Annales UMCS Informatica AI XI, 2 (2011) 81–93
DOI: 10.2478/v10065-011-0012-9

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI
http://www.annales.umcs.lublin.pl/

Performance of algebraic graphs based stream-ciphers using large finite fields

Abderezak Touzene[1*], Vasyl Ustimenko[2†],
Marwa AlRaissi[1], Imene Boudelioua[1]

[1] *College of Science Sultan Qaboos University, Sultanate of Oman*
[2] *Institute of Mathematics, University of Maria Curie Sklodowska, pl. M. Curie-Sklodowskiej 1, 20-031 Lublin, Poland*

Abstract

Algebraic graphs $D(n,q)$ and their analog graphs $D(n,K)$, where K is a finite commutative ring were used successfully in Coding Theory (as Tanner graphs for the construction of LDPC codes and turbo-codes) and in Cryptography (stream-ciphers, public-keys and tools for the key-exchange protocols. Many properties of cryptography algorithms largely depend on the choice of finite field F_q or commutative ring K. For practical implementations the most convenient fields are F_m^2 and rings modulo Z_2^m modulo 2^m. In this paper the reader can find the first results about the comparison of $D(n,2m)$ based stream-ciphers for $m = 8,\ 16,\ 32$ implemented in C++. They show that performance (speed) of algorithms gets better when m is increased.

1. Introduction

Algebraic graphs $D(n,q)$ over finite fields F_q without cycles of length less than $n+5$ have been introduced in [**1**]. They turn out to be a convenient tool

*E-mail address: touzene@squ.edu.om
†E-mail address: ustymenko_vasyl@yahoo.com

in Coding Theory and Cryptography (see [**2, 1**] and further references). In [**3**] a more general family of graphs $D(n, K)$, where K is a commutative ring, was used for generation of private–key algorithms. Notice that $D(n, F_q) = D(n, q)$.

The stream–cipher based on the graphs $D(n, Z_2^m)$, $m = 8, 16, 32$ is considered and evaluated in [**4**]. It has been shown that the confusion properties of the algorithm get better when the graph based encryption map takes into account the combination of two special affine transformations of the cipher space.

In [**5, 6**] we presented some encryption tools based on walking on algebraic graphs over finite fields. The implementation of public-key algorithm based on the graphs $D(n, K)$ for some special rings (or fields) is presented in [**7**]. According to [**8**], non identical encryption map based on $D(n, K)$ is a polynomial map of the cipher space of degree 3. In [**9**], such a map and with connection to the group theoretical discrete logarithm, a key-exchange protocol is proposed.

In this paper, we investigate a fast and secure symmetric key encryption tool based on the graphs $D(n, F_2^m)$, for various value of $m = 8, 16, 32$. We may use this encryption map in combination with affine transformation of the cipher space. Our objective is to study the performance of the encryption tool for different values of $m = 8, 16, 32$. The connected components of graphs $D(n, q)$ grow with the growth of n (as well as with the growth of q). It means that our algorithms are not unit ciphers see [**9**]. It is, in fact, a stream cipher. We will use the term unit for the character of our natural alphabet.

The idea of graph based encryption is to treat messages as vertices of a graph and encryption steps as arcs of a graph (see [**10, 11**]). The encryption tool looks like walking on a graph with a huge number of vertices. A plain text is seen as a succession of n bytes or an n–tuple in a Galois $F(2^8)$, $F(2^{16})$ and $F(2^{32})$ (see [**12, 13, 14**]). The processes of encryption follow a unique path, which starts from the plain text (vertex v_1 of graph), and ends with the vertex v_k , which consists of the cipher text. We consider a one–step walk as an arc that connects v_i to the next vertex v_{i+1} and which uses one character of the password. It is known that the proposed family of graphs has no cycle of length $n + 5$, for $n > 3$ where n is the length (number of bytes) of the plain text. This property ensures that if the length of the password $l < (n + 5)/2$, each password will have a unique walk path. Thus starting with the same plain data file, different passwords will produce different cipher data files.

The rest of this paper is organized as follows: Section 2 introduces some basic definitions. Section 3 focuses on walk on the graphs algorithm and its complexity. In Section 4, we present experimental measurements for our algorithm. Section 5 concludes the paper.

2. Basic Definitions

2.1. Cryptography basics

Let us assume that an unencrypted message, plaintext, which can be image data, is a string of bytes. It is to be transformed into an encrypted string or cipher text, by means of a cryptographic algorithm and a key: so that the recipient can read the message, encryption must be invertible. Conventional wisdom holds that in order to defy easy decryption, a cryptographic algorithm should produce seeming chaos: that is a cipher text should look and test random. In theory, an eavesdropper should not be able to determine any significant information from an intercepted cipher text. Broadly speaking, attacks to a cryptosystem fall into two categories: passive attacks, in which adversary monitor the communication channel and active attacks, in which the adversary may transmit messages to obtain information (e.g. cipher text of chosen plaintext). Passive attacks are easier to mount, but yields less. Attackers hope to determine the plaintext from the cipher text they capture; an even more successful attacks will determine the key and thus comprise the whole set of messages.

An assumption first codified by Kerckhoffs in the nineteen century is that the algorithm is known and the security of algorithm rests entirely on the security of the key. Cryptographers have been improving their algorithms to resist the following two major types of attacks:

1. cipher text only: the adversary has access to the encrypted communications.
2. known plain text: the adversary has some plain text and corresponding cipher text.

Nowadays, the security of the plain text rests on both the encryption algorithm (good resistance to attacks of types (1) and (2)), and the algorithm for the key exchange (public keys) with good resistance to active attacks of type (2), when the adversary can generate each plain text p and get the corresponding cipher text c (see [**12, 13**]).

2.2. Finite field arithmetic

A field is an algebraic object with two operations: addition and multiplication, represented by $+$ and $*$, although they will not necessarily be ordinary addition and multiplication. Using $+$, all the elements of the field must form a commutative group, with identity denoted by 0 and the inverse of a denoted by $-a$. Using $*$, all the elements of the field except 0 must form another commutative group with identity denoted 1 and inverse of a denoted by $a - 1$. (The element 0 has no inverse under $*$) Finally, the distributive identity must hold: $a * (b + c) = (a * b) + (a * c)$, for all field elements a, b and c.

Cryptography focuses on finite fields [**12, 13**]. It turns out that for any prime integer p and any integer n greater than or equal to 1, there is a unique field with p^n elements in it, denoted $F(p^n)$. Here "unique" means that any two fields with the same number of elements must be essentially the same. In the case n is equal to 1, the field is just the integers mod p, in which addition and multiplication are just the ordinary versions followed by taking the remainder on division by p.

In our algorithms as for the new U.S. Advanced Encryption Standard (AES) we use the finite field $F(2^8)$. In fact, our crypto–system works with bytes (8 bits), represented from the right as: $b_7b_6b_5b_4b_3b_2b_1b_0$. The 8–bit elements of the field are seen as polynomials with coefficients in the field Z_2: $b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0$. The field elements will be denoted by their sequence of bits, using two hexadecimal digits.

Addition in $F(2^8)$:

To add two field elements, just add the corresponding polynomial coefficients using addition in Z_2. Here addition is modulo 2, so that $1 + 1 = 0$, and addition, subtraction and exclusive-or are all the same. The identity element is just zero: 00000000 (in binary) or 0x00 (hex).

Multiplication in $F(2^8)$:

Multiplication in this field is much more difficult, but it can be implemented very efficiently in hardware or software [**14**]. The first step in multiplying two field elements is to multiply their corresponding polynomials just as in basic algebra (except that the coefficients are only 0 or 1, and $1 + 1 = 0$ makes the calculation easier, since many terms just drop out). The result would be up to a degree 14 polynomial, which is too big to fit into one byte. A finite field now makes use of a fixed degree eight irreducible polynomial (a polynomial that cannot be factored into the composition of two simpler polynomials). As for the AES, we use the following irreducible polynomial: $m(x) = x^8 + x^4 + x^3 + x + 1 = 0x11b$ (hex).The intermediate composition of the two polynomials must be divided by $m(x)$. The remainder from this division is the desired product.

In our crypto–system implementation, we use variable unit size of 8, 16, and 32 bits units leading to the operations in $F(2^8)$, $F(2^{16})$ and $F(2^{32})$ respectively.

3. Walk on Graph Algorithm

3.1. Theoretical background

The graph is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$. The elements of V are the vertices (or points) of the graph G, the elements of E are its edges. In our algorithm, we treat messages as vertices of the graph and encryption steps

as the edges. A vertex is seen as a sequence of n units or n–tuple in $F(2^m)$. Consider an n–tuple vertex $X = (x_0, x_1, \ldots, x_{n-1})$, where x_i an element in $F(2^m)$, whereas, an edge is defined using a linear system of equations that will be defined later. Our graph is a bipartite graph. The graph generated using the algorithm has a high girth (very large cycles).

The basic idea of Walks on Graphs Algorithm is to consider vertices of a given bipartite graph as the message being encrypted or decrypted, and arcs between vertices as encryption/decryption tools (i.e. key and algorithm system of equations). The left side of the graph is called the point side, and the right side is called the line side. Initially, we start at the point side with the plaintext and movements between the point side and the line side will be carried on by several steps using units of the encryption key and applying specific equations on the finite field $F(2^m)$. Our algorithm is designed to allow variable unit size in the encryption/decryption process. Moreover, once the unit size is specified (i.e. 1-byte, 2-bytes or 4-bytes) the Galois Filed is chosen accordingly. For example, when unit size is chosen to be of size 8-bits, the operations of the algorithm will be performed over $F(2^8)$ and so forth. In the encryption process, a unit of the message will be encrypted by a unit of the encryption key to produce a unit of the cipher text and so on. In the first step, the first unit of the cipher text will be generated by using solely the first unit of the encryption key. Next, the rest of the units will be encrypted in the following pattern. Consider, a plaintext $X = (x_0, x_1, \ldots, x_{n-1})$ of length n and its corresponding cipher text $Y = (y_0, y_1, \ldots, y_{n-1})$, and an encryption key $K = (k_0, k_1, \ldots, k_{l-1})$ of length l. The set of vertices X and Y are n–dimensional vector spaces over the finite field $F(2^m)$. The components of X, Y and K are the elements of $F(2^m)$, so all operations will be done in this field. The following linear system of equations is used for encryption and decryption. That is, these equations are used to define the point and line adjacency relation between X and Y:

$$\begin{aligned} y_1 - x_1 &= y_0 \cdot x_0 \\ y_2 - x_2 &= y1 \cdot x_0 \\ y_i - x_i &= y_1 \cdot x_{i-2} \\ y_{i+1} - x_{i+1} &= y_1 \cdot x_{i-2} \\ y_{i+2} - x_{i+2} &= x_1 \cdot y_{i-2} \\ y_{i+3} - x_{i+3} &= x_1 \cdot y_{i-2}. \end{aligned} \tag{1}$$

Note: the last four equations are defined for $i > 2$ and x_0 and y_0 are always the first element of the encryption/decryption key. Given an n–dimensional vector space over the finite field $F(2^m)$, there is a vertex X in point side to a

vertex Y in the line side. Each step of the algorithm consists of either moving from the point side to the line side, or vice versa by using the system of equations (1).

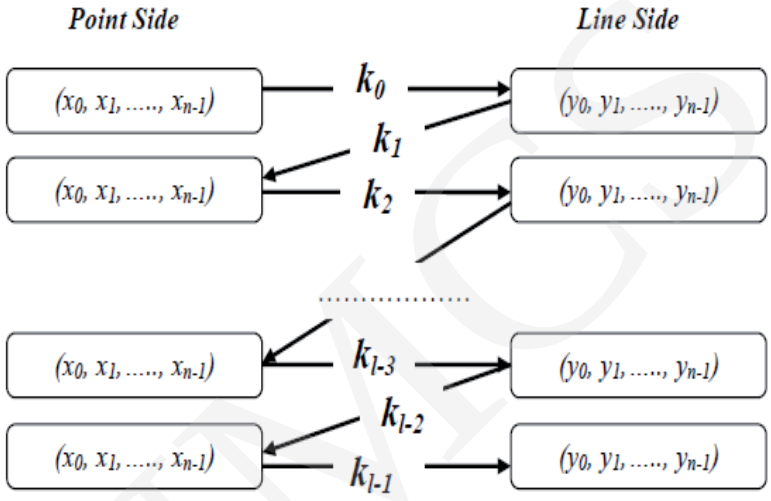

Fig. 1. Walks on the Graphs Algorithm Diagram.

The use of bipartite graph $D(n, q)$ of high girth guarantee that for a given length of a password the graph has no cycles and therefore there exists only one path leading from the plaintext to the cipher text as it is shown in the proposition.

Proposition 1. [**15**] *The graph* $D(n, q)$ *has:*

- *No cycle of length 4, for* $n = 2$,
- *No cycle of length 6, for* $n = 3$,
- *No cycle of length* $n + 5$, *for* $n > 3$.

The above proposition is very important, it states that if we use these graphs as an encryption tool for a data file of size n with a password of length $l < (n+5)/2$, each password will have a unique walk path. Starting with the same plain data file, different passwords will produce different cipher data files. In practice, the password length condition can be easily verified because in general the size of the data files n is much larger than l, the length of the password (from 10 to 30 bytes). Since there is only a unique path password $(k_0, k_1, \ldots, k_{l-1})$

from the vertex v_0 plain data to the vertex v_{l-1} which represents the encrypted data. For the decryption process, the same unique path traversed backward from v_{l-1} to v_0 is used. In fact, the decryption process uses the same algorithm as the encryption but using a reversed password $(k_{l-1}, k_{l-2}, \ldots, k_0)$.

3.2. Example of encryption using a 2-byte unit

In the example demonstrated below, we have considered the case where the unit is of the size two bytes i.e. $w = 16$ and both the plaintext and the password consist of two characters at a time. The example takes a plain text of the length $n = 7$ (Marwa Al-Raisi) and a password of the length $l = 3$ (Imene). The vertices are v_0, v_1, v_2 and v_3, where v_0 is the plaintext and v_3 is the cipher text. Note that X is a point vertex and Y is a line vertex. The system of equations for $n = 7$ is as follows:

$$\begin{aligned} y_0 &= k_0 \\ y_1 &= y_0 \cdot x_0 + x_1 \\ y_2 &= x_0 \cdot y_1 + x_2 \\ y_3 &= y_0 \cdot x_1 + x_3 \\ y_4 &= y_0 \cdot x_2 = x_4 \\ y_5 &= x_0 \cdot y_3 + x_5 \\ y_6 &= x_0 \cdot y_4 + x_6. \end{aligned}$$

Encryption Process

The encryption process starts with the first vertex i.e. the plaintext:

$$v_0 = (x_0, x_1, x_2, x_3, x_4, x_5, x_6) = (19809, 29303, 24864, 16748, 11602, 24937, 29545).$$

Plaintext: Marwa Al-Raisi

char	M	a	r	w	a	space	A	*l*	-	R	a	i	s	i
ASCII	77	97	114	119	97	32	65	108	45	82	97	105	115	105
$GF(2^{16})$ in decimal	19809		29303		24864		16748		11602		24937		29545	
symbol	x_0		x_1		x_2		x_3		x_4		x_5		x_6	

Password: Imene

char	I	m	e	n	e	NULL
ASCII	73	109	101	110	101	0
$GF(2^{16})$ in decimal	18797		25966		25856	
symbol	α_0		α_1		α_2	

To move from v_1 to v_2, we use the first element of the encryption key i.e. $k_1 = 18797$:

$$v_1 = (x_0, x_1, x_2, x_3, x_4, x_5, x_6) = (18797, 8017, 7556, 8947, 34475, 57403, 485).$$

Encryption Key

As mentioned before, the encryption key is constructed in a way such that if the *password* = $(\alpha_0, \alpha_1, \alpha_2, \ldots, \alpha_{l-1})$, then the encryption key $K = (k_0, k_1, k_2, \ldots, k_{l-1})$, where $k_{l-1} = \alpha_{l-1} + x_0$ and x_0 is the first element of the plaintext.

$GF(2^{16})$ in decimal	18797	25966	10337
symbol	k_0	k_1	k_2

$k_2 = \alpha_2 + x_0$
$= 25856 + 19809$
$= 10337$

Similarly, to move from v_2 to v_3, we use the second element of the encryption key i.e.

$$k_2 = 25966 :$$
$$v_2 = (25966, 52660, 21353, 9330, 55046, 54731, 48196).$$

The last walk is from v_3 to v_4 using $k_3 = 25966$

$v_3 = (10337, 54301, 25429, 49398, 57549, 44071, 11772)$, is the cipher text.

3.3. Algorithm implementation and complexity

According to our algorithm, the arithmetic operations in the system of equations (1) are in the finite field $F(2^m)$. For the implementation purpose, a multiplication table of $F(2^m)$ is pre–computed and stored offline in memory; to avoid the expensive cost of online multiplication operations. That is, the cost of multiplication over $F(2^m)$ is simply the cost of accessing a memory location. Now we consider the cost of a one step walk, say from a point vertex to a line vertex with a data file of n units (i.e. 1-byte, 2-bytes, or 4-bytes), and the cost of this walk step includes the cost of computing n line components using a formula for the system of equations (1) of the form:
$y_i - x_i = y_1 \cdot x_{i-2}$ (one addition and one multiplication).

Since the number of walks is equal to l, the length of the password, the cost of the algorithm is linear: $l * n$ (addition and one multiplication). That is, the complexity of the algorithm is $O(l * n)$. The experimental results in the next section confirm the linear complexity of the algorithm. The impact of the unit size on the speed of the algorithm is as follows: increasing the unit size from m=8 to m=16 will result in reduction of the computation cost by half. Increasing the unit size from $m = 8$ to $m = 32$ will result in a speed-up by 4 times.

A very important consequence of Proposition 1 is the simple way to compute the probability to guess the plain data from the cipher data. In fact, Proposition 1 tells us that there is a unique walk (path) starting with the cipher data vertex back to the plain data vertex. The unique way to guess the plain data is to

guess the reverse password. Let us compute the probability to guess the plain data from the cipher data, which is the same probability to guess the cipher data from the plain data. Starting from the plain data vertex, the probability to guess the next line vertex depends on the first password character. There are $q = 2^m$ different possible line vertices. The probability to guess the first step walk is $1/q$. Now the probability to guess the next walk step using the second character is $1/(q-1)$, we remove the possibility to go back to the previous vertex. We do the same reasoning for the other successive step walks. If the password to encrypt the data is of the length k, the probability to guess the message is $pkey = 1/\left(q\,(q-1)^{m-1}\right)$. As an example, for the passwords of the length $l = 10$, $pkey \approx 10^{-25}$. For $l = 20$, $pkey \approx 10^{-49}$. This shows how strong is our algorithm even with short passwords.

The important feature of such encryption is the resistance to attacks when the adversary intercepts the pair plain text and cipher text because the best algorithm of finding the path between given vertices (by Dijkstra, see [**16**]) has complexity $|V|\log(|V|)$ where $|V|$ is the order of the graph, i.e. size of the plain text space (huge).

4. Experimental Measurements

The algorithm is implemented using the C++ language. A readymade library [**17**] of procedures for finite field arithmetic in $F(2^m)$ for $m = 8$, 16 and 32 will be used to perform the necessary operations of multiplication and XOR (addition/subtraction) on a finite field. The library is written in C but it is compatible with C++ as well. It is especially tailored for m equal to 8, 16 and 32, but it is also applicable for any other value of m.

Prior to going into further details, it is important to talk a little about the implementation approach that reveals how data in the input files is converted to the $F(2^m)$ elements. The system accepts various types of data files such as video, image, text and audio. The system reads these files as streams of binary bits into units and directly converts each unit to its decimal representation. However, when dealing with text files and characters, a byte consists of 8-bits and the ASCII code of a character represents it in decimal. To find the polynomial that stands for a particular character, we convert the decimal value of the character to its binary representation. The binary bits correspond to the coefficients of the polynomial in $F(2^8)$. If $F(2^{16})$ is intended to be utilized in the algorithm, we consider a unit of two characters at a time. Similarly for $m = 32$, we divide the data to be encrypted into units of four characters and convert the units to their corresponding polynomial versions in $F(2^{32})$.

For a unit of the size 1-byte (i.e. $m = 8$), the fastest way to perform multiplication is to employ a multiplication table and store this table internally. This table consumes $2(m + 2)$ bytes, so it is only applicable when w is reasonably small. For example, when $m = 8$, this is 256 KB. However, when we select a unit of the size 2-bytes, this multiplication table consumes $2(2m+2)$ bytes and in the case of $m = 16$, this table is 234 bytes which is very large and cannot fit into memory. The proposed solution states that when multiplication tables cannot be employed, the most efficient way to carry out multiplication is to use log and inverse log tables, as described in. The log table consumes $2(m + 2)$ bytes and the inverse log table consumes $3 * 2(m + 2)$ bytes, so when $m = 16$, this is approximately 1 MB of tables which can easily fit into memory. Then we can calculate the product of a and b as:

$$a * b = i \log [\log[a] + \log[b]] .$$

While in the case of a 4-byte unit size (i.e. $m = 32$), it is obvious that the log tables cannot fit into memory (2×1010 bytes). A recommended resolution [**17**] is to create seven tables that are 256 KB each and these tables are used to employ the 32-bit numbers multiplication by breaking them into four eight-bit numbers each, and then performing sixteen multiplications and XORs to calculate the product.

Experiments:

The experimental evaluation of any algorithm is essential to acquire a realistic vision of the resources required by the algorithm. In this section, we will test the execution time of the algorithm upon various sizes of date files and passwords using different unit sizes (i.e. 1-byte, 2-bytes, and 4-bytes) and determine its time complexity in order to measure the efficiency of our algorithm.

Analyzing the algorithm structure shown in the previous section, we expect that as we increase the unit size, performance of the algorithm will improve since the plaintext and the password will be consumed faster. Consequently, the expected execution time to produce the cipher text will relatively decrease. The experiment included running the system on different text files of sizes 1 MB, 5 MB, and 9 MB along with a variety of passwords (l) ranging from 4-bytes to 20-bytes of size. Then, the time (in milliseconds) to encrypt/decrypt is recorded. The experiment was conducted using the machine that has a 2.99 GHz Intel(R) Core(TM)2 Duo CPU and a 1.96 GB of RAM. The results of these runs are shown below.

Note that the above three tables confirm the linear complexity of our algorithms for different values of m.

Unit size = 8-bits

l (bytes)\size of data	1 MB	5 MB	9 MB
4	3297	16453	29609
8	6578	32750	58906
16	13156	67047	120844

Unit size = 16-bits

l (bytes)\size of data	1 MB	5 MB	9 MB
4	1578	7859	14141
8	3156	15781	28297
16	6204	31078	57078

Unit size = 32-bits

l (bytes)\size of data	1 MB	5 MB	9 MB
4	750	3672	6625
8	1469	7328	13172
16	2937	14672	26360

Comparison Results Using Different Values for m

In this experiment we use different test files and a password of the length 20 bytes. The following line graph displays the results obtained by different unit sizes.

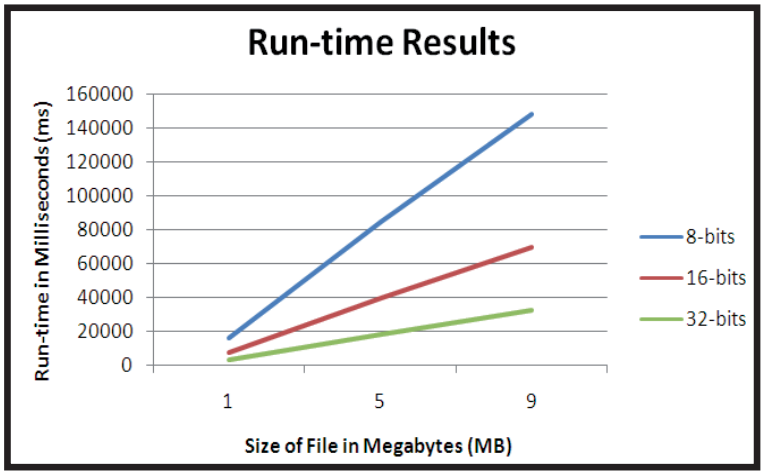

Fig. 2. Comparison of run-time results $m = 8$, 16 and 32.

Fig. 2 shows the unit size impact on the speed of the algorithm, as for a unit of the size 16 bits; there is a gain in speed by approximately 50 percent compared to the results of 8-bit unit size. Likewise, in the case of 32-bit unit size, this increase in speed is observable (4 times faster than the 8-bit unit size). Generally, as the unit size increases, the efficiency of the algorithm increases.

5. Conclusions

We implemented a new variable unit size symmetric stream key dependent cipher based on the algebraic graph using the finite field. We have tested our algorithm and confirmed that it is of linear complexity. Our experimental results have shown that increasing the unit size reduces the run-time cost of the algorithm by 4. However, this requires extra memory to store the multiplication tables for the corresponding finite field operations. Our future work is to extend the capability of the algorithm with a larger unit of the size 64-bit, 128-bit, etc. Consequently, we expect that the algorithm will become more efficient when a larger unit size is employed.

References

[1] Lazebnik F., Ustimenko V., Some Algebraic Constructions of Dense Graphs of Large Girth and of Large Size, DIMACS series in Discrete Mathematics and Theoretical Computer Science 10 (1993): 75.

[2] Kim J. L., Peled U. N., Perepelitsa I., Pless V., Friedland S., Explicit construction of families of LDPC codes with no 4-cycles, Information Theory, IEEE Transactions 50(10) (2004): 2378.

[3] Ustimenko V. A., Coordinatisation of regular tree and its quotients, in "Voronoi's impact on modern science", eds P. Engel and H. Syta, book 2, National Acad. of Sci, Institute of Matematics (1998): 228.

[4] Kotorowicz J., Ustimenko V. A., On the implementation of cryptoalgorithms based on algebraic graphs over some commutative rings, Condenced Matters Physics, Special Issue: Proceedings of the international conferences "Infinite particle systems, Complex systems theory and its application", Kazimerz Dolny, Poland, 2006, 11, 2(54) (2008): 347.

[5] Ustimenko V., Touzene A., CRYPTALL-a System to Encrypt All types of Data, Notices of Kiev Mohyla Academy (2004): 57.

[6] Touzene A., Ustimenko V., Graph Based Private Key Crypto System, International Journal on Computer Research, Nova Science Publisher 13(4) (2006): 12.

[7] Klisowski M., Ustimenko V., On the public keys based on the extremal graphs and digraphs, International Multi–conference on Computer Science and Informational Technology, October 2010, Wisla, Poland, CANA Proceedings.

[8] Wróblewska A., On some properties of graph based public keys, Albanian Journal of Mathematics 2(3) (2008): 229.

[9] Ustimenko V., Algebraic graphs and security of digital communications, Institute of Computer Science, University of Maria Curie Sklodowska in Lublin (2011): 151 (supported by European Social Foundation), available at the UMCS web.

[10] Ustimenko V., CRYPTIM: Graphs as Tools for Symmetric Encryption, In Lecture Notes in Computer Science, Springer 2227 (2002): 278.
[11] Ustimenko V., Graphs with special arcs and Cryptography, Acta Applicandae Mathematicae (1974): 117.
[12] Koblitz N., A Course in Number Theory and Cryptography, Second Edition, Springer (1994).
[13] Koblitz N., Algebraic Aspects of Cryptograph, Springer (1998).
[14] Hasan M. A., Look-Up Table-Based Large Finite Field Multiplication in Memory Constrained Cryptosystems, IEEE Trans. Comp. 49 (7) (2000): 749.
[15] Ustimenko V., Woldar A., Extremal properties of regular and affine generalized polygons as tactical configurations, Europ. J. Com. 24 (2003): 99.
[16] Dijkstra E., Note on two problems in connection with graphs, Num. Math. 1 (1959): 269.
[17] Plank J. (n.d.), Fast Galois Field Arithmetic Library in C/C++. Retrieved October 28 (2009), from The University of Tennessee, College of Art and Science: http://www.cs.utk.edu/ plank/plank/papers/CS-07-593.