Reprint

Efficient Architecture and Hardware Implementation of the Whirlpool Hash Function

P. Kitsos and O. Koufopavlou

IEEE Transactions on Consumer Electronics

Vol. 50, Issue 1, February 2004, pp. 208-213

Efficient Architecture and Hardware Implementation of the Whirlpool Hash Function

P. Kitsos and O. Koufopavlou, Member, IEEE

Abstract — *The latest cryptographical applications demand both high speed and high security. In this paper, an architecture and VLSI implementation of the newest powerful standard in the hash families, Whirlpool, is presented. It reduces the required hardware resources and achieves high-speed performance. The architecture permits a wide variety of implementation tradeoffs. The implementation is examined and compared in the security level and in the performance by using hardware terms. This is the first Whirlpool implementation allowing fast execution, and effective substitution of any previous hash families' implementations such as MD5, RIPEMD-160, SHA-1, SHA-2 etc, in any cryptography application*[1].

Index Terms — Cryptography, NESSIE, Pipelined architectures, Whirlpool Hash function.

1. Introduction

In recent years the demands for effective and secure communications in both wire and wireless networks is especially noted in the consumer electronics area. In modern consumer electronics, security applications play a very important role. The interest in financial and other electronic transactions is grown; so the security applications can provide an important way for consumers and businesses to decide which electronic communications they can trust.

A hash function is a function that maps an input of arbitrary length into a fixed number of output bits, the hash value. Hash functions are used as building blocks in various cryptographic applications. The most important uses are in the protection of information authentication and as a tool for digital signature schemes.

The most known hash function is the Secure Hash Algorithm-1 (SHA-1) [1]. The security parameter of SHA-1 was chosen in such a way to guarantee the similar level of security, in the range of 2^{80} operations, as required by the best currently known attacks. But, the security level of SHA-1 does not match the security guaranteed by the new announced AES encryption standard [2], which is specified 128-, 192-, and 256-bit keys.

Many attempts have been taken place in order to put forward new hash functions and match the security level with the new encryption standard. In August 2002, National Institute of Standards and Technology (NIST) announced the updated Federal Information Processing Standard (FIPS 180-2) [3], which introduced three new hash functions referred to as SHA-2 (256, 384, 512). In addition, the New European Schemes for Signatures, Integrity, and Encryption (NESSIE) project [4], was responsible to introduce a hash function with similar security level. In February 2003, it was announced that the hash function included in the NESSIE portfolio is Whirlpool [5]. All the above-mentioned hash functions are adopted by the International Organization for Standardization (ISO/IEC) 10118-3 standard [6].

Whirlpool hash function is byte-oriented and consists of the iterative application of a compression function. This is based on an underlying dedicated 512-bit block cipher that uses a 512-bit key and runs in 10 rounds in order to produce a hash value of 512 bits.

In this paper, an architecture and VLSI implementation of the new hash function, Whirlpool, is proposed. It reduces the required hardware resources and achieves high-speed performance. The proposed implementation is examined and compared, in the offered security level and in the performance by using hardware terms. In addition, due to no others Whirlpool implementations existence, comparisons with other hash families' implementations [7]-[13] are provided. From the comparison results it is proven that the proposed implementation performs better and composes an effective substitution of any previous hash families' such as MD5, RIPEMD-160, SHA-1, SHA-2 etc, in almost all the cases.

The organization of the paper is as follows: In section 2, the Whirlpool hash function is described. In section 3, the proposed architecture is presented. Performance analysis and comparison results with other works are reported in section 4. Finally, concluding remarks and some extensions are made in section 5.

2. The Whirlpool Hash Function

Whirlpool is a one-way, collision resistant 512-bit hash function operating on messages less than 2^{256} bits in length. It consists of the iterated application of a compression function, based on an underlying dedicated 512-bit block cipher that uses a 512-bit key. The Whirlpool is a Merkle hash function [14] based on dedicated block cipher, W, which operates on a 512-bit hash state using a chained key state, both derived from the input data. The round function and the key schedule, of the W, are designed according to the Wide Trail strategy [15].

In the following, the round function of the block cipher, W, is defined, and then the complete hash function is specified.

[1] P. Kitsos is with the VLSI Design Laboratory, Department of Electrical and Computer Engineering, University of Patras, 26500 Patras, Greece (e-mail: pkitsos@ee.upatras.gr).

O. Koufopavlou is with the VLSI Design Laboratory, Department of Electrical and Computer Engineering, University of Patras, 26500 Patras, Greece (e-mail: odysseas@ee.upatras.gr).

The block diagram of the *W* block cipher basic round is shown in Fig. 1.

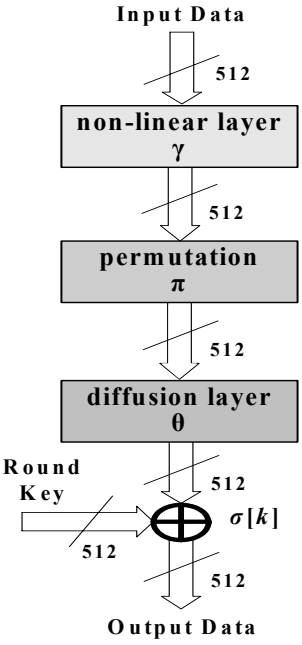

Fig. 1. Block diagram of the W basic round

The round function, $\rho[k]$, is based on combined operations from three algebraic functions. These functions are the non-linear layer γ, the cyclical permutation π, and the linear diffusion layer θ. So, the round function is the composite mapping $\rho[k]$, parameterized by the key matrix k, and given by:

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \pi \circ \gamma \tag{1}$$

Symbol "$\circ$" denotes the sequential (associative) operation of each algebraic function where the right function is executed first. The key addition $\sigma[k]$, consists of the bitwise addition (exor) of a key matrix k such as:

$$\sigma[k](a) = b \Leftrightarrow b_{ij} = a_{ij} \oplus k_{ij}, 0 \le i, j \le 7 \tag{2}$$

This mapping is also used to introduce round constants in the key schedule. The input data (hash state) is internally viewed as a 8x8 matrix over $GF(2^8)$. Therefore, 512-bit data string must be mapped to and from this matrix format. This is done by function μ such as:

$$\mu(a) = b \Leftrightarrow b_{ij} = a_{8i+j}, 0 \le i, j \le 7 \tag{3}$$

The first transformation of the hash state is through the non-linear layer γ, which consists of the parallel application of a non-linear substitution *S*-Box to all bytes of the argument individually. After, the hash state is passed through the permutation π that cyclical shifts each column of its argument independently, so that column j is shifted downwards by j positions. The final transformation is the linear diffusion layer θ, which the hash state is multiplied with a generator matrix. The effect of θ is the mix of the bytes in each state row.

So, the dedicated 512-bit block cipher $W[K]$, parameterized by the 512-bit cipher key K, is defined as:

$$W[K] = \left(O_1^{r=R} \rho[\mathrm{K}^r]\right) \circ \sigma[\mathrm{K}^0] \tag{4}$$

where, the round keys $K^0,\ldots,K^R$ are derived from K by the key schedule. The default number of rounds is R=10. The key schedule expands the 512-bit cipher key K onto a sequence of round keys $K^0,\ldots,K^R$ as:

$$\begin{aligned} K^0 &= K \\ K^r &= \rho[c^r](K^{r-1}), r > 0 \end{aligned} \tag{5}$$

The round constant for the r-th round, r>0, is a matrix c^r defined by substitution box (*S-Box)* as:

$$\begin{aligned} c_{oj}^r &\equiv S[8(r-1)+j], 0 \le j \le 7, \\ c_{ij}^r &\equiv 0, \qquad 1 \le i \le 7, 0 \le j \le 7 \end{aligned} \tag{6}$$

So, the Whirlpool iterates the Miyaguachi-Preneel hashing scheme [14] over the t padded blocks m_i, $1 \le i \le t$, using the dedicated 512-bit block cipher W:

$$\begin{aligned} n_i &= \mu(m_i), \\ H_0 &= \mu(IV), \\ H_i &= W[H_{i-1}](n_i) \oplus H_{i-1} \oplus n_i, 1 \le i \le t \end{aligned} \tag{7}$$

where, IV (the Initialization Vector) is a string of 512 0-bits.

As (4) and (5) shows the internal block cipher *W,* comprises of a data randomizing part and a key schedule part. These parts consist of the same round function.

Before being subjected to the hashing operation, a message M of bit length $L<2^{256}$ is padded with a 1-bit, then as few 0-bits as necessary to obtain a bit string whose length is an odd multiple of 256, and finally with the 256-bit right-justified binary representation of L, resulting in the padded message m, partitioned in t blocks $m_1, m_2, \ldots, m_t$.

3. HARDWARE ARCHITECTURE AND VLSI IMPLEMENTATION

The architecture that performs the Whirlpool hash function is shown in Fig. 2.

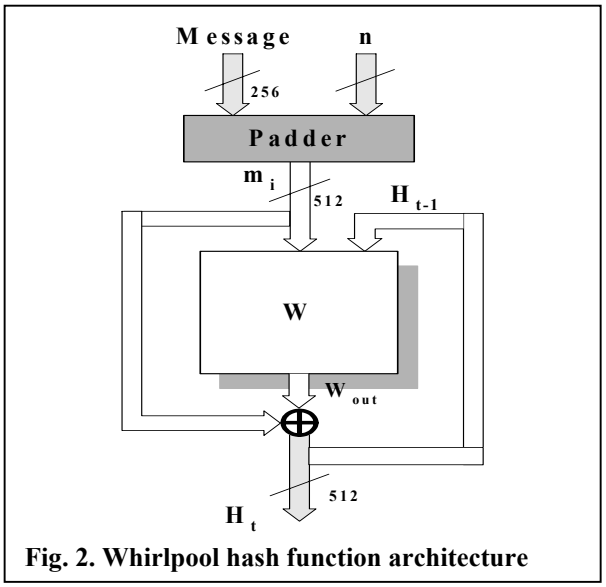

Fig. 2. Whirlpool hash function architecture

The *Padder* pads the input data and converts them to n-bit padded message. In the proposed architecture an interface with 256-bit input for *Message* is considered. The input n, specifies the total length of the message. The padded message

is partitioned into a sequence of t 512-bit blocks m_1, m_2, … , m_t. This sequence is then used in order to generate a new sequence of 512-bit string, H_1, H_2, … , H_t in the following way. m_i is processed with H_{i-1} as key, and the resulting string is XORed with m_i in order to produce the H_i. H_0 is a string of 512 0-bits and H_t is the hash value.

The block cipher W, is mainly consists of the round function ρ. The implementation of the round function ρ is illustrated in Fig. 3.

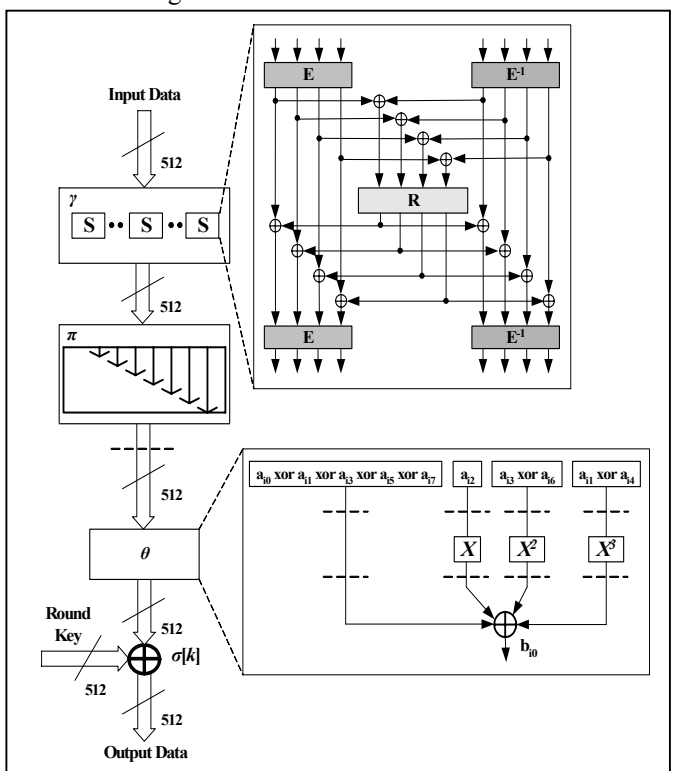

Fig. 3. Implementation of the round function ρ

The non-linear layer γ, is composed of 64 substitution tables (S-Boxes). The internal structure of the S-Box is shown in Fig. 3. It consists of five 4-bit mini boxes E, E^{-1}, and R. These mini boxes can be implemented either by using Look-Up-Tables (LUTs) or Boolean expressions.

Next, the cyclical permutation π, is implemented by using combinational shifters. These shifters are cyclically shift (in downwards) each matrix column by a fixed number (equal to j), in one clock cycle.

The linear diffusion layer θ, is a matrix multiplication between the hash state and a generator matrix. In [5] a pseudo-code is provided in order to implement the matrix multiplication. However, in this paper an alternative way is proposed which is suitable for hardware implementation.

The transformation expressions of the diffusion layer are given below (equation (8)). Bytes b_{i0}, b_{i1}, … , b_{i7} represent the eight bytes of the i row of the output of the layer θ hash state. Table X implements the multiplication by the polynomial $g(x)$=x modulo ($x^8+x^4+x^3+x^2+1$) in $GF(2^8)$ (i.e. $X[u] \equiv x * u$, where u denote the input of the table). Table X^2 is defined as $X^2 \equiv X \circ X$ and X^3 as $X^3 \equiv X \circ X \circ X$.

$$
\begin{aligned}
b_{i0} &= a_{i0} \oplus a_{i1} \oplus a_{i3} \oplus a_{i5} \oplus a_{i7} \oplus X[a_{i2}] \oplus X^2[a_{i3} \oplus a_{i6}] \oplus X^3[a_{i1} \oplus a_{i4}] \\
b_{i1} &= a_{i0} \oplus a_{i1} \oplus a_{i2} \oplus a_{i4} \oplus a_{i6} \oplus X[a_{i3}] \oplus X^2[a_{i4} \oplus a_{i7}] \oplus X^3[a_{i2} \oplus a_{i5}] \\
b_{i2} &= a_{i1} \oplus a_{i2} \oplus a_{i3} \oplus a_{i5} \oplus a_{i7} \oplus X[a_{i4}] \oplus X^2[a_{i5} \oplus a_{i0}] \oplus X^3[a_{i3} \oplus a_{i6}] \\
b_{i3} &= a_{i0} \oplus a_{i2} \oplus a_{i3} \oplus a_{i4} \oplus a_{i6} \oplus X[a_{i5}] \oplus X^2[a_{i6} \oplus a_{i1}] \oplus X^3[a_{i4} \oplus a_{i7}] \\
b_{i4} &= a_{i1} \oplus a_{i3} \oplus a_{i4} \oplus a_{i5} \oplus a_{i7} \oplus X[a_{i6}] \oplus X^2[a_{i7} \oplus a_{i2}] \oplus X^3[a_{i5} \oplus a_{i0}] \\
b_{i5} &= a_{i0} \oplus a_{i2} \oplus a_{i4} \oplus a_{i5} \oplus a_{i6} \oplus X[a_{i7}] \oplus X^2[a_{i0} \oplus a_{i3}] \oplus X^3[a_{i6} \oplus a_{i1}] \\
b_{i6} &= a_{i1} \oplus a_{i3} \oplus a_{i5} \oplus a_{i6} \oplus a_{i7} \oplus X[a_{i0}] \oplus X^2[a_{i1} \oplus a_{i4}] \oplus X^3[a_{i7} \oplus a_{i2}] \\
b_{i7} &= a_{i0} \oplus a_{i2} \oplus a_{i4} \oplus a_{i6} \oplus a_{i7} \oplus X[a_{i1}] \oplus X^2[a_{i2} \oplus a_{i5}] \oplus X^3[a_{i0} \oplus a_{i3}]
\end{aligned}
\tag{8}
$$

In Fig. 3, the implementation of the output byte b_{i0} is depicted in details. The other bytes are implemented in a similar way.

The key addition ($\sigma[k]$) consists of eight 2-input XOR gates for any byte of the hash state. Every bit of the round key is XORed with the appropriate bit of the hash state.

A first (1st) implementation of the W block cipher architecture is shown in Fig. 4. This implementation is suitable for applications with constrained silicon area resources. The implementation details of the non-linear layer γ, the cyclical permutation π, and the linear diffusion layer θ are shown in Fig. 3. In order to reduce the required hardware resources, the appropriate key schedule part is integrated with the data randomizing part.

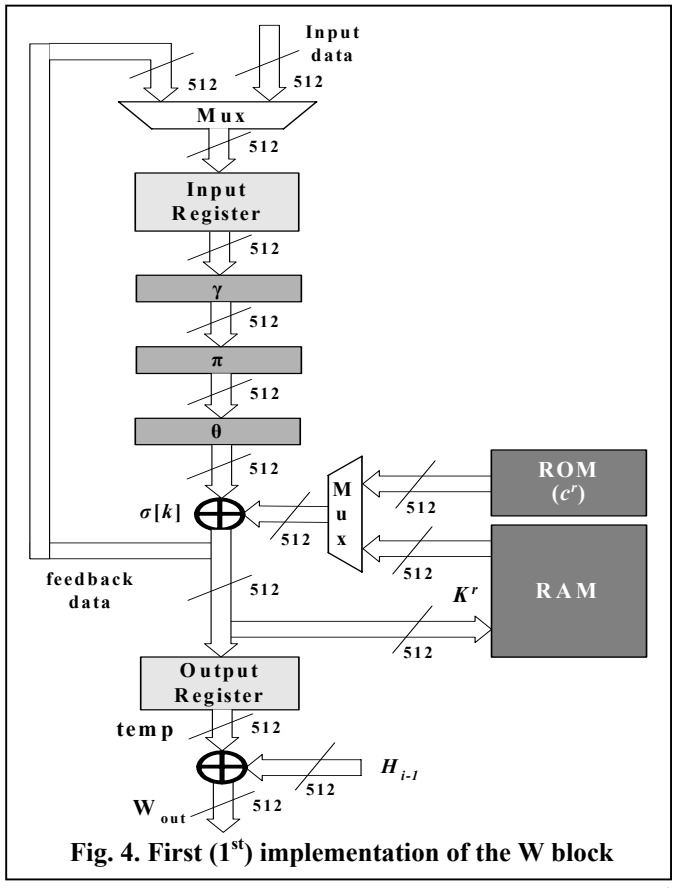

Fig. 4. First (1st) implementation of the W block

The execution of the W block cipher on the 1st implementation is performed in two phases. In the first phase, the round keys are produced and stored in the RAM. In the

second phase, the hash value is computed. The algorithm specifies 10 rounds for the hash state. In order to produce the round keys (first phase), the *Input data* is the Initialization Vector (*IV*). The *Input Register* is used in order to buffer the algorithm *Input data*. The output data of the θ layer is bitwise XORed with the c^r constant. Each execution round lasts one clock cycle.

After the first execution round, the first round key is stored in the RAM. It is used as input in the second execution round, through the multiplexer (*feedback data*), for the production of the second round key. This process is repeated 10 times (10 execution rounds) and lasts 10 clock cycles. The c^r constants are predefined and stored in the ROM. The multiplexer selects during the first phase the c^r constants, and during the second phase the round keys.

The computation of the hash value is take place during the second phase. In this phase the *Input data* are the m_i blocks. The output data of the θ layer is bitwise XORed with the appropriate round key, which is stored in the RAM. After 10 execution rounds the *Output Register* latches the *temp* result. This result is bitwise XORed with the H_{i-1} value (in this case is equal to the Initialization Vector (*IV*)) in order to compute the W_{out}. The W_{out} is XORed with the m_i (see Fig. 2), so the final, *hash value* H_i, is computed.

If another block m_{i+1} is required to be transformed, the previous process is repeated (by using as cipher key the H_i value). So, for t blocks the execution time is $20*t$ clock cycles.

For applications with high throughput requirement a second (2^{nd}) implementation is proposed. This implementation is an extension of the 1^{st} implementation. In order to compute the round keys on the fly, the RAM is replaced by the key schedule datapath. In this implementation the total execution time of each block is 10 clock cycles, and so, the throughput is doubled.

The proposed 2^{nd} implementation is shown in Fig. 5. This implementation has two similar parallel datapaths, the data randomizing and the key schedule. The input block m_i is set to the *Input data* simultaneously with the initial vector (*IV*) to the *Key*. In the key schedule datapath, the output data of the θ layer is bitwise XORed with the c^r constant. A round key is produced in one clock cycle. Each produced round key is used in the next clock cycle (through the multiplexer) for the production of the next round key. In the data randomizing datapath, the hash state of the θ layer is bitwise XORed with the appropriate round key. After, the intermediate *feedback data* are used as input to the next round (through the multiplexer). After 10 execution rounds the *Output Register* latches the *temp* value. This is bitwise XORed with the H_{i-1} value in order to compute the W_{out}.

In a clock cycle, one execution round is executed and, simultaneously, the appropriate round key is calculated. The system needs 10 clock cycles per block. For t blocks the execution time is $10*t$ clock cycles. So, the total throughput of the 2^{nd} implementation is double than the 1^{st} implementation's but it needs almost double silicon area.

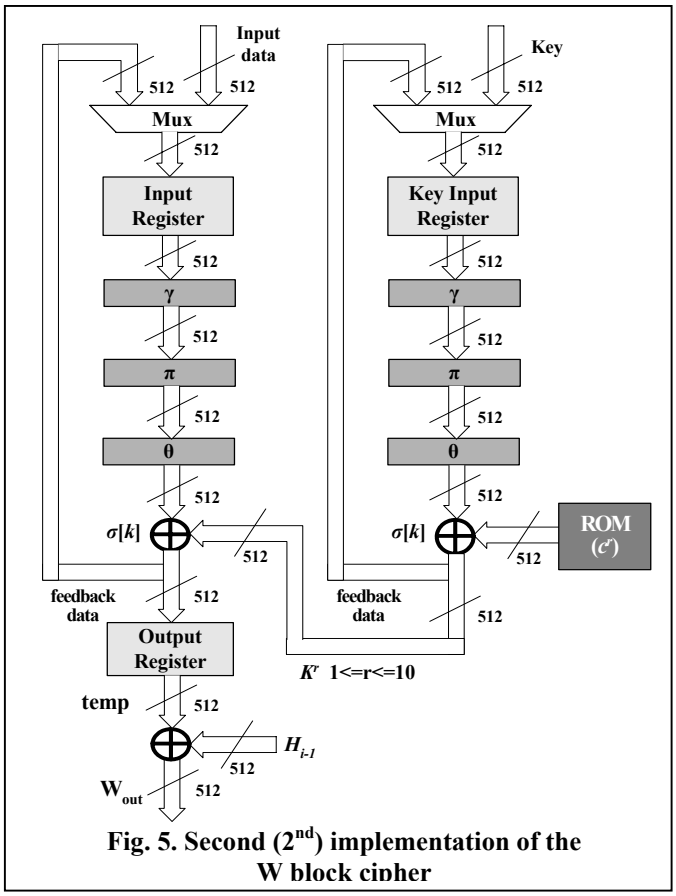

Fig. 5. Second (2^{nd}) implementation of the W block cipher

4. Performance Analysis And Comparison Results

Each one of the proposed implementations was captured by using VHDL, with structural description logic. Both implementations were simulated for the correct operation by using the test vectors provided by, first the NESSIE submission package [5], and second the ISO/IEC 10118-3 standard [6]. The VHDL codes of the two designs were synthesized for FPGA devices. The two implementations then were simulated again for the verification of the correct functionality in real time operating conditions. In addition, parts of the proposed implementations were designed by using two alternative techniques. The 4-bit mini boxes (E, E^{-1}, and R) were designed by using Look Up Tables (LUT) and Boolean expressions. So, for the implementation of the Whirlpool hash function four alternative solutions are proposed.

Someone could claim that the more efficient way in order to implement the mini boxes is by using ROM (internal or external to the FPGA) instead Function Generators (FPGA-LUTs). But, our study indicates that for 4-bit width ROM it is preferable to use FPGA-LUTs. Besides, when the ROM input width is less than 8-bit is impossible to map the ROM to the internal to the FPGA RAM (block selectRAM). Additionally, the use of FPGA-LUTs do not increase the algorithm execution latency.

The measurements of the performance analysis are shown

in Table 1. Since the Whirlpool is the latest hash standard no other implementations exist. So, comparisons with other hash families' implementations [7]-[13] are added in order to have a fair and detailed comparison with the proposed implementations. We symbolized as *BB* (Boolean expressions Based) the mini boxes implementations by using Boolean expressions, and as *LB* (LUT Based) the mini boxes implementations by using FPGA-LUTs.

TABLE 1
PERFORMANCE ANALYSIS MEASUREMENTS

IMPLEMENTATION	FPGA DEVICE	CLB SLICES*	FREQUENCY (MHz)	THROUGHPUT (Mb/s)
MD5 [7] (Iterative arch.)	DEVICE 1	880	21	165
MD5 [7] (Full loop unrolling arch.)	DEVICE 1	4763	71.4	354
SHA-1 [8]	DEVICE 2	2606	37	237
SHA-1 [9]	DEVICE 3	-	18	114
MD5 [9]	DEVICE 3	-	18	142
SHA-1 [10]	DEVICE 4	1004	42.9	119
MD5 [10]	DEVICE 4	1004	42.9	146
SHA-2 (256) [10]	DEVICE 4	1004	42.9	77
RIPEMD-160 [10]	DEVICE 4	1004	42.9	89
SHA-1 [11]	DEVICE 5	2212	87	530
SHA-2 (512) [11]	DEVICE 5	3441	55.5	670
SHA-1 [12]	DEVICE 6	1550	38.6	899.8
MD5 [12]	DEVICE 6	1369	60.2	467.3
SHA-2 (256) [13]	DEVICE 7	1060	83	326
SHA-2 (384) [13]	DEVICE 7	1966	74	350
SHA-2 (512) [13]	DEVICE 7	2237	75	480
Prop. 1st_impl_BB	DEVICE 8	3815	75	1920
Prop. 1st_impl_LB	DEVICE 8	3751	93	2380
Prop. 2nd_impl _BB	DEVICE 8	5713	72	3686
Prop. 2nd_impl _LB	DEVICE 8	5585	87.5	4480

***CLB SLICES, Configurable Logic Block Slices:** The basic building block of the FPGA. It is used for area measurements.

Both implementations (1st and 2nd) were realized by the same FPGA device. The reason for this selection is the large number of I/Os (256-bit input and 512-bit output). If an external interface with less I/Os is demanded a smaller device can be used. But, this reduces the total algorithm implementation performance. The algorithm constants (c^r) are stored in a ROM which is implemented by using LUT.

In order to store the necessary round keys the 1st implementation uses a 10x512-bit RAM. This RAM is mapped to the 5K bits distributed SelectRAM. The 1st implementation requires 20 clock cycles for each block. So, the BB implementation throughput is 1.92 Gb/s at 75 MHz clock frequency, and the LB implementation throughput is 2.38 Gb/s at 93 MHz.

The 2nd implementation was designed in order to support applications with high throughput requirements. It demands 10 clock cycles for each data block but requires more hardware resources. The BB implementation throughput is 3.7 Gb/s at 72 MHz clock frequency and the LB implementation throughput is 4.5 Gb/s at 87.5 MHz.

From Table 1, some important results can be extracted. The first is that the Whirlpool implementation requires more hardware resources compared with the other hash families' implementations. This is a logical result of the algorithm philosophy and not an implementation tradeoff. Second, it performs much better in terms of throughput compared with all the previous hash families published implementations [7]-[13]. Also, whereas Whirlpool structure is more complex its operating clock frequency is competitive and some cases better compared with the other hash families' implementations. Finally the Whirlpool has the smaller algorithm execution latency. It needs only 10 clock cycles in

order to transform each block compared with the 64 clock cycles of the MD5, and SHA-2 (256), and 80 clock cycles of the RIPEMD-160, SHA-1, and SHA-2 (384, 512). This is an important advantage of the hardware implementation.

5. CONCLUSION

An efficient architecture and VLSI implementations for the new hash function, named Whirlpool are presented in this paper. Since four implementations have been introduced, each specific application can be choose the appropriate speed-area trade-off implementation.

A pipelined architecture is one option in order to improve the time performance of the Whirlpool implementation. It is possible to insert a negative-edge pipeline register, in round function ρ, as the Fig. 3, shows (dash line, after the permutation π). This register can be inserted, roughly in the middle of the round function. This is an efficient way in order to reduce the critical path delay, with a small area (512-bit register) penalty. So, the clock frequency can be roughly doubled and the time performance will increase without any algorithm execution latency increase.

Another way in order to improve the implementation performance is the usage of more pipeline stages. It is possible to insert 3 pipeline stages for the implementation of the round function ρ. The first positive-edge pipeline register is inserted in the same position as in the previous paragraph described. The other two pipeline registers are inserted before and after the tables X, X^2, and X^3 (dash lines in Fig. 3). Someone could claim that this pipeline technique is inefficient, for the Whirlpool implementation, because in order to process the m_i block, the result (H_{i-1}) from the previous processed block (m_{i-1}) is needed as a cipher key. Afterward, this feature prohibits the possibility to process simultaneously more than one block. But, in applications with limited processor strength, like smart cards, the above pipeline technique is essential in order to reduce the critical path delay and the efficiently execution of the Whirlpool.

The National Security Agency (NSA) did not disclose the SHA-2 design criteria and also, the internal structure of Whirlpool is very different from the structure of the SHA-2 functions. So, it is unlikely an attack against one will hold automatically for the other. This makes the Whirlpool a very good choice for consumer electronics applications.

ACKNOWLEDGMENT

The authors thank Paulo S. L. M. Barreto for his valuable comments.

REFERENCES

[1] SHA-1 Standard, National Institute of Standards and Technology (NIST), Secure Hash Standard, FIPS PUB 180-1, on line available at www.itl.nist.gov/fipspubs/fip180-1.htm

[2] "Advanced encryption standard", on line available at http://csrc.nist.gov

[3] SHA-2 Standard, National Institute of Standards and Technology (NIST), Secure Hash Standard, FIPS PUB 180-2, on line available at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf

[4] "NESSIE. New European scheme for signatures, integrity, and encryption", http://www.cosic.esat.kuleuven.ac.be/nessie

[5] P. S. L. M. Barreto and V. Rijmen, "The Whirlpool hashing function". Primitive submitted to NESSIE, September 2000, revised on May 2003, http://planeta.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html

[6] International Organization for Standardization, "ISO/IEC 10118-3: Information technology – Security techniques – Hash functions – Part 3: Dedicated hash-functions". 2003.

[7] Janaka Deepakumara, Howard M. Heys and R. Venkatesam, "FPGA Implementation of MD5 hash algorithm", Proceedings of *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2001)*, Toronto, Ontario, May 2001.

[8] N. Sklavos, P. Kitsos, K. Papadomanolakis and O. Koufopavlou, "Random number generator architecture and VLSI implementation", Proceedings of *IEEE International Symposium on Circuits and Systems (ISCAS 2002)*, USA, 2002.

[9] Yong kyu Kang, Dae Won Kim, Taek Won Kwon and Jun Rim Choi, "An efficient implementation of hash function processor for IPSEC", Proceedings of *Third IEEE Asia-Pacific Conference on ASICs*, Taipei, Taiwan, August 6-8, 2002.

[10] Sandra Dominikus, "A hardware implementation of MD4-Family algorithms", Proceedings of *IEEE International Conference on Electronics Circuits and Systems (ICECS 2002)*, Croatia, September 2002.

[11] Tim Grembowski, Roar Lien, Kris Gaj, Nghi Nguyen, Peter Bellows, Jaroslav Flidr, Tom Lehman, and Brian Schott, "Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512", Proceedings of *fifth International Conference on Information Security (ISC 2002)*, LNCS, Vol. 2433, Springer-Verlag, Sao Paulo, Brazil, September 30-October 2, 2002.

[12] Diez J. M., Bojanic S., Stanimirovic Lj., Carreras C., Nieto-Taladriz O., "Hash algorithm for cryptographic protocols: FPGA implementation", Proceedings of 10th Telecommunications forum (TELFOR 2002), November 26-28, Belgrade, Yugoslavia, 2002.

[13] N. Sklavos and O. Koufopavlou, "On the hardware implementation of the SHA-2 (256, 384, 512) hash functions", Proceedings of *IEEE International Symposium on Circuits and Systems (ISCAS 2003)*, May 25-28, Bangkok, Thailand, 2003.

[14] A. J. Menezes, P. C. van Oorschot, S. A. Vastone, *Handbook of applied cryptography*, CRC Press, 1997.

[15] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis*, Ph. D. thesis, KU Leuven, March 1995.

P. Kitsos was born in Athens, Greece in 1975. He received the B.S. in Physics from the University of Patras, Greece. He is currently pursuing his Ph.D. in the Department of Electrical and Computer Engineering at the University of Patras. His research interests include VLSI design, hardware implementations of cryptography algorithms, security protocols for wireless communication systems, and Galois field arithmetic implementations. He has published many technical papers in the areas of his research.

O. Koufopavlou (M'90) received the Diploma of Electrical Engineering in 1983 and the Ph.D. degree in Electrical Engineering in 1990, both from University of Patras, Greece. From 1990 to 1994 he was at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. He is currently an Associate Professor with the ECE Department, University of Patras. His research interests include VLSI design, VLSI crypto systems, and high performance communication subsystems. Dr. Koufopavlou has published more than 100 technical papers and received patents and inventions in these areas. He served as general chairman for the IEEE ICECS'1999.