Hardware Implementations for the ISO/IEC 18033-4:2005 Standard for Stream Ciphers

Paris Kitsos

Abstract**—**In this paper the FPGA implementations for four stream ciphers are presented. The two stream ciphers, MUGI and SNOW 2.0 are recently adopted by the International Organization for Standardization ISO/IEC 18033-4:2005 standard. The other two stream ciphers, MICKEY 128 and TRIVIUM have been submitted and are under consideration for the eSTREAM, the ECRYPT (European Network of Excellence for Cryptology) Stream Cipher project. All ciphers were coded using VHDL language. For the hardware implementation, an FPGA device was used. The proposed implementations achieve throughputs range from 166 Mbps for MICKEY 128 to 6080 Mbps for MUGI.

Keywords**—**Cryptography, ISO/IEC 18033-4:2005 standard, Hardware implementation, Stream ciphers

I. Introduction

SECURING data in transmission is the most common real-life cryptographic problem. Basic security services require both encryption and authentication. This is mainly realized using a private-key algorithm and a Message Authentication Code (MAC).

It is common to classify encryption algorithms as public-key algorithms, that are typically used to establish secure connections over insecure channels, and private-key algorithms, that due to their inherent efficiency, are employed to secure the bulk data transmission phase. Although among the class of private-key, block ciphers are probably the best known and well-studied objects during the last years a major effort taken place in order to propose new and secure stream ciphers.

Stream ciphers are practical approximations to the one-time pad and can operate on blocks as small as a single bit [1]. One-time pad is a random key used to operate on plaintext of equal length once and never used again. The drawback of one-time pad is that the length of the key has to be equal to that of plaintext and that there is a difficulty in distributing the key.

Stream ciphers have different implementation properties than block ciphers that restrict the cryptanalyst. They only receive their inputs once (a key and an initialization vector) and then produce a long stream of pseudo-random data.

A stream cipher can start with a strong cryptographic operation to thoroughly mix the key and initialization vector into a state, and then use this state and a simpler mixing operation to produce the keystream. If the attacker tries to manipulate the inputs of the cipher encounters the strong cryptographic operation. As there are fewer cryptographic requirements to fulfill, the keystream generation function can be made significantly faster than a block cipher can be.

Fig. 1 shows the general diagram of the cipher process with stream cipher [2]. The stream cipher takes two parameters, the secret key, *K*, and the initialization vector, *IV*, and produces the keystream bits, z_t. In stream encryption each plaintext symbol, P_t, is encrypted by applying a group operation with a keystream symbol, z_t, resulting in a ciphertext symbol c_t. In modern cipher the operation is the simple bitwise XOR.

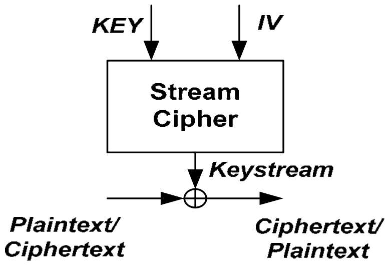


Fig. 1 The stream cipher process

Decryption takes the substraction of the keystream symbol from the ciphertext symbol. With the bitwise XOR this is the same operation.

In this paper the stream ciphers, that are recently adopted by the International Organization for Standardization ISO/IEC 18033-4:2005 standard, are described and implemented [3]. This standard comprises the MUGI [4] and SNOW 2.0 [5] stream ciphers. In addition, except the above ciphers, two others have been implemented, the MICKEY-128 [6] and TRIVIUM [7] stream ciphers. These ciphers are implemented for comparisons in design philosophy and time performance with the standard ciphers purposes. The designers' directions are generated by the new wireless standards where demand hardware environments where gate count, power consumption and memory are very limited.

The MICKEY-128 and TRIVIUM stream ciphers have been submitted and are under consideration for the eSTREAM, the ECRYPT (*European Network of Excellence for Cryptology)* Stream Cipher project [8]. The eSTREAM is a multi-year effort to identify new stream ciphers that might become suitable for widespread adoption.

Manuscript received January 13, 2006.

Paris Kitsos is with the Digital Systems and Media Computing Laboratory, School of Science & Technology, Hellenic Open University, Patras, Greece (phone: +302610367535, fax: +302610367520; e-mail: pkitsos@ieee.org).

The following of this paper is structured as below. In section II the implemented stream ciphers specifications are briefly described. In section III the hardware architectures and implementations are explained in detail while in section IV the synthesis results and comparisons between the stream ciphers are mentioned. Finally, the section V concludes the paper.

II. Stream Ciphers Specifications

A. MUGI Pseudorandom Number Generator

MUGI is a pseudorandom number generator (PRNG) used as a stream cipher. The design aims to be suitable for both software and hardware implementations. MUGI has two independent parameters as inputs. The first one is 128-bit secret key while the second one is 128-bit initial, public, vector. MUGI generates a 64-bit length random bit string in each round. The design of MUGI is similar to PANAMA stream cipher [9]. So, it consists of four main operational modules. As Fig. 2 shows the *Internal State* is divided in two parts, *State a* and *Buffer b*.

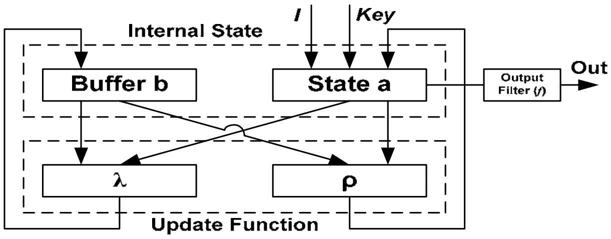


Fig. 1 A PANAMA-like stream cipher

The *Update Function* is divided in analogy to the internal state. Note that each update function uses another internal state as a parameter. We denote the update function of State *a* and Buffer *b* as *ρ* and *λ* functions respectively. The output filter *f* abstracts some bits of State *a* for each round.

The operation of MUGI is divided in two phases. The initialization phase and the keystream generation phase. The initialization phase is divided in 3 steps. Firstly, the secret key *K* is inserted into state *a*. Then initialize buffer *b* with running *ρ*. Secondly, adds the initial vector *I* into state *a* and initializes state *a* with running *ρ* and last the whole internal state is mixed. In the keystream generation phase runs *n* rounds update function and outputs a part of the internal state (64-bit) for each round.

More details about the cipher specification can be found in [4].

B. SNOW 2.0 Stream Cipher

The main parts of SNOW 2.0 are a *Linear Feedback Shift Register* (*LFSR*) of length of 16 and a *Finite State Machine* (*FSM*). The *FSM* has two input words, taken from the *LFSR*, and the running key is XORed between the *FSM* output and the last element of the *LFSR*. This operation diagram is depicted in Fig. 3.

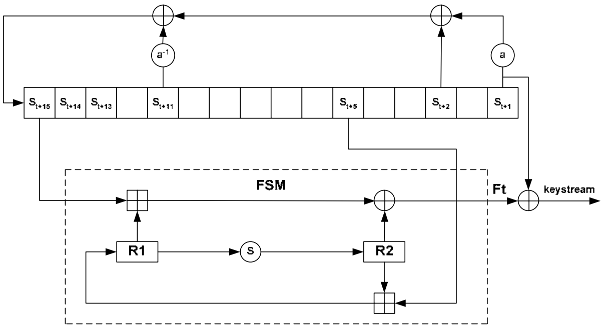


Fig. 3 The SNOW 2.0 stream cipher

The feedback polynomial is given by the (1).

$$\pi(x) = \alpha x^{16} + x^{14} + \alpha^{-1} x^5 + 1 \in F_{2^{32}}[x] \qquad (1)$$

where α is the root of (2)

$$x^4 + \beta^{23} x^3 + \beta^{245} x^2 + \beta^{48} x + \beta^{239} \in F_{2^8}[x] \qquad (2)$$

and β is a root of (3)

$$x^8 + x^7 + x^5 + x^3 + 1 \in F_{2^2}[x]. \qquad (3)$$

The *FSM* has two registers *R1* and *R2*, each holding 32-bit data. The value of the registers at time $t \geq 0$ is denoted as $R1_t$ and $R2_t$ respectively. The input to the *FSM* is (s_{t+15}, s_{t+5}) and the output of the *FSM*, denoted as *Ft*, is calculated as follows (4)

$$F_t = (s_{t+15} + R1_t) \oplus R2_t, \;\; t \geq 0 \qquad (4)$$

and the keystream is given by

$$z_t = F_t \oplus s_t, \;\; t \geq 1. \qquad (5)$$

The registers *R1* and *R2* are updated with new values according to

$$R1_{t+1} = s_{t+5} + R2_t \qquad (6)$$

and $R2_{t+1} = S(R1_t), \;\; t \geq 1.$ (7)

The addition (+) in the (4) and (6) is symbolized as ⊞ in Fig. 3 and we mean integer addition modulo 2^{32} whereas with symbol $\oplus$ we mean bitwise addition (XOR). The S-box is the similar to the RIJNDAEL S-box [10].

The operation of the cipher is the following. First, a key initialization is performed. This operation provides the *LFSR* with a starting state as well as giving the internal *FSM* registers *R1* and *R2* their initial values. The cipher is clocked for 32 times. Instead, the FSM output, F_t, is incorporated with the feedback loop. By time *t=0* we mean the time instead directly after the key initialization. Then the cipher is clocked once before producing the first keystream symbol, i.e., the first keystream symbol, denoted as z_1, is produced at time *t=1*. Then the cipher is clocked again and the second keystream symbol is read. This process is repeated until the cipher produces the keystream with length equal to plaintext/ciphertext length.

More details about the cipher specification can be found in [5].

C. MICKEY-128 Stream Cipher

MICKET-128 stream cipher is intended to have low complexity in hardware, while providing a high level of security. It uses irregular clocking of shift registers, with some novel techniques to balance the need for guarantees on period and pseudorandomness to avoid certain cryptanalytic attacks. The overall diagram of MICKEY-128 stream cipher is shown in Fig. 4a.

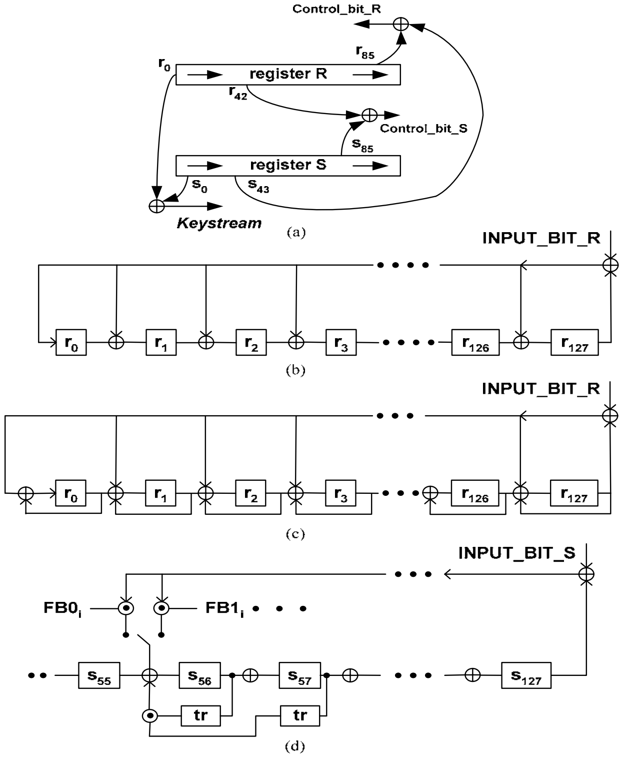


Fig. 4 The TRIVIUM stream cipher

MICKEY-128 takes two input parameters, the 128-bit secret key, *K*, and an initialization variable, *IV*, anywhere between 0 and 128-bit in length. Two 128-bit registers *R* and *S* are used in order to build it. The *R* register is a *Linear Feedback Shift Register* (*LFSR*) and the *S* register is a *Non-linear Feedback Shift Register* (*NFSR*). Two variables are defined. The *Control_bit_R* and the *Control_bit_S*. The *Control_bit_R* is defined as *s43* xor *r85* and the *Control_bit_S* is defined as *s85* xor *r42* where *s43*, *s85*, *r42* and *r85* are *42nd* bit and *85th* bit of the register *R*, the *43rd* bit and *85th* bit of the register *S*. When *Control_bit_R=0* the register *R* is a standard *LFSR* as Fig. 4b shows. When *Control_bit_R=1*, as well as shifting each bit in the register to the right, we also XOR it back into the current stage, as shown in Fig. 4c. This corresponds to multiplication by $x+1$. Fig. 4d shows the design of the *NFSR S* register. The variables *FB0i* and *FB1i*, the transformation *tr* and the mathematical equations, that the register *S* uses for each building, can be found in reference [6].

D. TRIVIUM Stream Cipher

TRIVIUM is a synchronous stream cipher designed to generate up to 264 bits of keystream from an 80-bit secret key, *K*, and an 80-bit initial value, *IV*. As for most stream ciphers, this process consists of two phases. First, the internal state of the cipher is initialized using the key and the *IV*, and then the state is repeatedly updated and used to generate keystream bits. Fig. 5 shows the architecture of the TRIVIUM stream cipher.

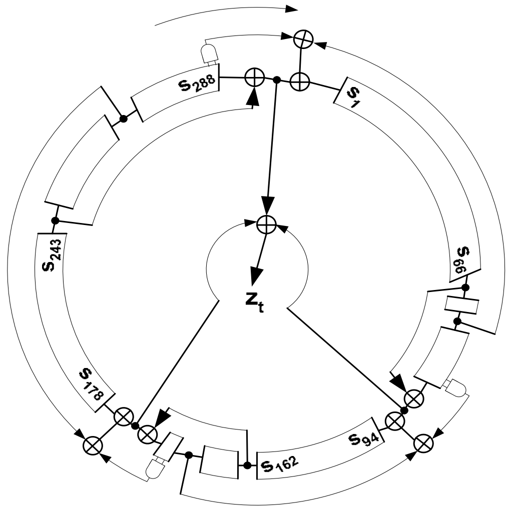


Fig. 5 The TRIVIUM stream cipher

The proposed design contains a 288-bit internal state denoted as ($s_1,\ldots, s_{288}$). So, the algorithm is initialized by loading an 80-bit key and an 80-bit *IV* into the 288-bit initial state, and setting all remaining bits to 0, except for s_{286}, s_{287}, and s_{288}. Then, the state is rotated over 4 full cycles, without generating key stream bits.

The keystream generation consists of an iterative process which extracts the values of 15 specific state bits and uses both of them to update 3 bits of the state and to compute 1 bit of key stream z_t. The state bits are then rotated and the process repeats itself until the requested $N \leq 2^{64}$ bits of key stream have been generated.

III. Architectures and Hardware Implementations

A. MUGI Architecture

The architecture that performs the MUGI pseudorandom number generator is shown in Fig. 6. As this figure shows the main parts of the proposed architectures are the State *a*, the Buffer *b* and the functions *ρ* and *λ*. In addition, one 128-bit register is used for latching of the secret key and initialization vector. Also, the *K/I init* component is used for key and initialization vector transformations [4] before the algorithm initialization phase starts. The *Auxiliary Buffer1* holds the Buffer *b* data while the *Auxiliary Buffer2* holds the State *a* data during the initialization phase. Finally, there are a 3x192 multiplexer (*MUX*) and two *XOR* gates, 128-bit and 64-bit respectively, accomplishing the generator architecture.

The initialization phase of MUGI is divided into 3 steps. Firstly, Buffer *b* is initialized with a secret key, *K*. Secondly, the initialization of the State *a* with the initial vector, *I*, takes place. Last the whole internal state is mixed. So, when the key is transformed is fed by the State *a* through the *IN2* multiplexer input. Then, *a*, iterates only the function *ρ* and puts a part of each *a*(*t*) into Buffer *b* as follows, $b_{15-i} = (\rho^{i+1}(a,0))_0$. In the previous equation ρ^i means the *i*-th iteration of ρ and $\rho\ (a,0)$ means the input from Buffer *b* is zero. In other words, the data stored into Buffer *b* is not used in this step. The *Auxiliary Buffer1* is responsible for this. In addition, the output data of the State *a* are never used in the first step of the initialization phase. The *Auxiliary Buffer2* is responsible for this. In the second step the mixed State *a* with value $a(K) = \rho^{16}(a_0,0)$ and the initial vector, *I*, are required. If *I* is added to State *a* through the *IN1* multiplexer input, State *a* is mixed again by 16 rounds iteration of function *ρ*. So, the mixed State *a* is represented as $\rho^{16}(a(K,I),0)$.

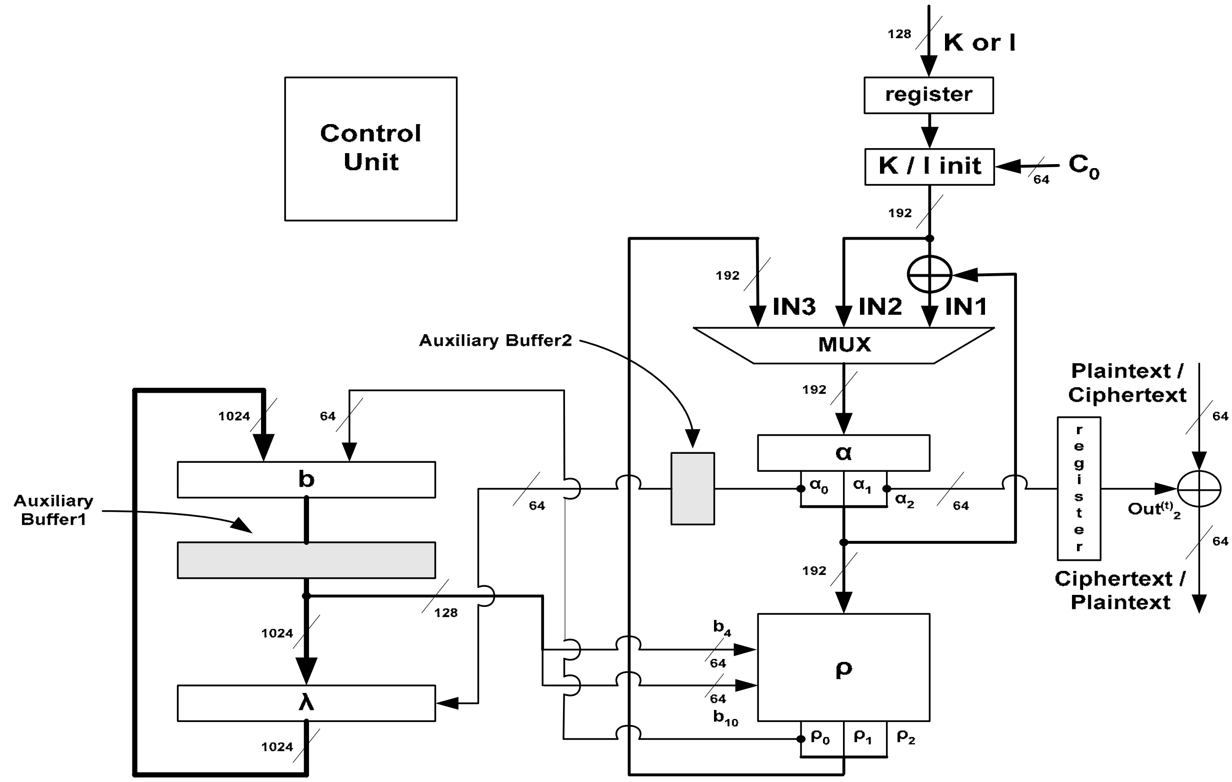


Fig. 6 MUGI Pseudorandom Number Generator Architecture

Finally, the last initialization phase step is a 16 rounds iteration of the whole update function *Update,*

$$a = Update^{16}(\rho^{16}(a(K,I),0),b(K)) \qquad (8)$$

where the notation $b(K)$ in the above equation denotes that the Buffer *b* is initialized by the secret key *K*

For security purposes, the algorithm output bits should not be available to the users during the initialization process. So, a 64-bit register is located at the generator output that does not latch its input bits during the initialization process.

After the initialization, the 64-bit register latches the generated bits and MUGI generates a 64-bit keystream. If we denote the output at round t as *Out*(*t*), then the output is given as,

$$Out(t) = a_2^{(t)} \qquad (9)$$

In other words MUGI outputs the lower 64-bit of State *a* at the beginning of the round process.

The State *a* and Buffer *b* are 192-bit and 1024-bit registers respectively. The update function of State *a* is the function *ρ*. It is a kind of target heavy Feistel structure with two F-functions and uses Buffer *b* as a parameter. The hardware architecture of function *ρ* is depicted in Fig. 7.

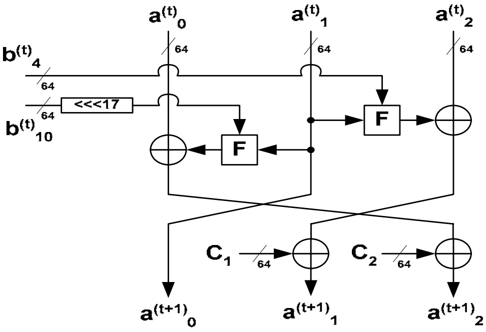


Fig. 7 The ρ Function Hardware Architecture

Furthermore, the hardware architecture of the *F*-function is depicted in Fig. 8. The bytewise substitution S-box is the same as the one in AES [10] while the linear transformation is the combination of a 4 x 4 matrix and a bytewise shuffling. The S-boxes have been implemented with LUTs. MUGI uses MDS matrix which is the component of AES.

In addition, the function λ is the update function of Buffer *b* and uses a part of State *a* as a parameter. The mathematical background of function λ can be found in [4]. The hardware implementation consists of simple XOR operations and bit-shifting. Also, the values of the *C0*, *C1* and *C2* constants are defined in [4].

Finally, the *Control Unit* is responsible for the correct operation of the whole algorithm.

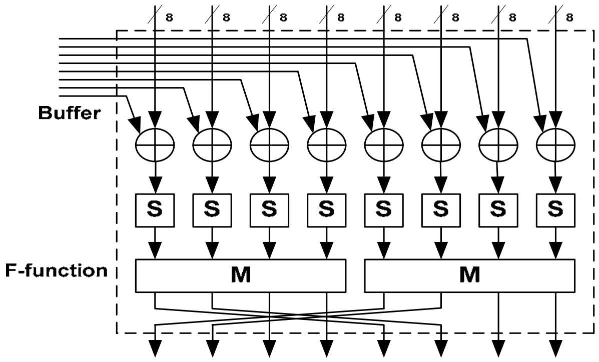


Fig. 8 The F-Function Hardware Architecture

B. *SNOW 2.0 Architecture*

The proposed SNOW 2.0 hardware architecture is depicted in Fig. 9. As this figure shows the architecture consists of two main parts. *The LFSR* and the *FSM*.

The *LFSR* is built with 16 32-bit registers. As Fig. 9b shows between each register a 32 2-input *OR* gate is located. The operation starts with the initial parallel loading of the initial values in each register according to the cipher specifications. After, the input is forced equal to zero and the cipher starts to operate due to the user's commands.

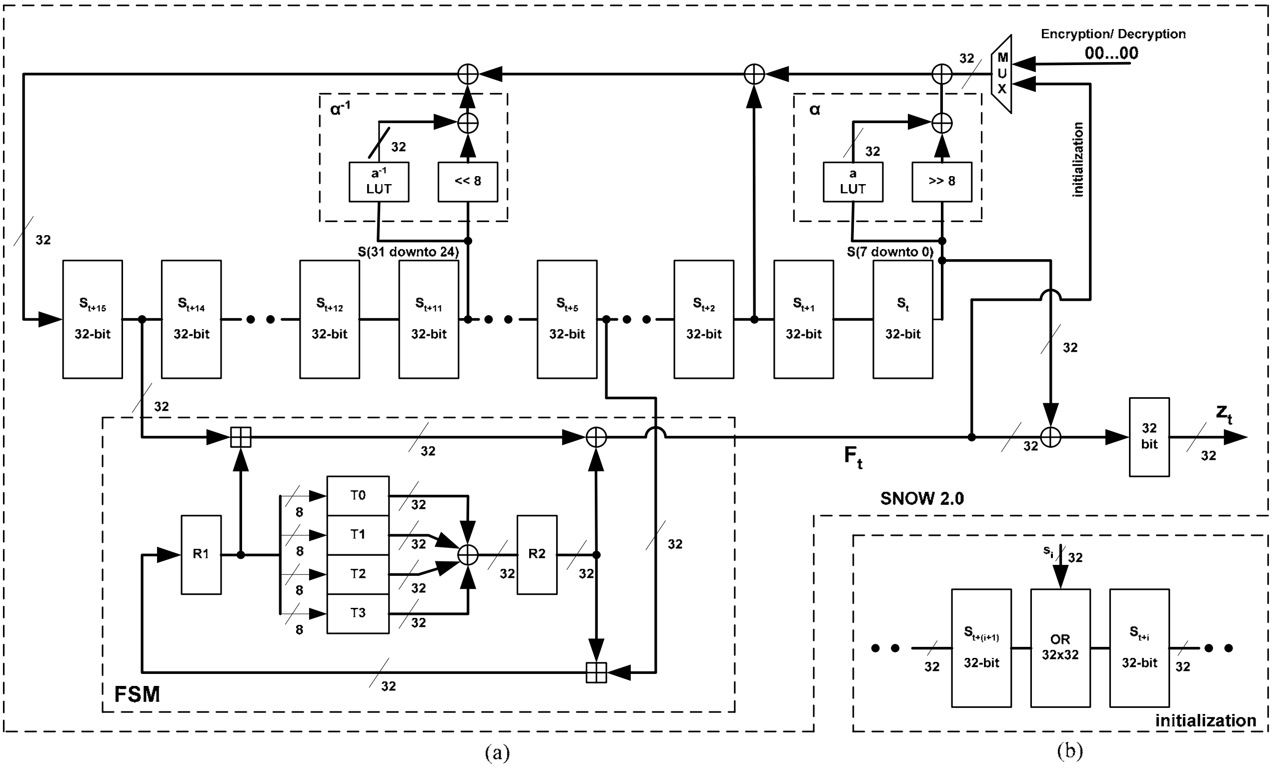


Fig. 9 SNOW 2.0 Stream Cipher Architecture

The multiplication with α and α^{-1} are implemented with *Look-Up-Tables* (*LUTs*) and 8-bit left or right shifting respectively. The values of α and α^{-1} are computed by the following equations.

$$\alpha = (w << 8)XORMUL\_\alpha[w >> 24] \tag{10}$$

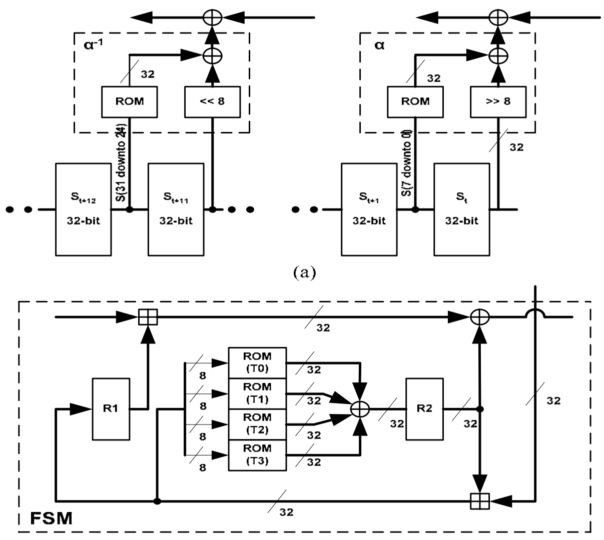
and

$$\alpha^{-1} = (w >> 8)XORMUL\_\alpha inverse[w] \tag{11}$$

The pre-computed tables *MUL_α* and *MUL_αinverse* are specified in [5].

The *FSM* consist of two 32-bit register, 2 integer adders (⊞) modulo 2^{32}, two 32-bit bitwise *XOR* (⊕) and four *LUTs*. These *LUTs* blocks are used for the *S-box* implementation.

Also, an efficient, in both area and time performance, design with *ROM* blocks is implemented. In this implementation the *MUL_α* and *MUL_αinverse* tables are implemented with *ROM* blocks. In addition, the tables for the *S-box* are implemented with *ROM* blocks. This implementation is illustrated in the following Fig. 10.



(a)

(b)

Fig. 10 The ROM Blocks Based Implementation

The only difference is that *ROM* blocks are synchronous in contrary to the *LUTs*. So, in cases of the multiplication with α and α^{-1} (see Fig. 9a) the taps is taken one state before in order to get the good values at the appropriate moment. Similar, for the *S-box* the *ROM* blocks take their inputs not form the R1 output but from the R1 input as the Fig. 10b shows.

For the adders implementations *Carry Save Adders* (*CSA*) are used. For security purposes, the cipher output bits should not be available to the users during the initialization process. So, a 32-bit register is located at the cipher output that does not latch its input bits during the initialization process.

In the initialization process the *FSM* output, F_t, is forced through the multiplexer, *MUX*, to the *LFSR* while during the encryption/decryption process the *LFSR* is forced through the multiplexer with zeros.

C. *MICKEY-128 Architecture*

The MICKEY-128 stream cipher architecture is shown in Fig. 11. This architecture consists of the registers *R* and *S* following the specifications demand. The way that the *Control_bit_R* and the *Control_bit_S* variables are defined is obvious. Also, the multiplexer *MUX* is shown that is needed in order to the data values are forced in the registers. Finally, the keystream bits are produced by the XORing of the *r0* and *s0* bits.

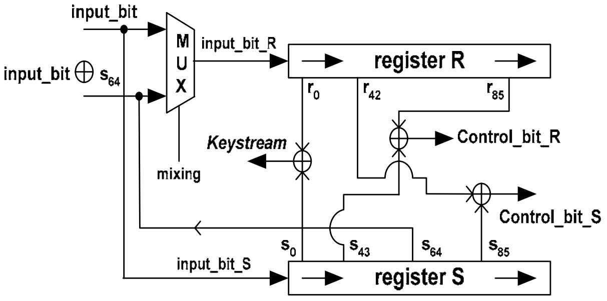


Fig. 11 MICKEY-128 Stream Cipher Architecture

The register *R* implementation is illustrated in Fig. 12.

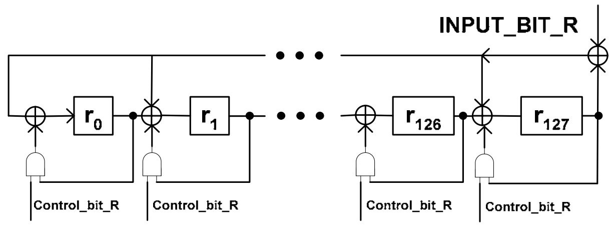


Fig. 12 The Implementation of the Register *R*

The *Control_bit_R* signal helps each AND gate to configure the register and works either in normal mode or multiplied the current stage with *x+1* polynomial.

For the register *S* a straightforward implementation was used.

For security purposes, similar to the previous ciphers, a flip-flop is inserted to the cipher output that does not latch its input bits during the initialization process.

D. *TRIVIUM Architecture*

As Fig. 5 shows the TRIVIUM is the simpler cipher implemented. Because of this simplicity a quite straightforward architecture is implemented. For the TRIVIUM hardware implementation 288 flip-flops, 11 2-input *XOR* gates, 3 2-input *AND* gates and 288 2-input *OR* gates are used. The *OR* gates are used in order to force the initial values in the flip-flops.

IV. Experimental Results and Performance Evaluation

Each one of the stream ciphers was captured by using VHDL, with structural description logic. All implementations were simulated for the correct operation by using the test vectors provided by each cipher specification. The VHDL codes were synthesized in a Xilinx Virtex-E V400EFG676 FPGA [11] for having a common hardware device for the comparison. Then the implementations were simulated again for the verification of the correct functionality in real time operating conditions. The selected FPGA has 160K-bit synchronous internal selectRAM blocks.

The synthesis results for the implemented stream ciphers are presented in Table I.

TABLE I
STREAM CIPHERS SYNTHESIS RESULTS

Stream Ciphers	FPGA Device (VIRTEX-E V400EFG676		
	# CLBs	# FGs	# DFFs
MUGI	2092	4183	2437
SNOW 2.0 (LUT)	1545	3089	701
SNOW 2.0 (ROM)	391	782	625
MICKEY 128	167	333	235
TRIVIUM	144	172	288

(#CLBs) Configurable Logic Blocks, (#FGs) Function Generators, (#DFFs) D Flip-Flops

As the above table shows the *MUGI* cipher consumed the more FPGA resources compared with the others. The *SNOW 2.0* (ROM) based implementation uses 48K-bit from the internal selectRAM blocks. The *TRIVIUM* stream cipher is the more compact compared with the others.

Performance comparisons in terms of frequency, throughput and implementation efficiency are presented in table II. The implementation efficiency (Mbps/#CLBs) is defined as the ratio between the cipher throughput and the number of CLBs that each cipher consumes. In addition, comparisons with other previously published stream ciphers are shown in this table.

As illustrated in table II the MUGI achieves the larger throughput. Also, it has the second implementation efficiency ratio. In addition, the SNOW 2.0 (ROM) based design is the most suitable for FPGA implementation due to its bigger ratio. Whilst their design simplicity the TRIVIUM has the worst efficiency ratio.

TABLE II
STREAM CIPHERS PERFORMANCE COMPARISONS

Stream Cipher	Freq. F (MHz)	Throughput (Mbps)	Implementation. Efficiency
MUGI	95	6080	2.9
SNOW 2.0 (LUT)	122	3904	2.5
SNOW 2.0 (ROM)	141	4512	11.5
MICKEY128	166	166	0.99
TRIVIUM	211	211	1.5
SNOW 2.0 [12]	-	5659	5.6
LILI-II [12]	-	243	0.3
SNOW 1.0 [13]	66.5	2128	2.8
RC4 [14]	61	120.8	0.8

To the best of our knowledge there are no published hardware implementations results for the MUGI, MICKEY 128 and TRIVIUM stream ciphers, which can be compared with our respective implementations. The implementation results for SNOW 2.0 are comparable with the ones in [12]. The proposed ROM based implementation is more efficient for FPGA implementation and covers less hardware resources, since in [12] the area was 1150 CLB slices. In contrary the proposed LUT based implementation achieve worse values in both area and time performance.

Compared with previous published stream ciphers [13]-[14] the proposed implementations achieve comparable and in the most cases better performance.

Finally, the MUGI and SNOW 2.0 stream ciphers outperform both the MICKEY 128 and TRIVIUM ciphers throughputs.

V. CONCLUSION

In this paper four stream ciphers are implemented in hardware and compared in terms of performance and consumed FPGA area. These ciphers were coded in VHDL language and synthesized in an FPGA device. The implementations for the MUGI, MICKEY 128 and TRIVIUM are the first in the literature. The largest throughput achieved by the MUGI cipher while the smallest achieved by the MICKEY 128 cipher. Finally, the most suitable for FPGA implementation is the ROM based SNOW 2.0 implementations that have the biggest implementations efficiency ratio.

REFERENCES

[1] M. J. B. Robshaw, “Stream Ciphers”, *RSA Laboratories Technical Report TR-701* Version 2.0, RAS Laboratories, July 1995.

[2] B. Schneier, *Applied Cryptography, Protocols, Algorithms, and Source Code in C*, John Wiley & Sons 1994.

[3] International Organization for Standardization, “ISO/IEC 18033-4:2005, Information technology -- Security techniques -- Encryption algorithms -- Part 4: Stream ciphers”, 2005.

[4] D. Watanabe, S. Furuya, H. Yoshida, and K. Takaragi, “MUGI pseudorandom number generator”, Specification, 2001, on line available at http://www.sdl.hitachi.co.jp/crypto/mugi/index-e.html

[5] P. Ekdahl, T. Johansson. A new version ot the stream cipher SNOW, available from http://www.it.lth.se/cryptology/snow/, 2002.

[6] Steve Babbage, Matthew Dodd, “The stream cipher MICKEY-128”, (ECRYPT) Stream Cipher Project Report 2005/016.

[7] Christophe De Canniθre and Bart Preneel, “Trivium - A Stream Cipher Construction Inspired by Block Cipher Design Principles”, (ECRYPT) Stream Cipher Project Report 2005/030

[8] ENCRYPT - European Network of Excellence in Cryptology, “Call for Stream Cipher Primitives”, Scandinavian Congress Center, Aarhus, Denmark, 26-27 May 2005, http://www.ecrypt.eu.org/stream/

[9] J. Daemen, and C. Clapp, “Fast hashing and stream encryption with PANAMA”, In Proc. of Fast Software Encryption: 5th International Workshop, FSE'98, Paris, France, March 1998.

[10] J. Daemen and V. Rijmen. The design of Rijndael: AES–The Advanced Encryption Standard. Springer-Verlag, 2002.

[11] Xilinx Virtex FPGA Data Sheets (2005), URL: http://www.xilinx.com

[12] P. Leglise, F.-X. Standaert, G. Rouvroy, J.-J. Quisquater, “Efficient implementation of recent stream ciphers on reconfigurable hardware devices”, In Proc. of 26th Symposium on Information Theory in the Benelux. May 19th-May 20th, 2005, Brussels, Belgium.

[13] K. Alexander, R. Karri, I. Minkin, K. Wu, P. Mishra, X. Li, “Towards 10-100 Gbps Cryptographic Architectures”, in proc. of CATT/WICAT Annual Research Review, 2003, on line available at http://wicat.poly.edu/tech report/tr/02-005.pdf

[14] M. D. Galanis, P. Kitsos, G. Kostopoulos, O. Koufopavlou, “Comparison of the Performance of Stream Ciphers for Wireless Communications”, in proc. of CCCT'04, Austin, Texas, USA, August 14-17, 2004.

Paris Kitsos was born in Athens, Greece in 1975. He received the B.S. in physics and its Ph.D from the Department of Electrical and Computer Engineering both at the University of Patras, Greece.

He is research fellow with the Digital Systems & Media Computing Laboratory, School of Science & Technology, Hellenic Open University (HOU). His research interests include VLSI design, hardware implementations of cryptographic algorithms and security protocols for wireless communication systems. Dr. Kitsos has published more than 50 scientific articles and technical reports, as well as is reviewing manuscripts for International Journals and Conferences/Workshops in the areas of his research.

Dr. Kitsos is an editorial board member of "Computer and Electrical Engineering, An International Journal (Elsevier Ltd.)", member of the IEEE (Institute of Electrical and Electronics Engineers), IEE (Institution of Electrical Engineers) and the International Association for Cryptologic Research (IACR). Finally, he has been organized special sessions in international conferences and workshops.