Hardware Implementation of The Chameleon Polymorphic Cipher-192

MAGDY SAEB
Arab Academy for Science, Technology & Maritime Transport,
Computer Engineering Department,
Alexandria, Egypt
On Leave to: Malaysian Institute of Microelectronic Systems (MIMOS)
Office of Chief Researchers,
Kuala Lumpur, Malaysia

ABSTRACT
The Chameleon Cipher-192 is a polymorphic cipher that uses a variable word size and variable-size user's key. The cipher employs a shuffler and two nonlinearity-associated filters for selective addition. The cipher structure is based on the simultaneous use of block and stream cipher approaches. Other elements of the cipher include a specially-developed hash function for key expansion. In addition, this hash acts as a PRG to provide a random input to the filters. These filters are designed to elaborate the enciphering sequence by irregularly interrupting the data encryption at pseudo random, however, recoverable intervals. The cipher provides concepts of key-dependent number of rotations, key-dependent number of rounds and key-dependent addresses of substitution tables. The parameters used to generate the different substitution words are likewise key-dependent. In a previous work, we have established that the self-modifying proposed cipher, based on the aforementioned key-dependencies, provides an algorithm polymorphism and adequate security with a simple parallelizable structure. In this work, we provide an analysis of this cipher and an FPGA implementation.
KEYWORDS: FPGA, Cipher, Polymorphic, Hardware, Analysis.

1. Introduction

The Chameleon Cipher-192 is a polymorphic cipher that can be efficiently implemented in hardware. Contrary to conventional ciphers where it is implicitly assumed that the cipher machine is not reprogrammable, the proposed polymorphic cipher utilizes the user key to change the parameters of its operations. Three constructs that are key-dependent are proposed. These are: Shuffle, Select/Remove and Change parameters [SAEB09]. One considers the user key as the system memory where both the user key data and cipher re-programmability instructions are stored. The proposed cipher is a word-based cipher with variable word and key sizes. The bit-level S-orb replaces the conventional S-box leading to a noticeable increase of addressing space and added security. The key stream and the number of rounds are both key-dependent; thus eliminating the possibility of trap door functions. The generated S-orb is key-dependent using a specially-developed hash- function. Two large integer numbers are used to generate the different S-orb words. These two numbers are also key-dependent. These key-dependencies provided the foundation from which this polymorphic cipher acquired its name. The objective of using selective additions is to enhance homophonic substitutions. In these homophonic bit-level substitutions, the mapping of characters varies depending on the sequence of bits in the message text. Inside the encryption process, the round keys act initially as pointers in the homophonic substitutions without directly being part of the computations. Finally, a poly-alphabetic substitution is performed on the data. This involves using bit-wise XOR between the partially ciphered data and the generated keys. The security of this cipher is a direct consequence of the polymorphic key-dependent design of the cipher operation parameters. The paradigm of polymorphic encryption provides the required security with relatively simple round function constructs. We have preserved the pseudo-random permutations using robust bit-wise homophonic substitutions. In addition, we have utilized the capabilities of contemporary processors' superior performance to achieve acceptable execution speeds. In the following sections, we provide a discussion regarding the building blocks of the cipher, the proposed nonlinearity-associated filters, the multiplexer-based shuffler and the hash function. Moreover, we present an analysis of the algorithm, the results of the FPGA implementation and our conclusions.

Manuscript received November 5, 2009
Manuscript revised November 20, 2009

2. Building Blocks

The formal description of the algorithm, as shown in [SAEB09], is summarized as follows:

Algorithm Chameleon-Cipher
[*Given a plain text message P, key K, the aim of the algorithm is to encrypt the plain text into a cipher text C and decrypt it again. To achieve this the algorithm utilizes a specially developed hash function to generate the key stream, and a dynamic transposition to permute the plain text, and finally modulo two addition to scramble a varying-size data unit*]

Encrypt:

Input: Plain text P, key K **Output:** Cipher C, word-size

Algorithm body:

Initialize the S-orb

Input: n is a positive integer ε $\mathbf{Z}^+$ equal to number of words of the S-orb, p_i, q_i are pairs of large positive integer numbers ε $\mathbf{Z}^+$ required to update the iterative application of the hash function.

Output: A 192-bit n-word table utilized as a pseudo random number generator PRG called the S-orb.

Begin {Initialize S-orb body }

i: = 0;

$h_0 := p_0 . h(K) + q_0$;

{Hash the user key using MDP-192}

While i <= n

$h_{i+1} := p_{i+1} . h_i(k) + q_{i+1}$;

Save in S-orb file;

End while;

End Initialize.

Begin {Encrypt}

{P[m] = m blocks in P file}

Divide the Plaintext file P into m-1 192-bit blocks; Append last block if necessary;

Read max-number of rounds from user key; {Input max-number of rounds from user key from assigned secret location in user key}

If max-number of rounds < 4 then max-number of rounds: = 4;

For round = 1 to max-number of rounds

While (P[m] ≠ EOF) {EOF: End Of File}

j := 0;

While j ≠ n

Read kw [j] of S file;

Using the round key kw[j], read value of integer given by bit location 23-to-29; {This address represents the address of the center element of the block}

For the next block address, slide the 7-bit window two bits to the right and find new block address;

Divide the plain text 192-bit block into six 32-bit words, or twelve 16-bit words, or twenty four 8-bit words depending on user word-size;

From the LSB and moving to the right of the word-to- be encrypted: {Input: P[m], kw[j](round key), Output: C_{i1}}
If k_i=1 then move depending on location weights 0,1,2,...7 to N, NE, …, NW respectively then xor with corresponding bit of round-key kw[j];
Else do nothing;
ROTL (r); {r is determined from key 5-bit field (16-20) value, output C_{i2}}
{Input: C_{i2}, kw [j](round key), Output: C_{i3}}
If k_i=0 then move depending on location weights 0,1,2,...7 to N, NE, …, NW respectively then xor with corresponding bit of round-key kw[j];
Else do nothing;
{Input: C_{i3}, $kw[j]_i$ (round key), Output: C_i}
Ci = Ci3 xor kw[j];
Save C_i in output file
End while;
Next round;
End Algorithm.

This algorithm is best described by the following conceptual block diagram. A specially-designed hash function is used iteratively to generate the round keys and the selection vector. This hash function is based on the user key and the two constants that are obtained from the first round key. The second part is the encryptor, which is based on two nonlinearity-associated filters, a shuffler and a masking xor-based operation.

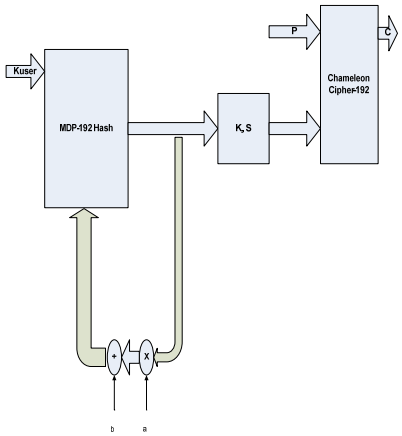


Fig. 1 The conceptual block diagram of the Chameleon Cipher

2.1 Nonlinearity-associated Filters

One of the basic building blocks of this cipher is the set of digital filters associated with nonlinearity. Similar filters are commonly used in digital signal processing applications [LING07]. These filters are designed to refine the enciphering sequence by interrupting the data scrambling at random, however, recoverable intervals. The filters' symbols are shown in Figures 2 and 3 where P, K, and S are the plaintext, the key and the select input respectively. As long as the K and S inputs are highly random, then the correlation between the P bit stream and C bit stream will be negligible.

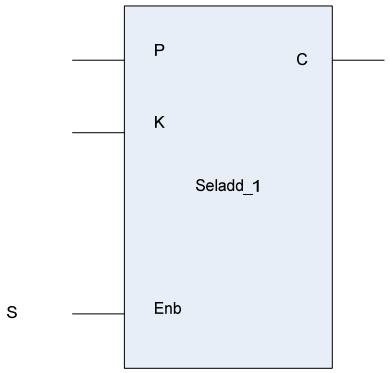


Fig. 1 The nonlinear filter seladd_1 symbol

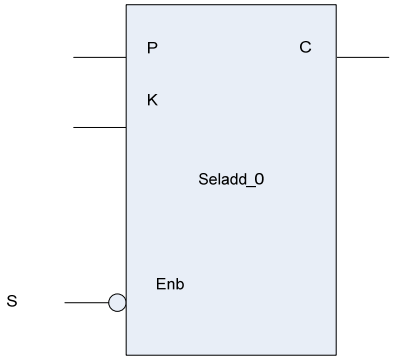


Fig. 2 The nonlinear filter seladd_0 symbol

The Chameleon Cipher-192, as explained in detail in [SAEB09], utilizes the user key for two major functions; to generate the sub-keys as in conventional ciphers and more importantly to vary the cipher encrypting parameters to achieve polymorphism. In the simplified hardware version of this cipher, we use the key as a source of data for generating the sub-keys. Simultaneously, the key serves as data flow and rotation controller to attain the required security through polymorphism. The simplification of the cipher is essential for low gate-count FPGA device implementation. The primary thought behind this hardware implementation is to provide two data paths; one path where bit-wise xor operation takes place and the second path where no operation takes place depending on the key value. This is called selective addition. Certainly, all operations are performed at the bit level; otherwise, there will be information leakage. To clarify this idea, the general conceptual diagram is shown in Figure 4. The select input S can be taken from another designated sub-key bit. If the select input, taken from the round key, is one, then a bit-wise xor operation is performed. If the selection bit is zero, no operation will take place. In this case, the cipher bit will be the same as the plaintext bit. This combined operation, as mentioned before, is called selective addition.

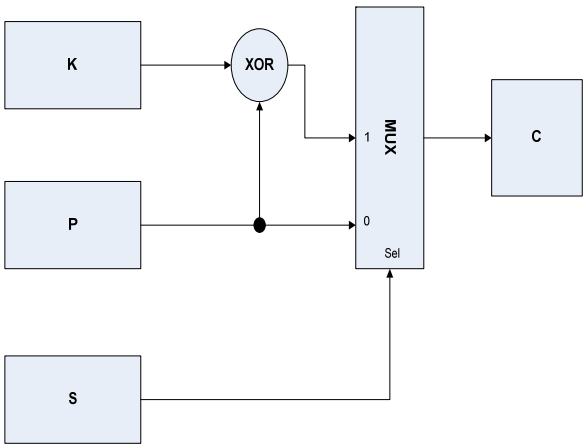


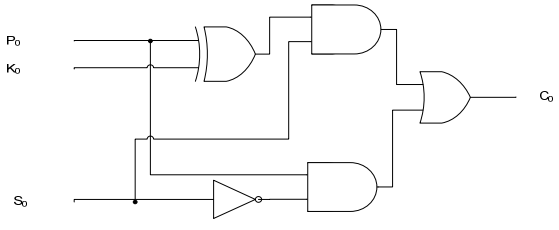
Fig. 3 The nonlinear filter basic operation; Selective addition conceptual diagram

This basic operation is summarized in Table 1. To eliminate any bias to the one value, the same circuit concept is utilized again, after performing a rotation operation. Alternatively, we use the complement of the S input.

Table 1: The basic circuit operation of the selective addition

Value of S (select input)	Operation
0	$C_i \leftarrow P_i$
1	$C_i \leftarrow K_i$ xor P_i

To clarify the operation of this circuit, we provide some details as shown in the following few lines. Redirecting the data flow, one uses a multiplexer, implemented at the gate level, as shown in Figure 5. The select input S_0 can be taken from another designated sub-key bit; P_0, K_0, and C_0 are the input plain text, the sub-key and the enciphered bit respectively.



F

ig. 4 The seladd_1 digital logic circuit

Table 2 is the truth table for such a one-based selective addition circuit.

Table 2: The truth table for seladd_1 filters

P	K	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

A basic requirement for any cipher to be viable is that it will show no bias to the number of ones or zeroes. This is usually referred to as the ciphertext being bit-balanced. Therefore, one has to perform the same operation, using zero-based selective addition rather one-based selection, to obtain the required bit-balance. The circuit shown in Figure 6 performs this function. The select input S_0 can be taken from another designated sub-key bit.

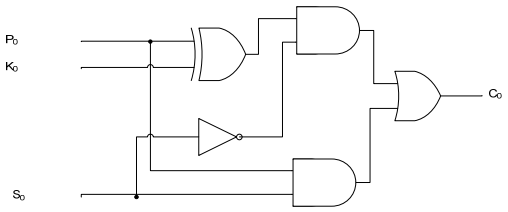


Fig.5 The seladd_0 digital logic circuit

Table 3 is the truth table for such a zero-based selective addition circuit.

Table 3: The truth table for seladd_0 filters

P	K	S	C
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

In both of these two circuits, one observes that the output cipher is bit-balanced, as long as the key is bit-balanced. The ASCII table used for plaintext is also bit-balanced. However, the frequency of various characters is different giving rise to roughly 30% bias to the number of zeroes in the resulting text messages. More bias in the number of zeroes or ones is noticed in some image files where a large area of the image may be of the same color. Increasing the cipher number of rounds will usually take care of this condition. On the other hand, the selection bits are normally bit-balanced. They are obtained from a sub-

key data itself where all sub-keys were previously generated using a hash function that, by design, provides a bit-balanced output. Moreover, the final xor operation of Chameleon Cipher-192 provides the required masking operation. It is worth mentioning that both of these two circuits can simply be implemented as lookup tables (LUT). This is illustrated in Figure 7, shown below. The implementation using the logic circuits of Figures 5 and 6 provide security through computation with small silicon area consumption and relatively longer execution times.

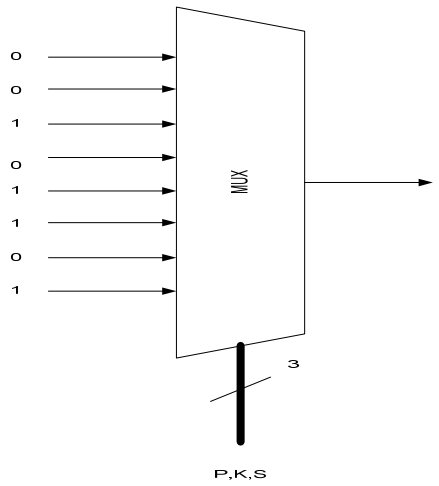


Fig. 7 Lookup LUT realization of the seladd_0

On the other hand, using a LUT such as the one shown in Figure 7 provides faster execution times and on the expense of larger consumed implementation area. One can visualize the two alternatives as security by computation versus security by randomization respectively. The randomization can be obtained by changing the LUT dynamically while initializing the system as long as the output is bit-balanced. However, this idea requires further investigation in future work.

2.2 The Shuffler

To provide better shuffling of bits, one uses a rotation operation between the two previously discussed operations. The rotation operation does not change the bit-balance condition of the register contents. The rotate left is performed a certain number of times that are key-dependent. This is achieved by using a set of multiplexers as shown in Figure 8. In this Figure the data flow is from the bottom register to the right hand register through the MUXs shown. The rotation operation is an essential cryptographic low level operation. It provides data shuffling without changing the state of bit-balance since the number of 1's and the number of 0's do not change. Unfortunately, there is no hardware-support for such an operation in most processors. The only alternative left to the programmer is to use shift operations. However, this type of implementation consumes a number of cycles equal to the number of register bits even for one rotation. Accordingly, this operation is quite slow and requires the largest amount of power consumption. Fortunately, one can implement a shuffler simply by using a set of MUX's. The delay in this case will be few cycles depending on the type of MUX used. However, the MUX select codes, two bits in the case shown in Figure 8, have to be mutually exclusive. To generate such a code, one may have to resort to some sort of shift register that, in turn, will slow down the whole operation. Therefore, a design decision was undertaken to resort to shuffling by using the same select code on all MUX's and changing the inputs to the various MUX's as shown in Figure 8. The estimated delay is about three cycles assuming one clock cycle for each logic gate in the data path. Still, the disadvantage of this approach is to limit the shuffling within the module bits. In order to correctly shuffle, say, 32 bits we need a 32-to-1 MUX's. In other words, repeating the 4-bit module eight times will not provide the same security, at least in theory, when compared to building a 32-bit module. The number of logic gates required for the 4-bit module is 28 logic gates. To use this module for 32-bit shuffling we need 8 modules with a total of 224 logic gates. On the other hand, if we use 32-to-1 MUXs, then each MUX requires 38 logic gates. For a 32-bit shuffling, we need 32 MUX's of this type. The total number of gates required in this case is 1216 logic gates. This is more than 5 times the 4-bit module. For a 64-bit block, the number of logic gates increases to 4544 logic gates. Then the proportion, as compared to the 4-bit module, is about 10 times more logic gates. One can call this the price of security! In this implementation, we will use a maximum block size of 32 bits. Consulting the literature, one can find trials to build some other cryptographic shufflers. However, these are based on extensive use of sequential logic with appreciable increase in data path delays. The MUX-based shuffler, on the other hand does not increase the estimated delay. Only the implementation area is noticeably increased. With continuing increase in FPGA device gate count, it appears that the consumed implementation area will be a less important factor when compared to the data throughput. Another point to be taken into consideration when designing cryptographic shufflers is the fact that MUX's are an essential design primitive in all FPGA families. Therefore, in this case, the designer would expect optimum implementation.

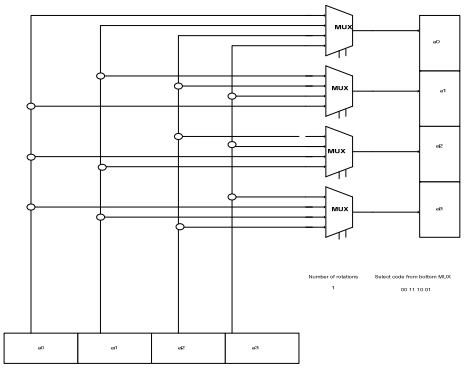


Fig. 8 A Shuffler using MUX's

2.3 The Hash Function MDP-192

The sub-keys and the select inputs, used in the selective addition modules, are obtained from a PRG that is constructed by using a specially designed hash function. This hash is called MDP-192 and is explained next. The design of new hashes should follow, we believe, an evolutionary rather than a revolutionary paradigm. Consequently, changes to the original structure are kept to a minimum to utilize the confidence previously gained with SHA-1 and its predecessors MD4 and MD5. However, the main improvements included in MDP-192 [SAEB09a], as shown in Figure 9, are:

The increased size of the hash; that is 192 bits compared to 128 and 160 bits for the other two schemes. That is the security bits have been increased from 64 and 80 to 96 bits. The message block size is increased to 1024 bits providing faster execution times. The message words in the different rounds are not only permuted but computed by xor and addition with the previous message words. This renders it harder for local changes to be confined to a few bits. In other words, individual message bits influence the computations at a large number of places. This, successively, provides faster avalanche effect. Moreover, adding two nonlinear functions and one of the variables to compute another variable, not only eliminates the possibility of certain attacks but also provides faster data diffusion. The fifth improvement is based on processing the message blocks employing six variables rather than four or five variables. This contributes, we believe, to better security and faster avalanche effect. The deliberate introduction of asymmetry in the procedure structure may help impede potential future attacks. The variables are not only permuted but also computed iteratively using data-dependent xor operation. On the other hand, the xor and addition operations do not cause appreciable execution delays for today's processors. Nevertheless, the number of rotation operations, in each branch, has been optimized to provide fast avalanche with minimum overall execution delays. The MDP-192 hash function was implemented using Cyclone II device and Altera integrated design environment. The hash is used to generate the sub-keys as discussed in [SAEB09] and the random selective input for the nonlinear filters.

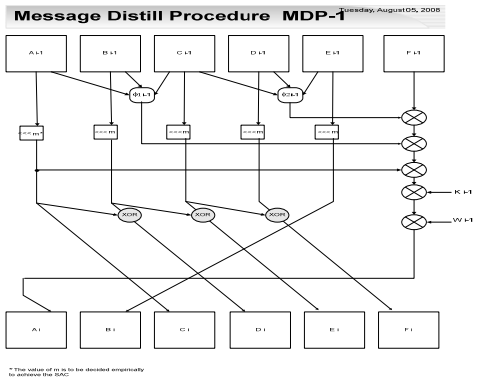


Fig. 9 Operation of MDP-192 hash function [SAEB09a]

3. Discussion of the Algorithm

Given the cipher text c and a part of the message m ∈ M, the attacker objectives [ZENR04] are:

1. To find the set of all probable message $M^{-} \in M$ where $m^{-} \in M$ if $\exists\, k \in K \mid E(k, m) = c$. The two functions E: K x M→C and D: K x C→M are satisfying D (k, E (k, m)) = m, ∀ m ∈ M and k ∈ K.
2. To find the set $K^{-} \subseteq K$ if $k^{-} \in K \Leftrightarrow \exists\, m^{-} \in M : E(k^{-}, m^{-}) = c$.

To design a semantically-secured cipher, two major building blocks are required; a pseudo random number generator PRG and an encryption function. The PRG is a function $G :\{0, 1\}^{l} \rightarrow \{0, 1\}^{*}$ that expands a short seed into a bit sequence of arbitrary length. This function has inner state $S_i \in \{0,1\}^{l}$ and an update function $f: \{0,1\}^{l} \rightarrow \{0,1\}^{l}$ and an output function $g: \{0,1\}^{n} \rightarrow \{0,1\}$, $n \leq l$. The update function f modifies the inner state between two outputs. The output function g computes the next output bit from the current inner state or part of it. This canonic structure was implemented in Chameleon by using a specially–designed hash function acting as the function G. This PRG is secure if and only if a pseudo One-Time-Pad using G is secure. Consequently, for a bounded number of the inner states of G, the encryption process had to be refined by adding two other sources of nonlinearities:

1. Nonlinearity-associated filters,
2. Irregular data encryption intervals.

The essential design elements of the Chameleon Cipher, as explained in detail in [SAEB09], are:

- Algorithm polymorphism based on the notion of a key-driven encryption rather than only using the user key to generate the sub-keys in conventional ciphers,
- The structure of the cipher is based on the simultaneous use of block and stream cipher approaches to conform with the concept of Universal Secure Encryption USE,
- The utilization of a simple nonlinearity-associated filters to refine the enciphered sequence by interrupting the data scrambling at random, however, recoverable intervals,
- A separate specially-designed hash function is used to generate the sub-keys. In case of a valid attack on the sub-keys this hash can be changed easily. In addition, if the user requires more security a hash cascade can be used,
- In the software version, a rotation operation is used to shuffle the data. However, in the hardware version this operation was replaced by a MUX-based shuffler that is purely combinational logic to appreciably reduce the number of cycles required to complete the shuffling.

In conventional ciphers, the transposition operation is performed on the character level. This results in information leakage and invites the attacker to reposition the cipher text characters to find some intelligible plaintext. However, if the transposition is performed on the bit-level, a very large number of wrong messages can be formed that hide the only correct one. In other words, an agglomerated large number of bits will replace a relatively small bundle of characters. Moreover, if conventional transposition is used on the same set of identical characters, no permutation will change this data block. This is quite clear in multimedia files. In Chameleon Polymorphic Cipher, these permutations are performed on the bit-level preserving the bit balance condition in the output cipher. Thus, the cipher is emulating communication white noise. This is a clear requirement in semantic security or what is known as USE discussed in [BLGO85], [GOMI82] [ZENR07] and [ZENR04]. The efficient substitutions performed by Chameleon provide a huge number of different transpositions with almost equal probability. This requires a large PRG with a relatively large number of internal states. In conventional ciphers, on the other hand, a relatively diminutive number of all possible keyed permutation tables can be produced. A major constraint that is facing the cryptographer is to generate a cipher with equal probability of generating 1's and 0's. This was achieved by proper design of the utilized mentioned filters. However, as we have pointed out before, some data blocks may not be bit-balanced. In this case, one can resort to artificially bit-balance these blocks by appending balancing values to plaintext. The other option is to rely on the indirect randomization obtained by multiple rounds of encryption. The first option may be more time-consuming when compared to an increase in the number of rounds. However, we have to concede that this point requires more investigation on different types of multimedia files. We have used the second option by applying an increased key-dependent number of rounds. Testing has showed conformity with the cipher bit-balanced requirement as shown in [SAEB09]. The bit-level selective substitution high computational cost is partially offset by today's multi-execution unit superscalar processors. Opposed to the complex proofs of computational security of conventional ciphers, Chameleon Polymorphic Cipher is based on simple combinatorics as shown in [SAEB09]. It was shown that the attacker has a diminutive probability of figuring out the correct algorithm used and at the same time a negligible probability of key collision. In this respect, if the hash function was successfully attacked, one can change it or even use a hash cascade to generate the sub-keys.

4. FPGA Implementation

The simplicity of the hardware building blocks of the Encryptor lends itself to straight forward implementations. Either using TTL conventional technology or even advanced FGPA technology can be used to implement the design. The circuit diagram for a demonstrative four-bit encryption module is shown in Figure 10. In this Figure, the four stages are clearly shown. The first stage is the one selective addition. The second and third stages are the shuffler using four MUX's and the zero selective addition stage. The last operation is the xor operation. To obtain minimum pin-to-pin delay and accordingly high throughput, we have not used any sequential logic in the construction of the encryption module. The sequential logic, omitted here for simplicity, is used to only store the data block before and after encryption. The expected delay of the circuit is 10 cycles per byte. The circuit, shown in Figure 10, was implemented using Verilog, [BRVR08], [WAKE01], [ASICWRLD] and Quartus II 6.1 Web Edition [ALTRA], Altera design environment.

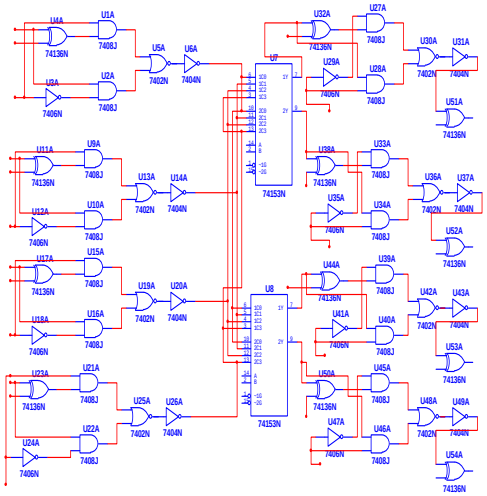


Fig. 10 The details of the circuit diagram for a 4-bit module using TTL family of logic gates

The implementation was performed on a EP2C5T144C6, Cyclone II family device. All results shown are for a simplified four-bit version of the Encryptor to convey the basic concepts of the design. The worst case pin-to-pin delay was 9.643 ns. The minimum delay was 8.550 ns and the average delay was calculated to be approximately 9.1173 ns. The system frequency is 100 MHz. A series of screen-captures of the different design software outputs are shown in Figures 11 to 17. Figures 11, 12, 13, and 14 provide indication of successful compilation, the nonlinear filter module RTL, the shuffler module RTL and the encryption module RTL respectively. Figure 15 displays the Encryptor simulation results where output pins 15, 16, 17, and 20 are the encrypted data. Figure16 and 17 demonstrate the timing report and the floor plan respectively. Figure 18 displays the schematic block diagram for the implementation of hash function.

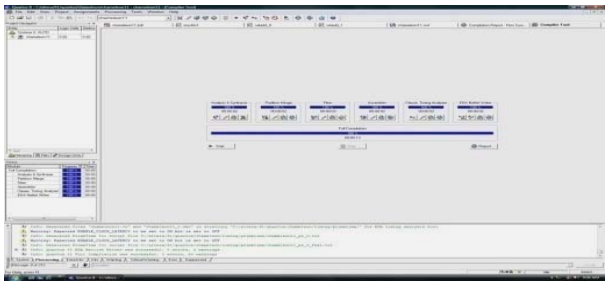


Fig. 11 Compiler tool screen

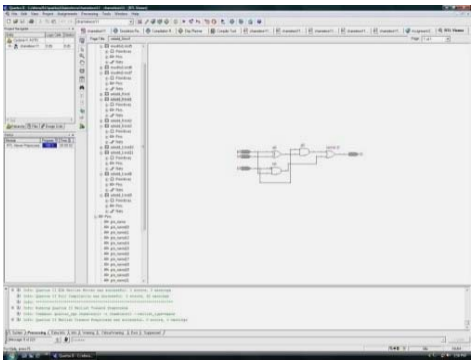


Fig. 12 RTL screen for the nonlinear filter module

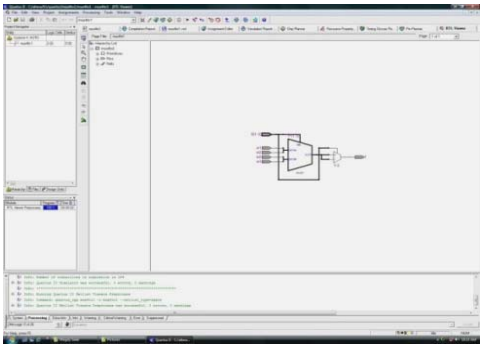


Fig. 13 RTL screen for part of the shuffler module

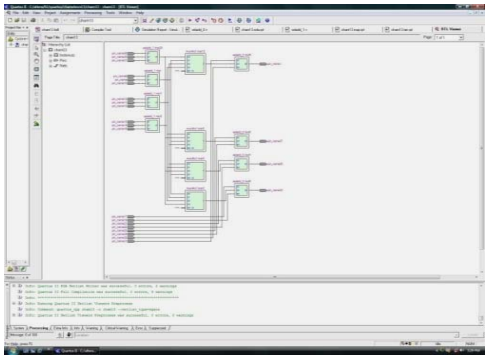


Fig. 14 RTL screen for the Encryption module

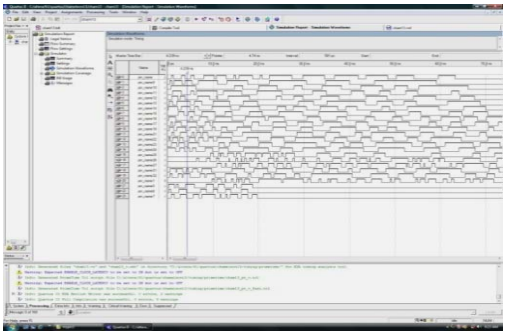


Fig. 15: Simulator screen showing the output encrypted data

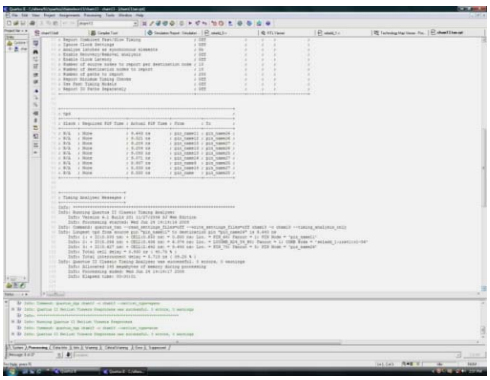


Fig.16 Timing report

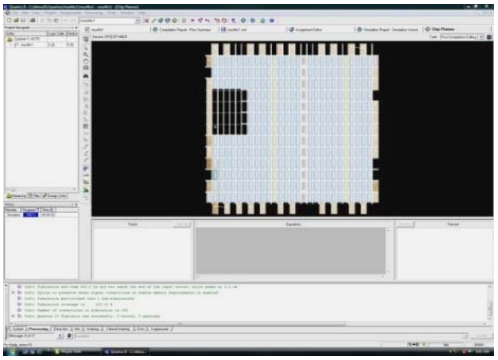


Fig.17 Floor plan

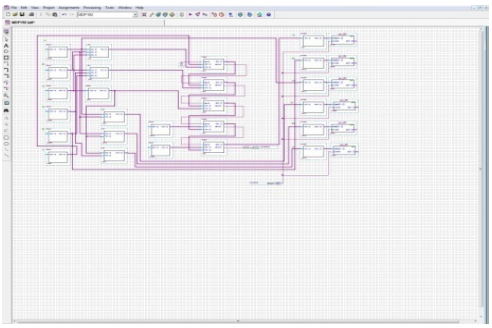


Fig.18 Schematic diagram of the MDP-192 hash function

5. Summary & Conclusion

We have given a brief discussion of the following hardware building blocks for the Chameleon Cipher (CC-192):

1. The nonlinearity-associated filters are based on irregular, however recoverable, interruption of the encryption process. One can view this approach as hiding encrypted bits in some plaintext bits using two keys; one key for encryption and one for hiding.
2. The FPGA-hardware implementation of these filters was discussed based on a computational model and a lookup table.
3. A pure combinational logic multiplexer-based shuffler, with minimum expected delay, is proposed and implemented.
4. The encryptor part of Chameleon was implemented using simple Verilog modules that were integrated using the schematic editor.
5. The PRG utilized is a specially-designed hash function that was implemented using simple Verilog code and a schematic editor.
6. The design of the cipher, the filters and the hash function are, in general, based on the notion that the cryptographic low level operations, which are bit-balanced are: xor, complement (invert) and rotate. All encryption processes are based on these three operations or their equivalents.

In this work, we have provided a hardware-implementation proof of concept. The estimated 10 cycles per byte delay was verified by the timing reports providing, at 100 MHz operating frequency, an average and maximum pin-to-pin delays of 9.1173 and 9.643 ns respectively. The total interconnect delay is 5.713 and the total cell delay is 3.930. However, at this stage of development, no area or timing optimization were performed. A comparison with other implementations is not applicable since this is the first time the cipher is FPGA-implemented. This and other related issues will be dealt with in future development of the device.

References

[ALTRA] www.altera.com/support/examples/verilog/verilog.html

[ASICWRLD] www.asic-orld.com/verilog/verilinks.html

[BLGO84] M. Blum, S. Goldwasser, “An Efficient Probabilistic Public key Encryption Scheme which Hides All Partial Information,” Proceedings of Advances in Cryptology-CRYPTO’84, pp.289-299, Springer Verlag, 1985.

[BRVR08] S. Brown, Z. Vranesic, Fundamental of Digital Logic with Verilog Design, McGraw-Hill International Edition, 2008.

[GOMI82] S. Goldwasser, S. Micali, “Probabilistic Encryption & How to Play Mental Poker Keeping Secret all Partial Information,” Annual ACM Symposium on Theory of Computing, pp. 365-377, San Fran, US., 1982.

[LING07] Wing-Kuen Ling, Nonlinear Digital Filters: Analysis and Applications, Academic Press, 2007.

[SAEB09] Magdy Saeb, "The Chameleon Cipher-192 (CC192): A Polymorphic Cipher," SECRYPT2009, International Conference on Security & Cryptography, Milan, Italy, 7-10 July, 2009.

[SAEB09a] Magdy Saeb, "Design & Implementation of the Message Digest Procedures MDP-192 and MDP-384," ICCCIS2009, International Conference on Cryptography, Coding and Information Security, Paris, June24-26, 2009.

[WAKE01] J. F. Wakerly, Digital Design Principles & Practice, third edition, Prentice Hall, 2001.

[ZENR04] Erik Zenner, On Cryptographic Properties of LFSR-based Pseudorandom Generators, Ph.D. Dissertation, University of Mannheim, Germany, 2004.

[ZENR07] Erik Zenner, "Why IV Setup for Stream Ciphers is Difficult," Dagstuhl Seminar Proceedings 07021, Symmetric Cryptography, March14, 2007.
http://drops.dagstuhl.de/opus/volltexte/2007/1012

Appendix:

SAMPLE VERILOG CODE

```
module seladd_1(p1,k1,s1,c1);
        input p1,k1,s1;
        output c1;
        xor(a1,p1,k1);
        and(g1,~s1,a1);
        and(h1,p1,s1);
        or(c1,g1,h1);
endmodule
```

```
module mux4to1 (w0,w1,w2,w3,S,f);
        input w0,w1,w2,w3;
        input [1:0] S;
        output f;
assign f = S[1]? (S[0]? w3:w2):(S[0] ? w1:w0);
endmodule
```

```
module seladd_0(p0,k0,s0,c0);
        input p0,k0,s0;
        output c0;
        xor(a0,p0,k0);
        and(g0,s0,a0);
        and(h0,p0,~s0);
        or(c0,g0,h0);
endmodule
```

```
module phiJ (x, y, z, f);
        input [31:0] x , y, z;
        output [31:0] f;
        assign f = (x & y)|(~x & z);
endmodule
```

```
module fulladd1(carryin , X, Y, S, carryout);
        input carryin;
        input [31:0]X,Y;
        output[31:0]S;
        output carryout;  wire [31:1] C;
 fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
 fulladd stage1 (carryin, X[1], Y[1], S[1], C[2]);
 :
:
endmodule
```

```
module fulladd (Cin, x, y, s, Cout);
        input Cin, x, y;
        output s, Cout;
        assign s = x^ y^ Cin;
 assign Cout = (x &y)|(x &Cin)|(y & Cin);
endmodule
```

Magdy Saeb received the BSEE. School of Engineering, Cairo University, in 1974; the MSEE. and Ph.D. in Electrical & Computer Engineering, University of California, Irvine, in 1981 and 1985, respectively. He was with Kaiser Aerospace and Electronics, Irvine California, and The Atomic Energy Establishment, Anshas, Egypt. Currently, he is a professor in the Department of Computer Engineering, Arab Academy for Science, Technology & Maritime Transport, Alexandria, Egypt, (on leave) to Malaysian Institute of Microelectronic Systems (MIMOS), Kuala Lumpur, Malaysia. His current research interests include Cryptography, FPGA Implementations of Cryptography and Steganography Data Security Techniques, Encryption Processors, Computer Network Reliability, Mobile Agent Security. www.magdysaeb.net.