) is relatively easy to compute for any given x.
- H(x) is one-way.
- H(x) is collision-free.

A hash function H is said to be one-way if it is hard to invert, where “hard to invert” means that given a hash value h, it is computationally infeasible to find some input x such that H(x) = h. If, given a message x, it is computationally infeasible to find a message y not equal to x such that H(x) = H(y), then H is said to be a weakly collision-free hash function. A strongly collision-free hash function H is one for which it is computationally infeasible to find any two messages x and y such that H(x) = H(y). The hash value represents concisely the longer message or document from which it was computed meaning that the target of a hash function is to provide a “signature” of M that is unique. One can think of the hash value as a "digital fingerprint" of the larger document.

From the algorithmic part of view, hash functions have a lot of resemblances to compression algorithms. Hash functions are iterative algorithms, which in order to compute the final hash value, perform a number of identical or slightly different operations. Each operation is based on a special block that receives as inputs, a message block from the initial message M of arbitrary length and the outputs of the previous blocks of the message. It is common to reference the final hash value as Message Digest (MD) as a tribute to Professor Ronald Rivest who authored the MD family hash functions.

Other applications that require the usage of a hash function for authentication are the Virtual Private Networks (VPN's) (NIST, 2005 a) that companies are establishing in order to exploit on-line collaboration. Moreover digital signature algorithms like DSA (NIST, 1994) that are used for authenticating services like electronic mail, electronic funds transfer, electronic data interchange, software distribution, data storage etc are based on using a critical cryptographic primitive like hash functions. Hashes are also used to identify files on peer-to-peer file sharing networks. For example, in ed2k links which are links used by the eDonkey2000 family of P2P programs, such as eDonkey2000 itself, e-Mule, Hybrid MLDonkey, and others. These links are primarily used to provide a unique identifier for file without referring to a specific location, like in FTP and HTTP links. In an ed2k link the hash is combined with the file size, providing sufficient information for locating file sources, downloading the file and verifying its contents.

Furthermore cryptographically secure hashes are used in signed code systems such as Microsoft's Windows Update (Microsoft, 2005) to ensure the integrity and authenticity of downloaded programs and patches. Hashes are also used to periodically check on-disk code integrity in systems such as Tripwire (Kim & Spafford, 1994). Last but not least US federal on 2004 came up with a new plan to help secure electronic voting (Biever, 2004), employing a hash function so as to achieve higher level of security and therefore more confidence in e-voting than ever before.

Hash functions serve a dual role in practical signature schemes: they expand the domain of messages that can be signed by a scheme and they are an essential element of the scheme's security (Mironov, 2005). Hashing cores are also critical for security in networks and mobile services, as in SSL (Stephen, 2000), which is a Web protocol for establishing authenticated and encrypted sessions between Web Servers and Web clients. A common method for client authentication is to require the client to present password previously registered with the server. Storing passwords of all users on the server poses an obvious security risk. Fortunately the server need not know the passwords - it may store their hashes (together with some "salt" to frustrate dictionary attacks) and use the information to match it with the hashes of alleged passwords (Morris & Thompson, 1979).

3. MACs: HMAC and CMAC

Hashing functions are also the main modules that exist in the HMAC (NIST, 2007 a) algorithm that produces Message Authentication Codes (MACs). A cryptographic message authentication code (MAC) is a short piece of information used to authenticate a message. A MAC algorithm accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a MAC (sometimes known as a tag). The MAC value protects both a message's integrity as well as its authenticity, by allowing verifiers (who also possess the secret key) to detect any changes to the message content.

A keyed-hash message authentication code, or HMAC, is a type of message authentication code (MAC) calculated using a cryptographic hash function in combination with a secret key. As with any MAC, it may be used to simultaneously verify both the data integrity and the authenticity of a message. Any iterative cryptographic hash function may be used in the calculation of an HMAC. The cryptographic strength of the HMAC depends upon the cryptographic strength of the underlying hash function, on the size and quality of the key and the size of the hash output length in bits.

Taking into consideration the rapid evolution of the communication standards that include message authentication and integrity verification, we realize that constructions like HMAC, are widely used in the most popular cryptographic schemes since provision of a way to check the integrity of information transmitted over or stored in an unreliable medium is a prime necessity in the world of open computing and communications. Typically, message authentication codes are used between two parties that share a secret key in order to authenticate information transmitted between these parties.

MACs differ from digital signatures, as MAC values are both generated and verified using the same secret key. This implies that the sender and receiver of a message must agree on keys before initiating communications, as is the case with symmetric encryption. For the same reason, MACs do not provide the property of non-repudiation offered by signatures: any user who can verify a MAC is also capable of generating MACs for other messages.

The most common way of producing a MAC is based on the incorporation of a cryptographic hash function, thus resulting to keyed-hash message authentication code, or HMAC. Besides authentication with hash functions, another very popular service that is also incorporated in most applications is encryption/decryption process which is based on the usage of a block cipher algorithm. So it seems that is also useful to have a method for producing MACs using such block cipher algorithms in cases where we want to avoid producing MACs using hash functions.

It was only on May 2005 when NIST (National Institute of Standards and Technology) decided to standardize a new algorithm for producing message authenticating codes (MACs) using secure block cipher algorithms. This new method is the CMAC standard (NIST, 2005 b). This algorithm was standardized recently and thus no many implementations have been presented concerning the CMAC standard. Moreover it has not been widely adopted by industry since its advantages are major but only in certain cases.

Another reason for the non-development of hardware CMAC implementations is that the functionality of a CMAC can be incorporated in a security scheme with an HMAC which uses a hash function and this is the case in most of today's applications. Industry must be persuaded for the necessity of this algorithm if it is to start introducing CMAC hardware implementations, which are costly comparing to software implementations. Later in this section a specific case of an application (e-voting) that can be benefited a great deal from the usage of CMAC instead of HMAC is going to be commented so as to bring out this necessity.

This necessity mainly has to do with the numerous applications that do not call for high throughput implementations (as long as the security part is concerned) but for small-sized or low-power implementations. In these cases the usage of CMAC instead of HMAC is ideal since there is no need to incorporate both a hash function and a cipher block but only a block cipher algorithm like AES (NIST, 2001 a) that can handle all aspects of security in the certain security scheme.

Such security applications are those used in e-voting where there is a lot of time available for the transaction to be performed and thus for the sake of small-sized and low power implementations CMAC seems to be a very good solution. The whole application of e-voting consists of numerous widespread portable clients located at places where people vote and few central servers which collect data from all distributed clients. It is obvious that these sessions must be secured so some kind of security scheme should be used. Because the voter has, relatively speaking, a lot of time to interact with the voting device there is no great need

for high throughput implementations as long as the portable clients are concerned. Instead it is a prior necessity that these portable voting machines are small, have reduced power dissipation and are cheap due to the great number of these devices that are needed in e-voting application. However it has to be mentioned that there is a need for high throughput implementation for the corresponding server of the application.
So in this case it seems that the best solution is to use CMAC for authentication instead of HMAC and fully exploit the incorporated cipher block algorithm that is nested in the numerous clients and few servers to encrypt the transmitted data. This way the cost is dramatically reduced since only a block cipher is used instead of a block cipher and a hash function. The same goes for power dissipation and integration area of the portable voting devices. As long as the corresponding servers are concerned multiple instances of the block cipher can be incorporated and certain design techniques can be used so as to achieve high throughput implementations.
Moreover it has to be noticed that this is a trend in our days, considering many standards are either under consideration or published by NIST in 2007 like XTS (NIST, 2007 b) and GCM (NIST, 2008) where security schemes tend to use a cipher block like AES in order to handle all security aspects in an application. It seems that there is a tendency to use only one basic cryptographic component i.e block cipher algorithm AES and based on that to broaden offered security services over storage devices, wireless networks and so on.
Another great advantage of this perspective is that the time-to-market constraint is shorter, besides the reduced cost. The need of providing all kind of security services with the usage of only one basic cryptographic component has emerged, as it is clearly seen especially during the last year with the hot debate about XTS and GCM .
CMAC like XTS and GCM focuses on the service that intends to offer ensuring the security level and simplifying the whole implementation without using sophisticated algorithms for each case. This way we manage to save a significant amount of integration area and power consumption, reduce the design and implementation complexity and offer the chance to concentrate only on the optimization of the incorporated block cipher algorithm in order to improve the performance of the whole security scheme.
Thus in applications where there are a few servers (for centralized data process) and a great number of distributed portable clients (for data collection), that need to be cheap, small and have minor power consumption without great concern about throughput, like in e-voting, the adoption of CMAC for MAC generation seems to be ideal since it will result to efficient solutions with great reliability.
At the end of this chapter CMAC mechanism will be analysed and finally implementation details concerning CMAC will be provided in brief considering the incorporated block cipher algorithm as an installable component.

4. Current Status and Design Issues

It is quite clear that all the pre-referred applications that incorporate security mechanisms producing MACs either with the usage of a hash function (HMAC) or a block cipher (CMAC) are addressing more users/clients and thus, the increase of their throughput is a prior necessity, particularly for the corresponding server of these services. This is because the cryptographic system, especially the server, has to reach the highest degree of throughput in order to satisfy immediately all requests for service from all users-clients. In some of these cryptographic schemes, where a hash function is used for the authentication part, the throughput of the incorporated hash functions determines the throughput of the whole security scheme.

Many leading companies offer and also improve hardware implementations of hash functions. Although software encryption is becoming more prevalent today, hardware is the embodiment of choice for military and serious commercial applications (Schneier, 1996). The NSA, for example, authorizes only encryption in hardware.

The first reason to do so is speed. Many cryptographers have tried to make their algorithms more suitable for software implementations; however specialized hardware will always win the speed race. Additionally, hashing is often a computation-intensive task. Tying up the computer's primary processor for this is inefficient. Moving hashing tasks to another chip, even if that chip is just another processor, makes the whole system faster.

Efficient software implementations of hash functions have been extensively studied in the literature (McCurley, 1994), (Nakajima & Matsui, 2002) and (Ballard, 2004). All these studies verify that hardware implementations are much faster than the corresponding software implementations.

The second reason is security. An encryption algorithm running on a generalized computer has no physical protection. A malicious attacker is able to go in with various debugging tools and surreptitiously modify the algorithm without anyone ever realizing it. Hardware encryption devices can be securely encapsulated to prevent this. In (Van Oorschot et al., 2005) the security problems of software implementations were proven for a great variety of different general-purpose processors (including UltraSparc, x86, PowerPC, AMD64, Alpha, and ARM).

The latter mentioned facts were strong motivation to propose some novel techniques and correspondingly designs and implementations for a certain hash function like RIPEMD-160 which was developed in the framework of the EU project RIPE (Race Integrity Primitives Evaluation).The proposed designs introduce a small area penalty but manage to increase significantly the overall throughput of the hash core. Our future goal is to manage to combine these techniques and propose an ultra high throughput design of the RIPEMD-160 hashing algorithm.

This high throughput is essential for certain cryptographic applications such as HMAC mechanism in security schemes of IPSec and SSL/TLS. The security schemes for these applications incorporate encryption and authenticating modules. Lately many implementations of the AES encryption modules have been designed that exceed or approach the limit of 20 Gbps of Throughput (Hodjat & Verbauwhede, 2004) and operate at very high operating frequencies. So it is crucial to design hash functions that also achieve high throughput and this can be done with the proposed techniques.

However it has to be noticed that RIPEMD-160 is not widely adopted and thus there has not been conducted thorough research in antithesis to MD-5, SHA-1, SHA-256 etc. This chapter intends to be a guide for alternatives ways to achieve authentication such as HMACs with RIPEMD-160, and CMACs with the usage of a block cipher algorithm. Above that, it also intends to propose optimizations for the corresponding hardware designs and implementations as it has already been achieved for more popular hash functions that are also used for authenticating purposes.

5. HMAC RIPEMD-160: A Hardware Design Perspective

5.1 RIPEMD-160 hash function design architecture

In this section an efficient pipelined design and implementation of the RIPEMD-160 (Dobbertin et al., 1996) hash function is going to be presented. Then in the following section novel techniques are going to be presented and used in order to optimize the presented implementation so as to achieve higher througput that is essential for servers in security schemes of IPSec, SSL/TLS etc. Finally at the end of this section the whole HMAC-RIPEMD-160 system architecture is going to be presented incorporating the optimized version of RIPEMD-160 that will be demonstrated.

The fixed length of each of the processed blocks in RIPEMD-160 is 512 bits. As it is described in (Dobbertin et al., 1996) Ripemd-160 hash function has a bitsize of the hash-result and chaining variable of 160 bits (five 32-bit words). Ripemd-160 uses two parallel processes of five rounds, with sixteen operations for each round (5 x 16 operations for the whole process).The input in every round is five bit words of data, the message word X_i and a 32 bit constant K_i.

Unfolding the expressions of a_t, b_t, c_t, d_t, e_t that describe the five input words, it is observed that b_{t-1}, c_{t-1} , d_{t-1} , e_{t-1} values are assigned directly to outputs c_t, d_t, e_t, at respectively. In Eq. (1) the expressions of a_t, b_t , c_t , d_t and e_t are defined.

$$\begin{gathered} e_t = d_{t-1} \\ d_t = ROL_{10}(c_{t-1}) \\ c_t = b_{t-1} \\ b_t = e_{t-1} + ROL_s[\, f_t(b_{t-1}, c_{t-1}, d_{t-1}) + a_{t-1} + X_i + K_j \,] \\ a_t = e_{t-1} \end{gathered} \tag{1}$$

where $ROL_x(y)$ represents cyclic shift (rotation) of word y to the left by x bits and $f_t(z, w, v)$ denotes the non-linear function which depends on the round being in process. From Eq.(1), it can be derived that the maximum delay is observed on the calculation of the b_t, value.

The operation block of RIPEMD-160 is illustarted in Figure 1 where it can be observed that the critical path consists of three addition stages and a multiplexer via which the values pass each time to/and feed the operation block.

The Ripemd-160 hash function consists of two parallel processes of five rounds, with sixteen transformations for each round (5 x 16 transformations for the process). All the rounds in the processes are similar but each one performs a different operation on five 32-bit inputs. The data (input message) are processed 16 times in each transformation round resulting in 80 transformations performed in total per process. Each operation is performed in every clock cycle and a certain 32-bit word X_i has to be supplied to the transformation round. This 32-bit X_i results from a specific process on the 512-bit input block_in through appropriate permutations on the initial 16 X_i word blocks that are exported from 512-bit input block.

The computation of each X_i (beyond the first 16 X_i that are computed by a simple split of the 512-bit input block_in) takes place in a specially designed unit named "Xi' permutation" unit.

The implemented Ripemd-160 core has five primary inputs h0, h1, h2, h3, h4 where the initial values are supplied. These can be the ones specified by the standard or others that have occurred from process of the former block of the message in case that the input message consists of more than one block. However in most cases the input intended to be authenticated is just one block. In the unlikely event of messages that consist of many blocks a scheduler is needed in order to apply pipelines stages. This chapter will not focus on this scheduler, as it is sure that it will go beyond the scope of the book.

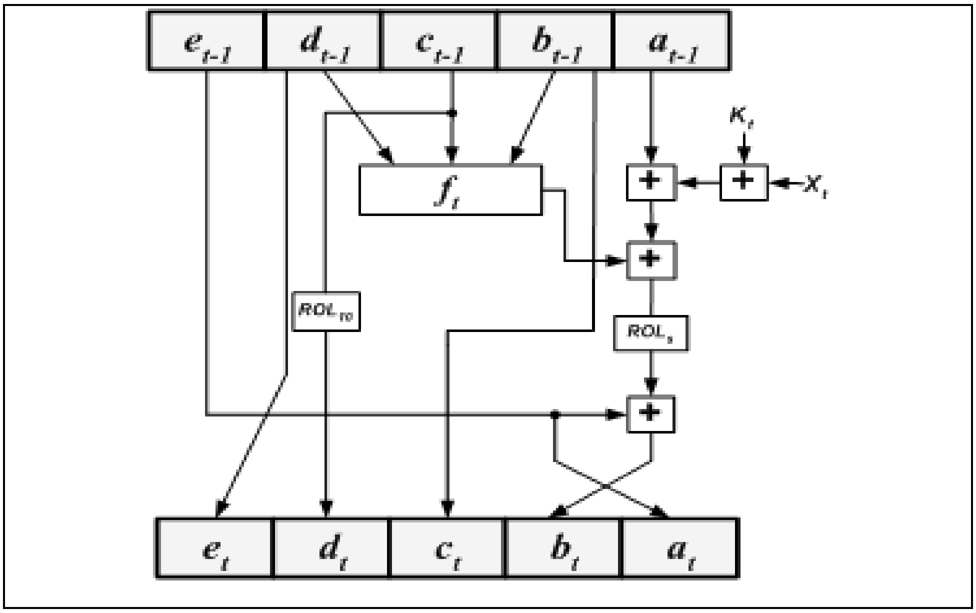

Fig. 1. RIPEMD - 160 operation block (also mentioned as "Round")

The proposed system architecture is illustrated in the following Figure 2. This implementation is suitable for any system that uses the Ripemd-160 hash function, constrained only by the assumption that the start_counter1 signal is applied one clock cycle before the final padded message is available in the 512-bit input block_in of Ripemd-160 core. Concurrently to the arrival of the 512-bit input block_in the signal start_round1 has to be applied and 80 clock cycles later the hash value is pending on the 160-bit output ripemd-160 hash value of the Ripemd-160 core.
In order to apply pipeline stages to the Ripemd-160 core, ten 160-bit registers have to be used between the data transformation rounds. In Figure 2, these registers are implanted at the end of each transformation round and this technique ensures that five 512-bit data blocks can be processed at the same time and finally a 160-bit message digest is produced every 16 clock cycles. All multiplexers are controlled by the Count_16 component. The Count_16 component also arranges when the 16 X_i's - that are pending in the input of the register and are related to the process in the next round - will be stored in the register. Taking in consideration that five messages can be concurrently processed, 2*5*16 X_i's (160 X_i's for every message) must be computed and stored. Instead of this, for every message only 2*16 X_i's are computed and saved, those used in the currently processed transformation round for the two processes. So, this way only 2*5*16 X_i's (32 X_i's for every message) must be computed and stored. The Xi's for the next transformation round of every message are computed and pending in the "Xi' permutation" units. When a message proceeds to the next

transformation round, the Xi's needed in the new round are stored in the corresponding registers. These registers supply the new round with the 16 X_i's and at the same time the 16 X_i's required for the next round, for the same message, are computed and pending on the corresponding "Xi' permutation" unit.

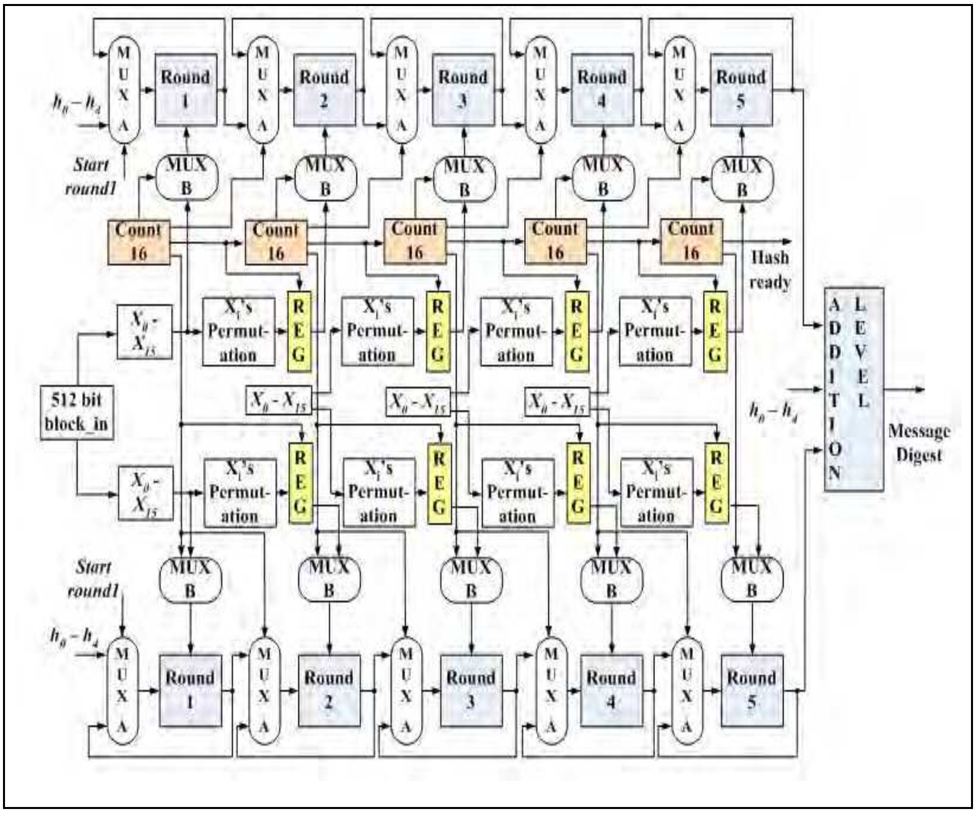

Fig. 2. Ripemd-160 Core

Implanting registers at the end of every transformation round and considering the fact that the process in every round lasts 16 clock cycles leads to the following conclusion; the implemented Ripemd-160 core can be supplied with a new 512-bit input block_in if only 16 clock cycles have past from the last time instance that the Ripemd-160 core was again supplied. This leads to a maximum throughput limit for the Ripemd-160 core which is determined by the fact that at each processing instance only 5 different messages at most can be concurrently at process. During this 16 clock cycles the 512-bit input block_in has to remain stable in order for the Ripemd-160 core to function properly. This way we save up time (1 clock cycle) and hardware that would be spent if it was decided to store in registers the 512-bit input block_in. This wouldn't improve at all the proposed implementation and it doesn't incur any design or functional problems.
The 512-bit input block_in is kept stable to the output of the unit that is responsible for the

padding procedure and will be mentioned to the next section. The five Count_16 components are responsible for the synchronization of all procedures that have to be done in order to obtain the hash value. Each one is enabled only when in the corresponding transformation round there is a message that is being processed. When the process in that round reaches the end the Count_16 component arranges that the process will continue to the next round. The five Mux_A multiplexers for each process ensure that every time the process of a message ends in one round, the process will be continued in the next round. The five Mux_B multiplexers also for each process, ensure that, at any time instance, the correct X_i is supplied to each operation of the round during the process of a message.

The presented implementation illustrates all the necessary implementation details and it is also indented to serve as a reference implementation to compare with in order to propose some improvements in its architecture which correspondingly are going to increase the performance of the whole RIPEMD hashing core as well as the corresponding HMAC implementation.

The latter presented hashing core was captured in VHDL and was fully simulated and verified using the Model Technology's ModelSim Simulator. Verification of the designs' operation was achieved using a large set of test vectors, apart from the test example proposed by the standards. The synthesis tool used to port VHDL to the targeted technologies was Synplicity's Synplify Pro Synthesis Tool. Simulation of the designs was also performed after synthesis, exploiting the back annotated information that was extracted from the synthesis tool. Further evaluation of the designs was performed using the prototype board for the Xilinx Virtex device family. The achieved operating frequency, the required integration area and the corresponding throughput in three different FPGA families are pictured in Table 1.

Technology	Area (CLBs)	Frequency (MHz)	Throughput (Gbps)
Virtex	1798	51.6	1.65
Virtex-E	1856	64.6	2.08
Virtex-II	1985	74.6	2.38

Table 1. RIPEMD-160 implementation synthesis result

The throughput it is calculated from Eq.2.

$$\text{Throughput} = \#\text{bits} * f_{\text{operation}} \,/\, \#\text{operations} \qquad (2)$$

where #bits is equal to the number of bits processed by the hash function, #operations corresponds to the required clock cycles between successive messages to generate each Message Digest and $f_{operation}$ indicates the maximum operating frequency of the circuit.

The results of Table 1 give a rough estimation of the performance of the proposed implementation in various FPGA platform boards. When placement and routing takes place, using vendor's tools, a more realistic performance of the RIPEMD implementation arises which slightly varies from the reported results in Table 1. The implementation is synthesizable in most FPGA families, resulting in a reusable, general purpose implementation.

5.2 Optimized design of RIPEMD-160 hash function

There is a big variety of choices that can be made at early stages of the design/development process to help us achieve increased performance. Although the need for high throughput is recognized, the performance of all hardware implementations concerning hash functions is degraded because not much effort has been paid on optimizing the inner logic of the transformation rounds. Instead academia and industry has been focused on using system level techniques like loop unrolling, pipeline etc, but they have not develop techniques able to optimize the critical path of these hash functions that is located in the transformation round as it analyzed previously. In this section two different techniques aiming to achieve this, are going to be presented. In each case the achieved optimization will also be stated. The next step of our research team is to combine these techniques with others in a certain methodology aiming to optimize the RIPEMD-160 hash function. This goal has already been achieved by our research team regarding other hash functions like MD-5, SHA-1, SHA-256 etc. [Michail et al, 2008].

The proposed technique to increase the operation block's frequency is based on algorithmic transformations allowing simultaneous spatial and temporal transformations. Spatial transformations are those that manipulate and re-order the position of the available resources in order to generate the same output, gaining either in performance or in power dissipation. Temporal transformations are those that manage values highly dependant from time and sequence of appearing. Applying a set of transformations on the algorithms of RIPEMD-160 it is possible to increase the hash core's performance significantly. A major advantage of applying a transformation on the algorithm is the capability to know explicitly the penalty this action requires to be paid.

In particular these two techniques are:

1) Spatial Pre-computation of additions contributing to the critical path.

2) Temporal Pre-computation of some values that are needed in following operations.

Examining the expressions described in (1) and represented in Figure 1, it is observed that some input values are assigned directly to some output values respectively. From Figure 1, it is derived that the maximum delay is observed on the calculation of the b_t, value. Obviously the critical path consists of three addition stages and a multiplexer that feeds back the operation block.

A notice that can be made by observing Figure 2 is that some outputs are derived directly from some inputs values respectively. So we can assume that it is possible during one operation to pre-calculate some intermediate values that will be used in the next operation. Therefore, while the main calculations are in progress, at the same time some values that are going to be needed in the next operation can also be in progress of calculation. Furthermore, moving the pipeline stage to an appropriate intermediate point we can store these intermediate calculated values, the critical path is divided resulting in a decrease of the maximum delay without paying any worth-mentioning area penalty.

These pre-calculations are applied only to those output values that derive directly (hardwired) from inputs. So while the main calculations are in progress, some intermediate values needed at the next operation can also be in progress of calculation. Further benefits from the pre-calculation process can be achieved by re-ordering in space the registers. Moving the pipeline stage to an appropriate intermediate point to store these intermediate calculated values, the critical path is divided resulting in a decrease of the maximum delay with negligible, if any, area penalty.

It should be pointed out that the existing pipeline registers are moved somewhere in the middle of each operation block so as to shorten the critical path and no extra inner-loop pipeline stage is introduced. This means that the existing four pipeline registers (for outer-route pipelining) are re-arranged in space, and it is not the case that more pipeline registers (for inner-round pipelining) are applied. Re-ordering of existing registers results in disjoining dependencies of the outputs from input signals. Although the operation block was changed, the logic it implemented was the same, and at the same time the critical path was shortened.

Thus, the RIPEMD-160 equations representing Figure 1 is transformed to generate the intermediate values a_{t-1}*, b_{t-1}*, c_{t-1}*, d_{t-1}*, e_{t-1}* and g_{t-1} as illustrated in Figure 3. In Figure 3 the pre-computation technique applied in RIPEMD-160 hash function is illustrated. Each operation block now consists of two units the "Pre-Computation" unit which is responsible for the pre-computation of the values that are needed in the next operation and the "Final-Calculation" unit which is responsible for the final computations of each operation.

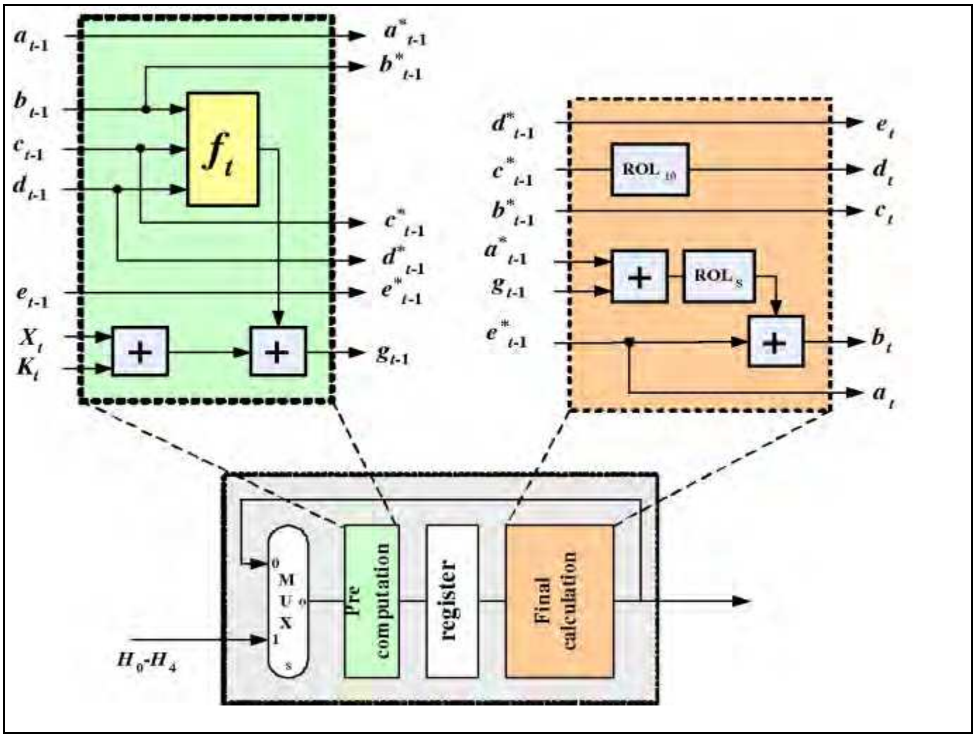

Fig. 3. Spatial pre-computation applied on the operation block of RIPEMD-160

Notice that in Figure 3 output b_t enters the multiplexer and feeds a no-load wire b_{t-1} which stores its value to the register as b_{t-1}*. Also notice at the "Pre-Computation" unit that the inputs a_{t-1}, c_{t-1}, d_{t-1}, e_{t-1}, which is equal with the values a_{t-1}*, c_{t-1}*, d_{t-1}*, e_{t-1}* respectively, are fed through the multiplexer from the intermediate register outputs e_{t-1}*, b_{t-1}*, c_{t-1}*, d_{t-1}* respectively. The introduced area penalty is small, only a single register for each "round", that stores the intermediate value g_{t-1}. In order to reduce the critical path by one addition level, we will continue with the application of the second technique, which introduces a temporal pre-computation of the values.

From the "Final-Calculation" stage of Figure 3, one can observe that in every operation, from the current value of d_{t-1}, derives directly the value of e_t (at the next operation). Also, from the current value of e_t, derives directly the value of a_{t+1}. Consequently, the value of a, is the same as the value of was two operations earlier.

So it is valid to write the following equation:

$$a_{t+1} = e_t = d_{t-1} \quad (3)$$

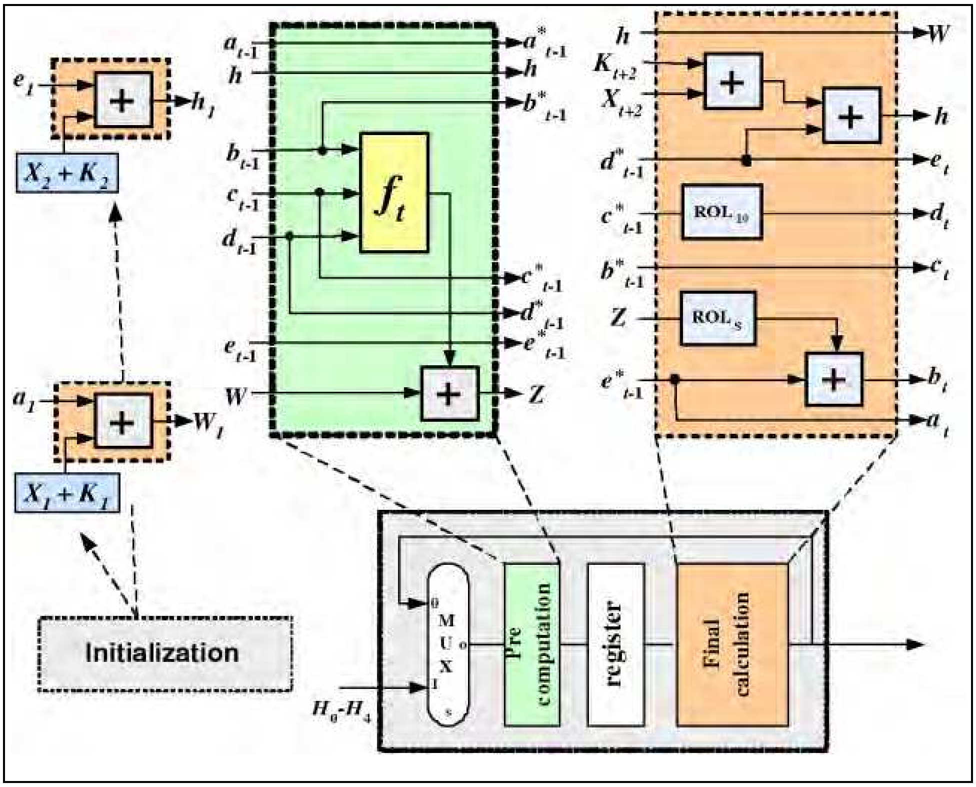

Fig. 4. The proposed RIPEMD-160 operation block after applying temporal transformation

Thus, we perform the temporal pre-computation of the sum $(X_{t+2} + K_{t+2}) + a_{t+1}$ two operations before it is used, by calculating the sum $(X_{t+2} + K_{t+2}) + d_{t-1}$ at the "Final-Calculation" unit, when the operation t is being executed. Then this sum at the "Pre-Computation" stage of the next operation (t+1) saved into the register h and represent the sum $(X_{t+2} + K_{t+2}) + e_t$. At the "Final-Calculation" unit of the same operation, the value of W derives directly from the value of h. The computed sum now of the value W represents the sum $(X_{t+2} + K_{t+2}) + a_{t+1}$.

Finally at the "Pre-Computation" unit on the next operation (which is the operation t+2) the sum Z= W + f_t is calculated. The computed sum now represents the value (X_{t+2} + K_{t+2}) + a_{t+1} + f_t. This sum is part of the computations needed for the calculation of b_{t+2} value. What remains for the computation of the value b_{t+2} is the rotation (Rols) of the value Z and then the addition in this result of the value e_{t+2}, as is performed in the "Final-Calculation" in Figure 4.
Observing Figure 4 we see that the critical path is not located any more in the computation of the b_t value but in the computation of the value of Z. This means that the critical path in Figure 4 has been reduced, from three addition stages, a Non Linear Function f_t and a multiplexer in Figure 1, to two addition stages, a Non Linear Function f_t and multiplexer. Thus, the critical path is shortened by one adder level, which contributes approximately 30% to the overall maximum delay.
However, we must notice that an initialization of the values of W and h is needed as it is illustrated in Figure 4. The introduced area penalty is only two 32-bit registers, which are used for storing the intermediate results of the values W and h that are required. This area penalty sure enough is worth paying for an increase of throughput at about 35%.
The proposed technique to improve the throughput of RIPEMD-160 needs also to be verified. Thus the optimized design of RIPEMD-160 was developed and captured in VHDL and was fully simulated and verified using the Model Technology's ModelSim Simulator. In Table 2, the experimental results are offered. From the experimental results, it was verified that RIPEMD-160 proposed implementation was about 35% faster than the conventional implementation.

Technology	Area (CLBs)	Frequency (MHz)	Throughput t (Gbps)
Virtex	2057	68.2	2.18
Virtex-E	2256	86..6	2.75
Virtex-II	2548	94.6	3.02

Table 2. RIPEMD-160 optimized implementation synthesis result

It is noticed that researchers are moving on to optimize the inner operational block so as to achieve higher throughputs for their implementations. In order to make fair comparisons the proposed RIPEMD-160 implementation was evaluated for a medium performing FPGA technology as it is practically impossible to integrate it in all the necessary technologies. Then it is compared to competitive implementations as they are given in Table 3. In Table 3, reference is given to other implementations of RIPEMD-160 as they were reported in the technical literature by other researchers. These implementations are highly optimized, depending on the design approach the designer has selected to follow.
The implementations of (Bosselaers et al., 1996), (Ng et al., 2004) present low throughput as they are software based. The highest throughput is reported by (Sklavos & Koufopavlou, 2005), which achieves a throughput of 2 Gbps. This is about 35% less than the throughput that is achieved for the implementation of RIPEMD-160 that is presented in this work. Furthermore, comparing the implementations of other researchers to the proposed one, it

can be observed that all of them fall short in throughput, in a range that varies from 2 – 41 times less than the proposed implementation. In the following Table, it is worth mentioning that the proposed implementation outperform even ASIC implementations, exhibiting the collateral benefit of the application of the proposed design technique.

Implementation	Technology	Frequency (MHz)	Throughput (Mbps)
Proposed	FPGA Virtex xcv300E	86.7	2752
(Bosselaers et al., 1996)	Software (Pentium)	90	45.5
(Ng et al., 2004)	FPGA Altera	26.6	84
(Sklavos & Koufopavlou, 2003)	FPGA Virtex xcv300E	65	2000
(Satoh & Inoue, 2005)	ASIC (0.13 um CMOS)	-	1442
(Dominikus, 2002)	ASIC (0.6 um)	-	89
(Dominikus, 2002)	FPGA Virtex xcv300E	42.9	65
(Khan et al., 2005)	FPGA Virtex xcv300E	37	117

Table 3. Comparison of the proposed implementation of RIPEMD-160 with the competition

Finally, it also has to be mentioned that more implementations exist but with much lower achieved throughput and most of them are outdated. These were not considered for comparisons since it would be rather unfair not only because these were the first attempts to optimize hash functions but also because at that time the existing implementation tools were not that sophisticated like the ones that exist nowadays. It should also be pointed out that there are some implementations that focus on area requirements, without applying any pipeline stage. In these implementations effort has been paid only for minimizing area. Those implementations are designated for use in mobile and portable devices where area requirements and power dissipation is the major factor. Throughput is not such a dramatic factor in these cases because the clients can wait a few seconds for the transaction to be carried out successfully

5.3 HMAC mechanism design architecture

The scope of an HMAC implementation is to authenticate both the source of a message and its integrity without the use of any additional mechanisms. HMACs have two functionally

distinct parameters, a message input and a secret key known only to the message originator and intended receiver(s). HMAC is used not as a cipher, but rather as a mechanism for signing a packet with a key at one end of the connection, and then verifying the signature at the other end using the same key. Without the key it is infeasible to generate a packet with the correct signature.
HMAC mechanism is mainly consisted of the utilized hash function which is the factor that determines the throughput and the operating frequency of the HMAC mechanism. So it can be assumed that implementing hashing cores with very high throughput will result in high throughput implementation of HMAC mechanism which will in his turn result in high throughput implementation of the whole security scheme, whish is essential in IPSec, SSL/TLS and elsewhere. In this work the usage of the RIPEMD-160 hash function is adopted for reasons that have already been presented, presenting the alternative HMAC-RIPEMD-160 implemntation.
In this section the HMAC hardware architecture is going to be presented incorporating the optimized version of RIPEMD-160 hash function that has already been presented .The whole HMAC system architecture is illustrated in Figure 5 according to the standard that describes its functionality. As it can be seen two hashing cores have to be incorporated in the HMAC imlementations regardless of the selected hash function. So, the architecture in Figure 5 is quite generic and another hashing function can also be incorporated. In this work two pipelined implementations of RIPEMD-160 hasing core with optimized operation block is going to be used for the implementation of the whole HMAC mechanism, thus resulting to a pipelined HMAC implementation.
Analyzing the illstrated system design in Figure 5, we should begin from the the MS RAM where all message schedules Wt of the padded message are stored. The Constants Array is a hardwired array that provides the constant values Kt and the constant initialization values. Additionally, it includes the Wt generators and the Key Generation unit that is required at the HMAC mechanism.
As it was previously mentioned this components are essential for the hashing cores to function properly. The HMAC must firstly be initialized before processing any message. The initialization procedure corresponds to computing the hash values of two certain 512-bit blocks, which are the corresponding keys, and this is performed independently in the two RIPEMD-160 cores. When the initialization procedure is completed, the hash values from the outputs of the two RIPEMD-160 cores are stored in the Constants Array unit and are then used continuously in the two RIPEMD-160 cores are the new initial values. These values as well as the corresponding keys shall be protected and this is the reason why they are stored in registers in the Constants Array unit and not in the RAM memory which is considered as vulnerable.
This is the first time that a message can be supplied for process to the first RIPEMD-160 core as long as it has been padded, while the initialization procedure was in progress. After being processed in the first RIPEMD-160 core the message will be padded and forwarded for process in the second RIPEMD-160 core that exists in the HMAC mechanism and is illustrated in Figure 5 with the notation “HASHING CORE”. As it has already been explained l, in the proposed hardware implementation of RIPEMD-160 five pipeline stages have been applied.This means that in the presented implementation ten pipeline registers are available and thus ten different messages can be concurrently processed.
In the proposed implementation each hmac value is computed after 161 clock cycles (80 for

each one of the two RIPEMD-160 cores and one clock cycle for the intermediate padding-register).The method of intermediate storing the values that have arise from the processing of the two keys at the RIPEMD-160 cores, saves the time of processing two 512-bit blocks for every message and also allows pipeline technique to be applied, increasing this way both speed and throughput of the HMAC implementation. These stored intermediate values shall be treated and protected in the same way as the secret keys.

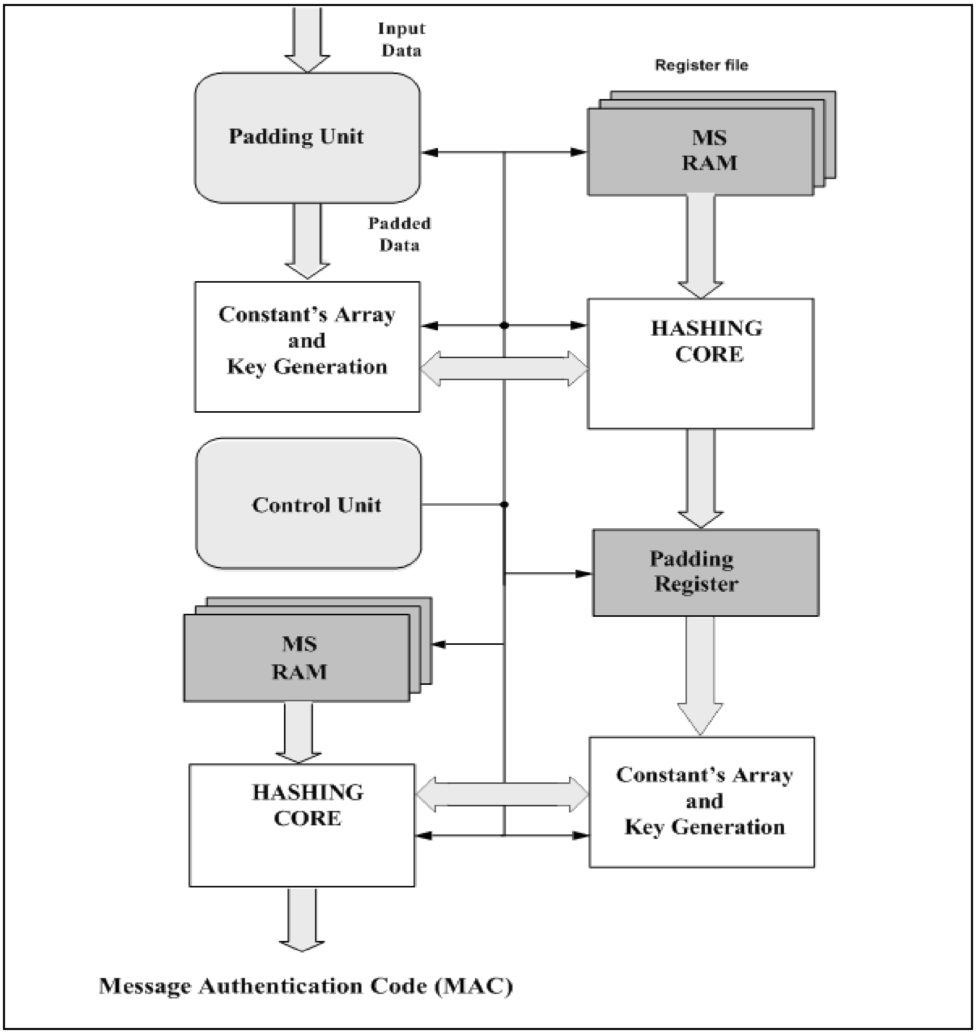

Fig. 5. HMAC system architecture

6. CMAC: A Reliable Alternative

In previous sections we have already outlined the importance of CMAC as an alternative way to provide authentication services in certain applications. This necessity mainly has to do with the numerous applications that do not call for high throughput implementations (as long as the security part is concerned) but for small-sized or low-power implementations. In these cases the usage of CMAC instead of HMAC is ideal since there is no need to incorporate both a hash function and a cipher block but only a block cipher algorithm like AES that can handle all aspects of security in the certain security scheme. In this section implementation details concerning CMAC will be provided.

It has to be noticed that, to the best of our knowledge, there is no other hardware implementations concerning CMAC (CMAC neural network has no relation to CMAC for producing MACs for integrity). This is partially due to the fact that this algorithm was recently proposed and industry has not yet introduced to the market related hardware products. However some software CMAC implementations do exist but obviously these can be neither commented nor compared to our work which proposes a hardware implementation.

The CMAC algorithm incorporates a symmetric key block cipher like AES (Advanced Encryption Standard), TDEA or any other FIPS (Federal Information Processing standards) approved block cipher algorithm. Block ciphers consist of two operations one for encryption and one for decryption. However in the implementation of the CMAC algorithm only one of these operations is needed, and usually the encryption process is selected. The cipher encryption process is a permutation on bit strings of a fixed length; the strings are called blocks and their size varies depending on the used block cipher. For the AES it is 128 bits whereas for TDEA is 64 bits.The CMAC algorithm implies the usage of a secret key which is the block cipher key. The corresponding key shall be protected for its secrecy and used exclusively for the CMAC mode of the chosen block cipher.

In (NIST, 2005 b), the authentication procedure using a cipher block is described. There all the prerequisites are mentioned and thoroughly described. This process has some similarities with the procedure that is followed in order to produce a MAC using a hash function.The sender produces the MAC using the eight steps described in (NIST, 2005 b) and thus producing the CMAC that is the MAC with the usage of a cipher block like AES. The CMAC standard implies the usage of a FIPS approved cipher block and the most widely currently used block cipher algorithm is the AES which is considered as absolutely secure. For implementation's shake this block cipher was designed in order to be used in the whole CMAC structure according to its standard without any sophisticated optimization technique. However the implementation of the block cipher will not be presented in this chapter since our intention is to provide an abstract knowledge about an alternative way to offer authenticating servises, thus we shall focus on CMAC and not on AES.

However it has to be stressed that only the encryption process was developed since the CMAC uses either the encryption or the decryption process of the selected block cipher algorithm. However in a real's world security scheme both processes exist for encrypting and decrypting the exchanged data between the communicating parts and at the computation of a CMAC value simply only the one process is used. Other block cipher algorithms that can be used in the CMAC mechanism are all FIPS approved block ciphers such as TDEA. However we decided to use the encryption process of the AES block cipher because it is the most widespread worldwide.

As it will be referred the critical path of the whole CMAC mechanism is located at the AES core and so at a later time effort can be paid in optimizing the used cipher block (i.e AES) and thus optimize the CMAC mechanism. In the implementation scheme in this section the block cipher algorithm that has been designed and implemented is noted as "Block Cipher". This is illustrated in Figure 6 where the whole CMAC architecture is outlined.

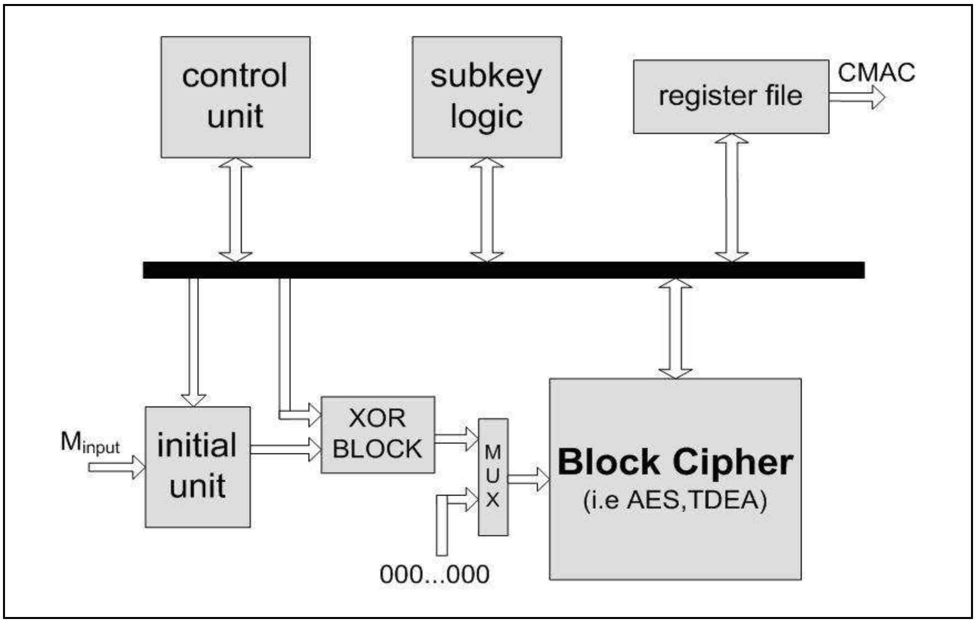

Fig. 6. CMAC Implementation

Any FIPS approved cipher block algorithm can be used instead of AES and TDEA in the same way as it is shown in the previous figure. The interface for the communication of the different components will remain the same. The CMAC implementation consists of several parts.

The overlying software-based system feeds the CMAC scheme (implemented in hardware) with the properly padded (when needed) input blocks M_{input} which are parts of the whole message that is intended to be protected for authentication. The padding process is a quite simple process that can be carried out by the software. This does not imply security problems since there is no secrecy or keys information that should be protected at this point.

The initial unit part is the hardware component responsible for the conditional bitwise operation xor between the M_{input} and one of the subkeys K_1 or K_2.

This part must be implemented in hardware although it could easily be carried out in software and is illustrated at the left in Figure 7. The necessity for hardware implementation of the specific parts arises from the fact that in this unit certain operations concerning the two subkeys exist. If this process is carried out in software then the whole security system is vulnerable since the two subkeys can easily be retrieved. Hardware implementation increases dramatically security since the K_1 and K_2 values are stored in registers. Furthermore in order to ensure an even more higher level of security in our implementation

the key K and the two subkeys K_1 and K_2 are not stored in a RAM memory but in shadow registers. This way we avoid the danger someone who manages to access and read the security system's memory (RAM) to reveal our keys and subkeys that are stored there.
At the initialization phase the two subkeys K_1 and K_2 must be computed from the main key K. This procedure is carried out by the hardware component "subkey logic" that is illustrated at the right in Figure 7. This component loads the key K from the register file and an already available output of the used block cipher algorithm and in two clock cycles it computes the subkeys K_1 and K_2 that are then saved in the register file.

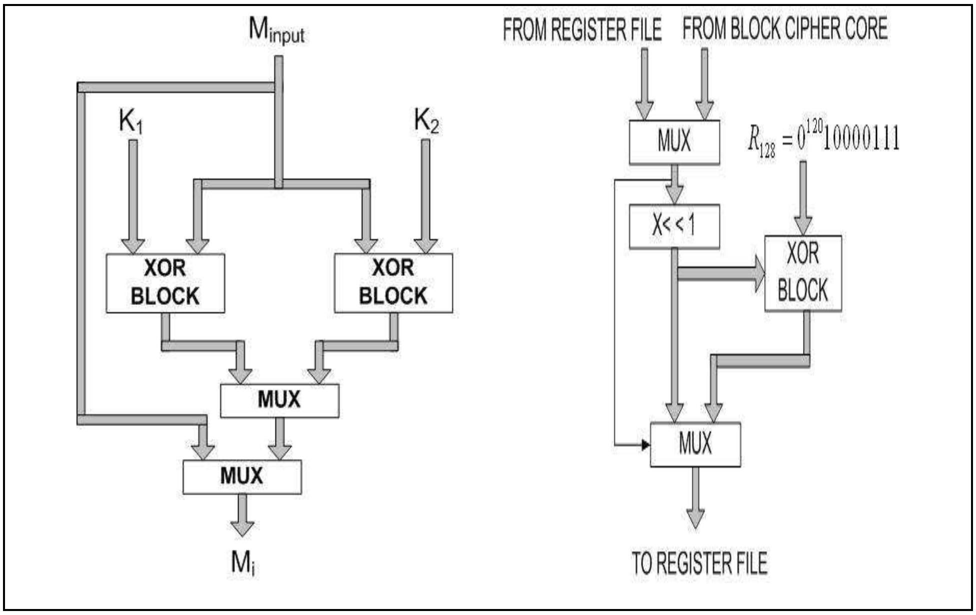

Fig. 7. Units "Initial Unit" (left) and "Subkey Logic"(right)

The selected output from the cipher block is the so before K_1 and K_2 computations take place the must be first computed. Thus this component is enabled only after has been computed and is disabled after two clock cycles resulting to a low power operation. This component is enabled again only if the used main key K has changed for some reason.
The "XOR BLOCK" component is a simple component performing the bitwise xor operation. The "Register File" consists of some registers that are used for storing keys, subkeys, the final CMAC value as well as some useful intermediate values.
Finally the control unit consists of some small counters and is responsible for the correct synchronization of all the components that exist in the CMAC scheme. It is also responsible for the enabling of the several components just when they are needed and disabling them just after that.
So incorporating any FIPS approved secure block cipher algorithm in the presented design will result in an efficient implementation of CMAC mechanism. It can be seen that CMAC is simpler, smaller and thus less costly comparing to HMAC. So it is obvious that in cases that were described and are eligible to incorporate CMAC, this selection is advantageous.

7. Conclusion

In this chapter two ways of providing authentication services (HMAC and CMAC) have been presented. Certain applications' criteria that have to be taken into consideration to choose between CMAC and HMAC were stated. Implementation details and system architecture were provided for both alternatives focusing on the manageability of these designs so as to be easy to incorporate any secure hash function in case of HMAC or any secure block cipher algorithm in case of CMAC. We focused on the design and implementation of RIPEMD-160 hash function that can be used for authentication in an HMAC mechanism and two techniques were presented that resulted to the optimization of the RIPEMD-160 hash function by 35% in throughput with a minor area penalty.

Acknowledgement

This work was partially supported by the project PENED 2003 No 03ED507, which is funded in 80% by the European Union- European Social fund and in 20% by the Greek state-Greek Secretariat for Research and Technology.

8. References

Ballard, K. (2004). http://digestit.ken-nethballard.com/blog/index.php?/archives/P2.html

Biever C. (2004). US boosts e-voting software security, NewScientist.com news, *http://www.newscientist.com/article.ns?id=dn6593*

Bosselaers, A., Govaerts, R. and Vandewalle, J. (1996). Fast hashing on the Pentium, Proceedings of the Advances in Cryptology (Crypto1996), pp. 298-312

Dobbertin H., Bosselaers A., Preneel B. (1996). RIPEMD-160: A Strengthened Version of RIPEMD

Dominikus, S. (2002). A Hardware Implementation of MD4-Family Hash Algorithms. Proceedings of 9th IEEE International Conference on Electronics, Circuits and Systems (ICECS 2002), Vol. III, pp. 1143-1146, Sep. 2002.

Hodjat A. and Verbauwhede I., A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA. IEEE Symposium on Field-Programmable Custom Computing Machines Systems, (FCCM '04) pp. 308-309, 2004

Johnston D. and Walker J.,(2004). Overview of IEEE802.16 Security, IEEE Security and Privacy

Khan, E., El-Kharashi, M.W., Gebali, F. and Abd-El-Barr, M. (2005). An FPGA Design of a Unified Hash Engine for IPSec Authentication. Proceedings of the Fifth international Workshop on System-on-Chip For Real-Time Applications (IWSOC2005), pp. 450-453

Kim G.H. and Spafford E.H., (1994). The Design and Implementation of Tripwire: A File System Integrity Checker, 2nd ACM Conf. Computer and Communications Security, pp. 18-29

Loeb L. (1998). Secure Electronic Transactions: Introduction and Technical Reference, Artech House Publishers

McCurley K.S. (1994). A Fast Portable Implementation of the Secure Hash Algorithm, Sandia National Laboratories Technical Report SAND93–2591.

Michail H.E, Kakarountas A. P., Milidonis A. S. and Goutis C. E., (2008). A Top-Down Design Methodology for Implementing Ultra High-Speed Hashing Cores, IEEE Transactions on Dependable and Secure Computing , accepted for publication

Microsoft, (2005). Internet Explorer 6: Digital Certificates, *http://www.microsoft.com/windows/ie/ie6/using/howto/security/digitalcert/using.mspx*.

Mironov I. (2005), Hash functions: "Theory, attacks, and applications", Microsoft Research, Silicon Valley Campus

Morris R. and Thompson K. (1979)., Password security: A case history, Communications of ACM, vol. 22(11), pp. 594-597

Nakajima, J. and Matsui, M. (2002). Performance Analysis and Parallel Implementation of Dedicated Hash Functions, Lecture Notes in Computer Science (LNCS), vol. 2332, pp. 165–180. Springer.

Ng, C.-W., Ng, T.-S. and Yip, K.-W. (2004) .A unified architecture of MD5 and RIPEMD-160 hash algorithms. Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2004). Vol. 2, pp. 889-892.

NIST, FIPS 186, (DSS), (1994). Digital Signature Standard (DSS) , US Dept of Commerce

NIST, (2001) a. Advanced Encryption Standard (AES) Home Page , US Dept of Commerce http://csrc.nist.gov/encryption/aes

NIST, SP 800-32., (2001) b. Introduction to Public Key Technology and the Federal PKI Infrastructure, US Dept of Commerce

NIST, SP800-77, (2005) a .Guide to IPSec VPN's, US Dept of Commerce

NIST, (2005) b. Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication , US Dept of Commerce http://csrc.nist.gov/publications/nistpubs/ index.html#sp800-38B1998

NIST, FIPS 198-1, (2007) a. The Keyed-Hash Message Authentication Code (HMAC), US Dept of Commerce

NIST, (2007) b. XTS-AES, IEEE 1609 approved, under consideration from NIST for FIPS 140-2 , US Dept of Commerce http://grouper.ieee.org/groups/1619tmp/1619-2007-NIST-Submission.pdf

NIST, (2008). Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, FIPS PUB 800-38D, US Dept of Commerce http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf

Satoh, A. and Inoue, T. (2005). ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS. Proceedings of International Conference on Information Technology: Coding and Computing (ITCC2005), Vol. I, pp. 532-537.

Schneier, B. (1996). Applied Cryptography – Protocols, Algorithms and Source Code in C, Second Edition, John Wiley and Sons.

Sklavos, N. and Koufopavlou, O. (2005). On the hardware implementation of RIPEMD processor: Networking high speed hashing, up to 2 Gbps, Computers & Electrical Engineering, Vol. 31, Is. 6, pp. 361-379.

Stephen T., (2000). SSL & TLS Essentials: Securing the Web, John Wiley and sons Publications

Van Oorschot, P.C., Somayaji, A. and Wurster, G. (2005). Hardware-Assisted Circumvention of Self-Hashing Software Tamper Resistance. IEEE Transactions on Dependable and Secure Computing, Vol. 02, No. 2, pp. 82-92.