School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Random Stream Cipher

Saeed Aghaee

Aug
2007

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 07102
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--07102/--SE

Abstract

Stream ciphers are counted as an important part of symmetric encryption method. Their basic idea comes from One-Time-Pad cipher using XOR operator on the plain text and the key to generate the cipher. The present work brings a new idea in symmetric encryption method, which inherits stream key generation idea from synchronous stream cipher and uses division instead of xoring. The Usage of division to combine the plain text with stream key gives numerous abilities to this method that the most important one is using random factors to produce the ciphers.

Key words

Cryptography, Random stream cipher, symmetric encryption

1. Introduction

In cryptography, a method is called symmetric encryption when the same key is used to encrypt and decrypt the plain text. Stream ciphers are the most considerable topic in symmetric encryption. Stream ciphers, use an initial key as the secret key and iterate the states to produce stream key which is used to encrypt and decrypt a block of data. Afterward, the stream key is combined with the same length block of data to generate the cipher.
Random stream cipher is a method in cryptography bringing random factor to its generated cipher. It means that, one message can correspond to many ciphers which all result in the same message.

The base idea in generating stream keys is the same as current stream cipher methods. Therefore, any current method can be used to generate the stream key. Random stream cipher makes the cipher harder to breakdown regarding the random factor, which increases the uncertainty probability. Later on, random stream cipher will have its own method to generate stream keys.

1.1. Classification

Random stream cipher is classified in stream cipher cryptography, which can be used either as public key or secret key (I discussed only secret key). Random stream cipher can use both synchronous stream cipher and asynchronous stream cipher in order to generate stream key.

In random stream cipher the plain text is divided into blocks and in comparison with other stream cipher blocks length are not fixed and randomly generated. It also replaces the function to combine stream key and plain text, which is usually simple XOR operator, with a method in which the cipher is obtained from the result of dividing the plain text by stream key. Besides, stream block number in a plain text and their length are randomly achieved therefore it results in a strong cipher resistant to usual stream cipher attacks.

1.2. Random stream cipher in general

Random stream cipher uses synchronous stream cipher method in which a stream of digits is generated independently of the plain text and cipher. Random stream cipher is divided into three parts:

1. Key generator part: is responsible to generate a stream key independently from cipher and plain text. This part can be inherited from other stream ciphers.

2. Stream block generator: randomly separate the plain text to the parts called stream blocks.

3. Stream cipher generator: combines the stream key with the stream block of plain text to generate the corresponding stream cipher.

The whole procedures are depicted as follows:

$Q_{i+1}=f(Q_i ,k);$
$r_i=g(Q_i ,k);$
$C_i=h(r_i,b_i);$

r_i *= stream key of state i*
C_i *= cipher corresponding to i-th stream block*
b_i *= i-th stream block*
k=secret key

Where Q_0 is the initial state and can be determined by *K. f ()* is a function that returns the next state from current state and *K. g ()* returns the stream key using *K* and the current state, and finally *h ()* combines stream key with the stream block using random coding method to generate the corresponding cipher.

1.3. Problems and difficulties

The idea of the current report, other than introducing the method; is also to analyze and implement it.

Analyzing the method means describing its behavior when it comes to encrypt and decrypt data and the points becomes clearer when it gets compared with other stream ciphers. Analyzing the method is more based on its generated cipher, because of the fact that the most important property differing random stream cipher with other types of stream cipher, is embedded in its generated ciphers (see chapter 5).

The primary goal of the implementation is to create a simple working application. There are a number of complex problems regarding the implementation needing to be solved, such as, how to divide two numbers in binary when they are bigger than the standard predefined types like integer, double and so forth. Therefore, there is a need to define a special type for binary digits (see chapter 6).

1.4. Report outlines

The current report will describe the three major parts of random stream cipher in detail. Each section is assigned to each part. Chapter 1 will describe the key generator part skipping variable length key generator. Chapter 2 will contain all aspects of stream blocks and describe it in detail. The third chapter gives information about stream cipher and how its format should be look like.

The 4-th chapter will be about random stream cipher analysis describing the differences between the cipher generated by random stream cipher with other methods and at the end it mentions a usual attack on stream cipher and compares it with random stream cipher based on the previous discussion about the cipher generate by this method.

Random stream cipher implementation will be discussed in a separated chapter describing the architecture and the source code in some details. It should be mentioned that the implementation which will be discussed at the end is not optimized, in anther word; the implementation can not be suggestive of the performance of the method.

At the end the conclusion part comes and summarizes the presented report including advantages, disadvantages and differences with other types of stream cipher.

2. Key generator

Stream key is a sequence of 'n' bits, which is determined by the key generator function. If the key generator function generates the same stream key length at each iteration, the function is called fix length key generator otherwise variable key generator. Random stream cipher can use both fix and variable length key generator functions. Later on, it will be discussed that using variable key generator makes cipher stronger to breakdown.

Due to the fact that random stream cipher does not have its own method to generate stream key, so current methods such as rabbit stream cipher, vest stream cipher and so forth are used to undertake generating stream key. In the present work, fixed length stream key methods are discussed.

2.1. Fixed length key generator

There is variety of stream cipher methods working with fixed length stream keys. Each of them has a number of vulnerabilities plus a number of advantages that makes it considerable. For instance, *RC4* (also known as ARC4 or ARCFOUR) is widely used owing to its high speed. Somehow, there are a number of known attacks to break down *RC4*, which can be enumerated as its disadvantages.

As another example, *rabbit stream cipher* [1] is a pretty new method, which was invented in 2003. Rabbit stream cipher is very fast and reliable in comparison with other methods. Besides, the key length which rabbit generates is suitable and can easily be integrated with random stream cipher.

Taking to account that there are no stream cipher method without vulnerability, rabbit stream cipher has been chosen as the fixed length key generator of current implementation of random stream cipher.

Table 2.1 depicts stream ciphers methods and compares them using some criteria such as speed, best known attack and so forth. The table has been taken from reference [2] which is an online dictionary and free encyclopedia. As it can be seen in the table, *rabbit stream cipher* is a pretty new method (it was invented in 2003). By considering *speed* and *best known attacks* columns in the table and making a simple comparison, it is obvious that *rabbit stream cipher* is one of the fastest and most reliable methods.

2.1.1. Rabbit stream cipher in general

The following explanation of rabbit stream cipher is a summarization of reference [1]. Briefly, it uses a 128-bit secret key to initialize the internal state and iterates the system 4 times. Rabbit stream cipher generate 128-bit stream key at each iteration which is a combination of each internal state bits. The size of the internal state is 513 bits divided between eight state variables. The variables are updated by eight coupled non-linear integer valued functions. The counters guarantee that the stream keys will not be repeated during the iterations.

Table 2.1: comparison of Stream ciphers

			(bits)			**Attack**	
Stream Cipher	**Creation Date**	**Speed (cycles/byte)**	**Effective Key-Length**	Initialization vector	**Internal State**	**Best Known**	**Computational Complexity**
A5/1	1989	Voice (Wphone)	54	114	64	Active KPA OR KPA Time-Memory Tradeoff	~2 seconds OR $2^{39.91}$
A5/2	1989	Voice (Wphone)	54	114	64?	Active	4.6 milliseconds
FISH	1993	Quite Fast (Wsoft)	Huge	?	?	Known-plaintext attack	2^{11}
Grain	Pre-2004	Fast	80	64	160	Key-Derivation	2^{43}
HC-256	Pre-2004	4 (WP4)	256	256	65536	?	?
ISAAC	1996	2.375 (W64-bit) - 4.6875 (W32-bit)	8-8288 usually 40-256	N/A	8288	(2006) First-round Weak-Internal-State-Derivation	4.67×10^{1240} (2001)
MUGI	1998-2002	?	128	128	1216	N/A (2002)	$\sim2^{82}$
PANAMA	1998	2	256	128?	1216?	Hash Collisions (2001)	2^{82}
Phelix	Pre-2004	upto 8 (Wx86)	256 + a 128-bit Nonce	128?	?	N/A (2005)	N/A (2005)
Pike	1994	0.9 x FISH (Wsoft)	Huge	?	?	N/A (2004)	N/A (2004)
Py	Pre-2004	2.6	8-2048? usually 40-256?	64	8320	Cryptanalytic Theory (2006)	2^{75}
Rabbit	2003-Feb	3.7(WP3) - 9.7(WARM7)	128	64	512	N/A (2006)	N/A (2006)
RC4	1987	Impressive	8-2048	8	2064	Shamir	2^{13} OR 2^{33}

			usually 40-256			Initial-Bytes Key-Derivation OR KPA	
Salsa20	Pre-2004	8.91 (WG4) - 16.44 (WP4)	256 + a 64-bit Nonce	512	512 + 384 (key+IV+index)	Differential (2005)	N/A (2005)
Scream	2002	4 – 5 (Wsoft)	128 + a 128-bit Nonce	32?	64-bit round function	?	?
SEAL	1997	Very Fast (W32-bit)	?	32?	?	?	?
SNOW	Pre-2003	Very Good (W32-bit)	128 OR 256	32	?	?	?
SOBER-128	2003	?	upto 128	?	?	Message Forge	2^{-6}
SOSEMANUK	Pre-2004	Very Good (W32-bit)	128	128	?	?	?
Trivium	Pre-2004	4 (Wx86) - 8 (WLG)	80	80	288	Brute force attack (2006)	2^{135}
Turing	2000-2003	5.5 (Wx86)	?	160	?	?	?
VEST	2005	42 (WASIC) - 64 (WFPGA)	Variable usually 80-256	Variable usually 80-256	256 – 800	N/A (2006)	N/A (2006)
WAKE	1993	Fast	?	?	8192	CPA & CCA	Vulnerable
Stream Cipher	**Creation Date**	**Speed (cycles/byte)**	**(bits) Effective Key-Length**	Initialization vector	**Internal State**	**Attack Best**	

3. Stream Block

Due to the fact that length of blocks is independent from stream keys, hence, Stream Blocks determination can be done at the first step. A proper block should have the following criteria:

- Blocks are taken from binary digits of the plain text.
- Blocks length should be dividable by 8(if the plain text is not dividable by *8* we can modify it by adding *0* bit to the end of the plain text to make it dividable by *8* and it does not affect the original plain text)
- The first bits of block should be nonzero.
- Blocks length should be generated randomly.
- Every plain text must at least have two stream blocks

3.1. Stream block starting with zero bits

Note that, for zero stream blocks, regardless of the stream key, the cipher would be zero (it can be a weakness in cipher) and for stream blocks that start with zero bits or in another word the stream blocks which their first byte is smaller than 128 in decimal, after decryption, the first zero bits would be lost, because zeros in the left of binary digits are not counted. Hence, the firs bits of stream blocks should not be zero.

The solution is to XOR the first byte of such stream blocks with 255 (*11111111* in binary) and informs another party by modifying the cipher (one bit can be allocated in each stream cipher).

3.2. Number of stream blocks in a plain text

A plain text that has been divided into one stream block is vulnerable when an attacker has access to encryption machine, thus, he can encrypt his own plain text then compare the plain text with the cipher to find the stream key. When the plain text has more than two blocks and its length is big enough, owing to the fact that the stream block are chosen randomly, even if the attacker has the plain text with the cipher he will not be successful to find the stream keys. Therefore, increasing the block numbers and plain text length would result in high uncertainty probability.

Stream blocks length is independent from stream key; however, stream blocks number depends on stream key length despite the fat that it is randomly determined. Let *MRL* be the minimum stream key length that stream key generator function can generate which in fix length key generator it would be constant and in variable key generator that generates stream keys with the length in the rage *[j, k]* (where $j<k$), it would be j. *ML* and *M* are plain text length and the plain text. Therefore:

MRL=min ({len (r)})
ML=len(M)

*2 < SBN < 2*ML/MRL*

Where, SBN is stream block number. The above formula is suggestive of the fact that SBN should be at least 2 and 2*ML/MRL is the maximum value of SBN. Hence, the rang to random SBN; depends on both ML and MRL.

3.3. Plain text normalization

As it is mentioned the maximum value of SBN is *2*ML/MRL*. But consider the case in which ML < MRL, so, the maximum value would be less than 2. In this case the plain text should be normalized by adding E zero byte to the end of the plain text so that the plain text length will be increased. E can be obtained randomly from the following range:

*MRL-ML < E < 2*MRL-ML*

Where MRL-ML is the minimum number of zero bytes that should be added to the end of the plain text to yield *ML>=MRL*. Randomly choosing E would increase uncertainty probability.

3.4. Overall process of generating stream blocks

The first step in random stream cipher is stream blocks determination. Stream block generator should feed back the information such as:

- Stream blocks number
- Stream blocks length
- XOR state
- Plain text normalization (if necessary)

First of all, plain text should be checked for normalization. If the plain text length was less than the minimum stream keys length, adding enough zero bytes to it would normalize the plain text.

The next step is randomly determination of stream blocks number from the range depending on the minimum value of the stream keys. As it was mentioned the range that SBN should be chosen from should be:

*2 < SBN < 2*ML/MRL*

After ward, stream blocks are specified one by one with a random length .Let *LS* be the remaining size of the plain text after taking *i-1* stream blocks. *i-th* stream block length would be randomly taken from the range *[1, LS-1]*.

Eventually, the first bits of stream blocks were zero or in another word, the first byte of stream block was smaller than 128 in decimal, the first byte is xored with a number bigger than 128.

Here is the Stream block generator Pseudo code:

```
random_stream_block_number()

for i→0 to stream_block_number-1
do
random a number from 0 up to data_size
if it is not repeated before
add it to array r[]

sort out array r[]

for I →0 to stream_block_number
do
stream_block[i]= r[i-1]- r[i]
```

First *random_stream_block_number* randomly selects a number in the mentioned range of stream block number and loads *stream_block_number* with that value. Afterward, a number is obtained randomly from 0 up to *data_size* (the plain text size) and it is added to a temporary array if and only if it's not repeated before. In order to find out that the number has not been chosen before a simple binary search is performed on the temporary array *r[]*. After loading *r[]* with not repeated numbers between 0 to *data_size*, it should be sorted by a fast sorting algorithm in order to achieve *stream_block*. Using sorted *r[]*, *stream_block[i]* (i-the stream block size) can be acquired by subtracting *i-1-th* element of *r[]* by its next element.

4. Stream cipher generator

This part is responsible for combing the stream block and stream key and generates the stream cipher. Unlike usual stream cipher the function does not use XOR operation, instead it uses division to generate cipher in the way that stream block is the dividend, stream key is the divisor and finally the quotient will be the cipher.
Considering a plain text divided into *n* stream blocks, the function simply divides each block by the stream key, which is used in decryption as well.

The message is divided into n stream blocks $(b_1, b_2,.., b_n)$. Let $(r_1, r_2,.., r_n)$ be a sequence of stream keys in which r_i is corresponded to b_i. And $(c_1, c_2... c_n)$ is the sequence of generated stream ciphers. Each stream block is combined with its stream key at *i-th* iteration:

$c_i = b_i / r_i$
$h(b_i, r_i) = c_i$

The stream block can be retrieve from the cipher by:

$b_i = c_i * r_i$

4.1. Fraction part

The problem in the formula using to encrypt the stream blocks is usage of division and therefore the result might have fraction part. It is possible to limit the fraction part, for example considering only *n* digits for the fraction part that both parties agree on the value of *n*. But fraction parts do not give an integer most of time:

$b_1=20$
$r=3$
$n=4$

Encrypt: $c_1=20/3=6.6667$
Decrypt: $b_1=6.6666*3=19.9998$

As it can be seen in the above example another party gets 19.9998 after decryption, whereas the actual stream block was 20. At this point, it is feasible to guess the actual number by rounding up the result to the closest integer number. It is better not to have a small *n* as a matter of fact it makes it hard to round up the stream cipher when the stream key and stream block get bigger.

When *n* was chosen properly (not to small) it is guaranteed that another party can obtain the correct message by rounding up the achieved result from decryption.

4.2. Size of fraction part

The size of the fraction should be equal to maximum length of stream keys. It should be constant for every fraction part in the cipher even if the actual fraction part is less than the maximum length of the stream keys.

FPZ= fraction part size
FPZ=max({len(r1),...,len(r2)})

Considering the following division formula, the goal is to have b_0 in the rang [b-1, b] so that it can be rounded up to obtain *b*:

$a = b/c$
$b_0 = a*c$
$len(a) = n$
$a = i\ .\ f$

Where, *i* is the integer part of the quotient, *f* is the fraction part and *len (x)* function returns the size of *x*. if *len(f) < len(c),* so *b* can be gained without needing b_0 to be rounded up. Assume that len (f) is infinite or *len(f) > len(c).* It can be shown that counting only *len(c)-1* first digits of 'f' would be sufficient to obtain b_0 in the rang *[b-1, b].*

Lemma 1: the *j-th* fraction part digit multiplying by a number with size *n* results in a number with size *n-j* (in binary). *K* is the number in the rang *(0, 1)* with only a *1* bit at the *j-th* place of the fraction part.

$0 =< a <2^{n+1}$
$k= 2^{-j}$
$a * k= b$
$0=<b< 2^{n-j+1} \rightarrow len(b) = n-j$

Using the above lemma and replacing *n-1* by *j*, it can be concluded that having *n-1* digits for quotient fraction part after multiplying it with the dividend will result in a number with the same size as the dividend, in another world, it gives a number in the range *[b-1, b],* where *b* is the dividend.

4.3. Stream cipher format

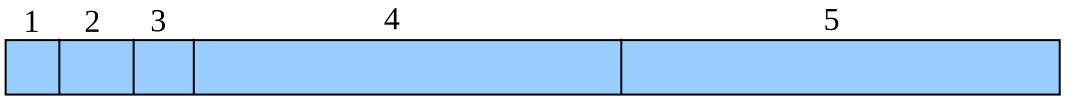
Generated cipher should contain all information that another party has to know to decrypt it. The information to be contained is as following:

1. Xor state
2. Integer part
3. Fraction part

Xor state can be shown by allocating a first bit of stream cipher to it. Bit *1* can be suggestive of a xored stream block and in the contrast, bit *0* shows that the stream block has not been xored.

Integer part can be represented immediately after the xor state bit by allocation some bytes as its offset. Besides, there is a need to have and offset to determine the end of integer part so that the integer part can end in the middle of a byte.

Due to the fact that fraction parts have the same length (maximum length of stream keys) and both parties aware of that, hence, there is no need to have an offset to show its length and can follow the integer part to complete the stream cipher.



1. *Xor state*
2. *Integer part offset*
3. *Integer part end offset*
4. *Fraction part*

Figure 4.1: stream cipher format

Figure 4.1 shows the structure of the cipher. Each part in the rectangle which labeled by a number is allocated to a piece of information such as xor state, integer part offset and so forth.

5. Analysis of Random stream cipher

In random stream cipher, stream blocks play an important role. As it was mentioned, Stream blocks number and stream blocks length are both determined randomly, and this random factor results in high uncertainty probability to obtain the stream key having the plain text and the cipher text. In another word, chosen plain text and chosen cipher text attacks would be disabled for random stream cipher. In addition to variety of cipher having one message, the generated ciphers can differ in their length too. Here are two different ciphers for one plain text:

Plain text: Hello Random Stream Cipher!

Cipher number 1: ##   #Žèª€#òÝJUA•„#Š´#   ##@##   #   ##jñ#vÁîã

Cipher number 2: ##   #   #;¢ª#òÝJUA”##   #“À"C»i›¸Ò¾úJ"ueŸp:

Note that, the ciphers differ with each other whereas the plain text is the same.

5.1. Differences with other types of stream cipher

The differences between random stream cipher and current types of stream cipher are mentioned here.

1. The most important difference is that in usual stream cipher cryptography, the function to combine stream key and plain text uses XOR operator. But in random stream cipher, the function uses division on the plain text blocks and the corresponding stream keys.

2. In usual stream cipher cryptography, length of key should be equal to length of plain text blocks, but in random stream cipher, it does not matter, and key length can be varied from plain text blocks.

3. In usual stream cipher cryptography, length of cipher text is equal to its plain text, but in random stream cipher, the cipher is different from its plain text.

4. In random stream cipher, for the same plain text and secret key, there are many ciphers to be corresponded to. In another word, the cipher is obtained randomly from a wide range depending on the plain text. It is the thing that does not exist in current cryptography methods.

The first one, referring to the difference in plain text and stream key combination method, is suggestive of the fact that random stream cipher needs more calculation than other methods using xoring method. Hence, random stream cipher works slower than other stream cipher types.

The rest of the differences all refers to the generated cipher and the random factor that causes the cipher to be obtained randomly (but in a specific wide range depending on the plain text). Focusing on the random cipher will clarify the differences.

5.2. Random cipher

In random stream cipher, randomly generating stream blocks results in different ciphers with the same secret key. It is simply because of the fact that, changing the dividend value results in changing the quotient by having the same divisor, here the dividend is the stream block, the quotient is the stream cipher and the divisor is the stream key. Therefore, random factor affects the cipher when it comes to separate the stream blocks, which is done randomly.

c= b / r
r is constant, b is variable $\rightarrow$ *c is variable*

Let's determine how many different ciphers can be generated for a plain text with a constant key. Consider that fixed length key generator is used and the length of stream block is 128-bit. *M* is the plain text with the length *N*. If *K* is the variable containing the stream blocks number; therefore for every *N* we have the following equation:

$b_1+b_2+...+b_k=N$

Where b_i is the i-th stream block size.

By counting the possible options to make the equation, we get:

$$\binom{N-1}{K-1}$$

We know *K* can be any number between 2 and 2**N*/16; therefore, the total different stream ciphers that can be generated would be:

$$\sum_{I=2}^{2\cdot\frac{N}{16}}\binom{N-1}{I-1}$$

The above formulas are suggestive of the fact that increasing the plain text length and stream blocks number result in a bigger rang from which the cipher is obtained.

As it can be seen the random factor does not let the cipher expose any information about the plain text. In another word, the cipher can not be corresponded to its plain text regarding the fact that the cipher for the plain text is chosen randomly from a wide range. This property of ransom stream cipher brings numerous advantages. To clarify the point, a usual attack on stream cipher is considered.

5.3. Substitution attack on random cipher

Current information regarding substitution attack has been collected from reference [3]. The most common attack on stream cipher is substitution attack. In this attack an adversary knows the plain text or a portion of that. The adversary does not know the key and he tries to alter the cipher to see what will be changed. For instance, supposed that a portion of the plain text is $1000 and the adversary knows that. He wants to change this portion to something that he wishes, for example $1111. He can simply xor the cipher with $1000 xor $1111 to achieve his result. Let *k* be the used key, so what we have is:

K xor $1000 xor $1000 xor $1111 = K xor$1111

So another party will get $1111 after decrypting the cipher.

In usual stream ciphers methods, each byte in the cipher is corresponded to a byte in the plain text and the only thing that is unknown is the cipher, which has been xored with the plain text byte. Hence, if the adversary knows a portion of the plain text he will know where to find it in the cipher. But, the cipher that has been generated by random stream cipher does not have any common thing with its plain text so knowing a portion or even the whole plain text will not be useful, because there is a high uncertainty probability (as it was calculated) to find out in which stream block and with what offset (from the beginning of that stream block) the portion is located.

6. Random stream cipher implementation

This chapter has a general overview on the implementation of random stream cipher. Some parts of code related to rabbit stream cipher, are taken from reference [2]. Besides, plain text normalization (as it was mentioned in 3.3.) has not been implemented. The source code has been coded by C++ programming language.

6.1. Rsc.cpp

This source file contains the three important parts of random stream cipher, which are: key generator, stream block generator and encryption/decryption part.

6.1.1. Key generator part

In the current implementation (see appendix a), rabbit stream cipher has been used as the fixed length key generator. The functions responsible to generate the key stream are as followings:

- *Key_setup:* initializes the secret key and makes the first internal state. Afterward, it iterates the internal state 4 times. This function should be invoked before encryption or decryption of any data.

- *Next_state:* generates next internal state using eight internal variables and counters. Then It produces the next 128-bit stream cipher from the current internal state.

After initializing the secret key and iterating the internal state for 4 times, stream blocks should be determined. First of all stream block number should be obtained randomly, and then based on the stream block number, stream blocks are separated randomly from the plain text.

6.1.2. Stream blocks generator part

The second part is stream blocks generator. The functions to determine stream block number and stream blocks are:

- *stream_block_number_init()*: which select a number randomly between the mentioned rang for stream blocks number (see chapter 3.2).

- *stream_block_init()*: after initializing the stream block number this function separate stream blocks from the plain text using randomly selected size for each (too read the complete description see chapter 3.4)

6.1.3. Encryption and decryption part

The current implementation includes a function called *encrypt()* (see appendix A). This function first initializes the stream blocks, and then combines each stream block with the corresponding stream key to generate the stream cipher. The function receives one input stream as the plain text and one output stream to contain the cipher.

Function *encrypt()*, before combining the stream key with the stream block, determines the xor state for that stream block. In order to do that it uses *xor_state()* function which receives the first byte of the stream block as a reference. It modifies the first byte by specifying whether or not it gets xored and then xoring it by 255 (in decimal).

In order to encrypt a cipher, function *decrypt()* is used. Like *encrypt()* function, this function receives two streams: an input stream for the cipher and an output stream for the decryption result. The function reads the cipher in order to obtain information such as xore state integer part and fraction part. After ward it decrypts the stream cipher by multiplying the stream key with the stream cipher.

6.2. Bit_vector.cpp

This class inherits from standard vector class of *C++* library. Bit vectors are shown by Boolean vectors so, *false* would be equal to *bit 0* and *true* would be suggestive of *bit 1*. The aim of this class is to provide *rsc.cpp* with:

- This functionality to be able to divide or multiply two binary numbers with a **nearly unlimited size**. It is necessary, owing the fact that stream block are determined randomly and regarding the size of the plain text, stream blocks as binary numbers might be very large.

- Creating an instance of *bvector* using a byte array. It is necessary owing to the fact that *bvector* functions are just working with *bvector* instances.

- Implement binary division so that encrypt function can use it to divide the stream block by the stream key.

- Implementing binary multiplication to be used by decrypt function.

- Converting *bvector* to *byte* in order to have the result in bytes.

6.3. Array_util.cpp

Array_util.cpp contains two simple functions, one for sorting out an array and another for searching a key in an array.

Searching function uses insertion sort algorithm. It was used owing to the fact that insertion sort works well with small arrays.

7. Conclusion

The aim of the presented work was to introduce a new method replacing by one-time-pad cipher which is used in current stream ciphers that is xoring the key with the plain text to achieve the corresponding cipher. The method was described and the ciphers that are generated by this method have been analyzed and discussed (see chapter 5). Finally, the structure of random stream cipher implementation was explained in detail (see chapter 6) along with the source code (see appendix A).

To sum up, the current work that was presented is the initial point for random stream cipher to be developed in the future. There are a number of areas in which there is possibilities to be improved. The aim is to implement random stream cipher faster and more secure.

The implementation, which is presented in this paper (see appendix A), is not the optimized one and could be more efficient and faster. It is just to show the ability of the method and be suggestive of a visual perspective of that. Therefore, it is not a proper sample to test the performance of random stream cipher.

7.1. Advantages and disadvantages

The basic idea which is dividing the stream key with the plain text to generate the cipher, frees the plain text and the cipher to have different lengths, the property that other stream ciphers do not have it. This property gives the method the ability to break down the plain text into a number of stream blocks and randomly select their sizes that results in a different cipher in each time the plain text is encrypted, in other words, a randomly generated cipher.

As it was mentioned, having a random cipher, with the same plain text and stream key, is the most important property of this method which results in failing the current cipher or plain text attacks. For instance, chosen-plain text attack or chosen-cipher attack will be failed in random stream cipher because of the fact that there are many ciphers to be corresponded to a plain text and vise versa. Other than the resistance of random stream cipher against current attacks on stream ciphers, based on Mathematics, it can be said that random stream cipher is very hard to break down, owing to the fact that the cipher is generated randomly and does not give any information about its plain text.

Despite the fact that the presented implementation is not a proper candidate to be suggestive of random stream cipher performance, random stream cipher consumes more resources and memory in comparison with other stream cipher methods. As a result, random stream cipher comparing with other stream ciphers encrypts and decrypts data with lower speed. Taking to account that sometimes we are about to prefer security rather than speed for a method.

7.2. Usability

Based on the above information about advantages and disadvantages of random stream cipher, it can be concluded that random stream cipher is more secure and reliable; however, it runs slower than other stream ciphers. So, random stream cipher can be a good choice for local small networks like LANs (local area networks) that are in need of a secure and reliable communication. Examples could be military networks, Universities and so forth.

8. References

[1] **Authors**: Martin Boesgaard, Mette Vesterager,
Thomas Pedersen, Jesper Christiansen, and Ove Scavenius

Title: Rabbit: A New High-Performance Stream Cipher

URL: http://www.cryptico.com/Files/filer/rabbit_fse.pdf

[2] http://www.answers.com/topic/stream-cipher

[3] http://en.wikipedia.org/wiki/Stream_cipher_attack

Appendix A

A.1. Random stream cipher source code

The functions which have been taken from *Rabbit: A New High-Performance Stream Cipher* (reference #2), are labeled by *//reference [2]* at the beginning before their definitions. The code, as it was mentioned, contains several separated source files.

Rsc.h

```
#ifndef _RSC_H_
#define _RSC_H_

#include <cstdlib>
#include <cstddef>
#include <fstream>
#include "bit_vector.h"
#include "array_util.h"

//reference [2], some code have been added to make it
//compatible with random stream cipher
using namespace std;

#define MSL 16

// Structure to store the instance data (internal state)
typedef struct
{
unsigned int x[8];
unsigned int c[8];
unsigned int carry;
unsigned char sk[16];
bool xored;
} t_instance;

void stream_block_number_init(size_t data_size);
void stream_block_init(size_t data_size);
void key_setup(t_instance *p_instance,  unsigned char *p_key);
void initialize(t_instance *p_instance, unsigned char *p_key);
void xor_state(t_instance *p_instance,unsigned char &first_byte);
void encrypt(t_instance *p_instance, ifstream &p_src,ofstream &p_dest, size_t data_size);
void decrypt(t_instance *p_instance, ifstream &p_src,ofstream &p_dest, size_t data_size);
#endif
```

Rsc.cpp

```
#include "rsc.h"

int stream_block_number;
unsigned int *stream_block_size;

// generate stream block number
/*++++++++++++++++++++++++++++++++++++++++++++++++++*/
void stream_block_number_init(size_t data_size)
{
int max=2*(int)ceil((unsigned int)data_size/MSL);
int min=(int)ceil((unsigned int)data_size/(MSL*2))+2;

if (max<=min) max=min+1;
srand ( time(NULL) );

stream_block_number=(rand()%(max-min))+min;
}

// generate stream blocks
/*++++++++++++++++++++++++++++++++++++++++++++++++++*/
void stream_block_init(size_t data_size)
{
register int i,t;
unsigned int *r;
int xored;

stream_block_number_init(data_size);

stream_block_size=(unsigned int*)malloc(sizeof(unsigned int)*stream_block_number);
r=(unsigned int*)malloc(sizeof(unsigned int)*(stream_block_number-1));

srand ( time(NULL) );

for(i=0;i<stream_block_number-1;i++)
{
t=rand()%(data_size-1);
if(repeated(r,0,i,t))
i--;
else
r[i]=t;
}

//sort the array
insertion_sort(&r,0,stream_block_number-1);


for(i=0;i<stream_block_number;i++)
{
if(i==0)
stream_block_size[i]=r[i]+1;
else if(i==stream_block_number-1)
stream_block_size[i]=data_size-r[i-1];
else
stream_block_size[i]=r[i]-r[i-1];
}
}

// Square a 32-bit number to obtain the 64-bit result and return
// the upper 32 bit XOR the lower 32 bit
//reference [2]
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
unsigned int g_func(unsigned int x)
{
// Construct high and low argument for squaring
unsigned int a = x&0xFFFF;
unsigned int b = x>>16;

// Calculate high and low result of squaring
unsigned int h = ((((a*a)>>17) + (a*b))>>15) + b*b;
unsigned int l = x*x;
// Return high XOR low;
return h^l;
}


// Calculate the next internal state
//reference [2]
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void next_state(t_instance *p_instance)
{
// Temporary data
unsigned int g[8], c_old[8], i;
// Save old counter values
for (i=0; i<8; i++)
c_old[i] = p_instance->c[i];
// Calculate new counter values
p_instance->c[0] += 0x4D34D34D + p_instance->carry;
```

```
p_instance->c[1] += 0xD34D34D3 + (p_instance->c[0] < c_old[0]);
p_instance->c[2] += 0x34D34D34 + (p_instance->c[1] < c_old[1]);
p_instance->c[3] += 0x4D34D34D + (p_instance->c[2] < c_old[2]);
p_instance->c[4] += 0xD34D34D3 + (p_instance->c[3] < c_old[3]);
p_instance->c[5] += 0x34D34D34 + (p_instance->c[4] < c_old[4]);
p_instance->c[6] += 0x4D34D34D + (p_instance->c[5] < c_old[5]);
p_instance->c[7] += 0xD34D34D3 + (p_instance->c[6] < c_old[6]);
p_instance->carry = (p_instance->c[7] < c_old[7]);
// Calculate the g-functions
for (i=0;i<8;i++)
g[i] = g_func(p_instance->x[i] + p_instance->c[i]);
// Calculate new state values
p_instance->x[0] = g[0] + _rotl(g[7],16) + _rotl(g[6],16);
p_instance->x[1] = g[1] + _rotl(g[0], 8) + g[7];
p_instance->x[2] = g[2] + _rotl(g[1],16) + _rotl(g[0],16);
p_instance->x[3] = g[3] + _rotl(g[2], 8) + g[1];
p_instance->x[4] = g[4] + _rotl(g[3],16) + _rotl(g[2],16);
p_instance->x[5] = g[5] + _rotl(g[4], 8) + g[3];
p_instance->x[6] = g[6] + _rotl(g[5],16) + _rotl(g[4],16);
p_instance->x[7] = g[7] + _rotl(g[6], 8) + g[5];
//calculate new stream key
*(unsigned int*)(p_instance->sk+ 0) =p_instance->x[0] ^(p_instance->x[5]>>16) ^
(p_instance->x[3]<<16);
*(unsigned int*)(p_instance->sk+ 4) = p_instance->x[2] ^(p_instance->x[7]>>16) ^
(p_instance->x[5]<<16);
*(unsigned int*)(p_instance->sk+ 8) = p_instance->x[4] ^(p_instance->x[1]>>16) ^
(p_instance->x[7]<<16);
*(unsigned int*)(p_instance->sk+12) =p_instance->x[6] ^(p_instance->x[3]>>16) ^
(p_instance->x[1]<<16);
}


// key_setup
//reference [2]
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void key_setup(t_instance *p_instance,  unsigned char *p_key)
{
// Temporary data
unsigned int k0, k1, k2, k3, i;
// Generate four subkeys
k0 = *(unsigned int*)(p_key+ 0);
k1 = *(unsigned int*)(p_key+ 4);
k2 = *(unsigned int*)(p_key+ 8);
k3 = *(unsigned int*)(p_key+12);
// Generate initial state variables
p_instance->x[0] = k0;
p_instance->x[2] = k1;
```

```
p_instance->x[4] = k2;
p_instance->x[6] = k3;
p_instance->x[1] = (k3<<16) | (k2>>16);
p_instance->x[3] = (k0<<16) | (k3>>16);
p_instance->x[5] = (k1<<16) | (k0>>16);
p_instance->x[7] = (k2<<16) | (k1>>16);
// Generate initial counter values
p_instance->c[0] = _rotl(k2,16);
p_instance->c[2] = _rotl(k3,16);
p_instance->c[4] = _rotl(k0,16);
p_instance->c[6] = _rotl(k1,16);
p_instance->c[1] = (k0&0xFFFF0000) | (k1&0xFFFF);
p_instance->c[3] = (k1&0xFFFF0000) | (k2&0xFFFF);
p_instance->c[5] = (k2&0xFFFF0000) | (k3&0xFFFF);
p_instance->c[7] = (k3&0xFFFF0000) | (k0&0xFFFF);
// Reset carry flag
p_instance->carry = 0;
// Iterate the system four times
for (i=0;i<4;i++)
next_state(p_instance);
// Modify the counters
for (i=0;i<8;i++)
p_instance->c[(i+4)&0x7] ^= p_instance->x[i];
}

//initialize relative stream cipher for decryption and encryption
void initialize(t_instance *p_instance,  unsigned char *p_key)
{
//initialize the key
key_setup(p_instance,p_key);
}

//check the first byte of each stream block for xoring
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void xor_state(t_instance *p_instance,unsigned char& first_byte)
{
unsigned char FNSK=255;

if(first_byte>=128)
{
p_instance->xored=false;
}
else if(first_byte<128)
{
p_instance->xored=true;
first_byte=FNSK ^first_byte;
}
```

```
}


// Encrypt data
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void encrypt(t_instance *p_instance, ifstream &p_src,
ofstream &p_dest, size_t data_size)
{
unsigned long int i;
streampos  j=0,g=0;
unsigned int offset,k;
bvector *sbb, //stream block bit vector
*skb, //stream key bit vector
scib,  //stream cipher bit vector (int part)
scfb,
decb;  //stream cipher bit vector (fraction part)
unsigned char xored,end_offset,*buffer;

//initialize the stream blocks
stream_block_init(data_size);

srand ( time(NULL) );

for (i=0; i<stream_block_number; i++)
{
//go to the next state
next_state(p_instance);

//create stream block
buffer=(unsigned char*)malloc(stream_block_size[i]);
p_src.seekg(g);
p_src.read((char *)buffer,stream_block_size[i]);
//determine xor state
xor_state(p_instance,buffer[0]);
sbb=new bvector(buffer,stream_block_size[i],0);

//create stream key
skb=new bvector(p_instance->sk,16,0);

//encrypt the data
sbb->binary_division(*skb,scib,scfb,MSL*8);

//transfer the results to the out stream
//xor state
p_dest.seekp(j);j+=1;
xored=(p_instance->xored) ? 1:0;
p_dest.write((char *)&xored,1);
//integer part offset
scib.to_byte(NULL,offset,end_offset);
p_dest.seekp(j);
p_dest.write((char *)&offset,sizeof(offset));
j+=sizeof(offset);

//end offset
p_dest.seekp(j);j+=1;
p_dest.write((char *)&end_offset,1);
//integer part
buffer=(unsigned char*)malloc(offset);
scib.to_byte(buffer,offset,end_offset);
p_dest.seekp(j);
p_dest.write((char *)buffer,offset);
j+=offset;
//fraction part
buffer=(unsigned char*)malloc(16);
scfb.to_byte(buffer,k,end_offset);
p_dest.seekp(j);
p_dest.write((char *)buffer,16);
j+=16;

free(buffer);

//increase the pointer
g+=stream_block_size[i];

//print out
sbb->binary_print();
cout<<"===================================================";
}
}


// Decrypt data
/*++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void decrypt(t_instance *p_instance, ifstream &p_src,
ofstream &p_dest, size_t data_size)
{
streampos  i=0,j=0;
unsigned int offset;
bvector *skb,
*scib,  //stream cipher bit vector (int part)
*scfb,
sbb;
unsigned char xored,end_offset,*buffer;
```

```
while (i<data_size)
{
next_state(p_instance);

skb=new bvector(p_instance->sk,16,0);

//retrieve xored state
buffer=(unsigned char*)malloc(1);
p_src.seekg(i);i+=1;
p_src.read((char *)buffer,1);
p_instance->xored=(*buffer==1)?true:false;

//retrieve int part offset
p_src.seekg(i);
p_src.read((char*)&offset,sizeof(offset));
i+=sizeof(offset);

//retrieve int part end offset
buffer=(unsigned char*)malloc(1);
p_src.seekg(i);i+=1;
p_src.read((char *)buffer,1);
end_offset=*(buffer);

//create int part bit vector
buffer=(unsigned char*)malloc(offset);
p_src.seekg(i);
p_src.read((char *)buffer,offset);
i+=offset;
scib=new bvector(buffer,(size_t)offset,end_offset);

//creat fraction part bit vector
buffer=(unsigned char*)malloc(16);
p_src.seekg(i);
p_src.read((char *)buffer,16);
i+=16;
scfb=new bvector(buffer,16,0);

//decrypt the cipher
skb->binary_multipilication(*scib,*scfb,sbb);

//transfer the results to the out stream
sbb.to_byte(NULL,offset,end_offset);
buffer=(unsigned char*)malloc(sizeof(unsigned char)*offset);
sbb.to_byte(buffer,offset,end_offset);
//apply xor state
if(p_instance-xored) buffer[0]=buffer[0]^255;
p_dest.seekp(j);
p_dest.write((char *)buffer,offset);
j+=offset;

free(buffer);
sbb.binary_print();
cout<<"==================================================";
}
}
```

---

bit_vector.h

```
#ifndef _bit_vector_h_
#define _bit_vector_h_

#include<iostream>
#include <vector>
#include <cmath>
using namespace std;

class bvector : public vector<bool>
{
public:
bvector() : vector<bool>(){};
bvector(size_t num,bool init_value=true) : vector<bool>(num,init_value) {};
bvector(unsigned char* data,size_t data_size,unsigned char end_offset);
int binary_compare(bvector second,size_t start_point1,size_t start_point2);
void binary_add(bvector second,bvector& result,bool ignore_carry);
void binary_subtract(bvector second,bvector& result);
void binary_division(bvector divisor,bvector& int_result,
bvector& frac_result,size_t frac_size);
void bvector::binary_multipilication(bvector int_part,bvector frac_part,bvector &result);
void binary_print();
void to_byte(unsigned char* bytes,size_t & data_size,unsigned char& end_offset);
void bvector::binary_trim();
};

#endif
```

---

bit_vector.cpp

```
#include "bit_vector.h"
```

```
//bvector constructor from bytes array
/*++++++++++++++++++++++++++++++++++++++++++++++++++*/
bvector::bvector(unsigned char* data,size_t data_size,unsigned char end_offset):
vector<bool>(data_size*8)
{
register unsigned int i,iter;
register int j,k;
unsigned char number,remainder;

for(i=0;i<data_size;i++)
{
number=data[i];
iter=i*8;

for(j=7;j>=0;j--)
{
if(number < 1)
(*this)[iter+j]=false;
else
{
remainder = number%2;
number=number >> 1;
if(remainder==1)
(*this)[iter+j]=true;
else
(*this)[iter+j]=false;
}
}
}

if(end_offset>0) this->resize(this->size()-end_offset);
}


/*++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::binary_trim()
{
while(!this->front())
{
if (this->size()==1)
return;
this->erase(this->begin());
}
}
```

```
//print out the bit vector
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::binary_print()
{
register unsigned int i;
size_t msize=this->size();

cout<<endl;
for(i=0; i< msize; i++ )
if(this->at(i)) cout<<"1"; else cout<<"0";
cout<<endl;
}

//compare two bit vector
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
int bvector::binary_compare(bvector second,size_t start_point1,size_t start_point2)
{
register unsigned int i=start_point1,j=start_point2;
size_t f_size,
s_size;

this->binary_trim();
second.binary_trim();

f_size=this->size();
s_size=second.size();

if((f_size)>(s_size))
return 1;
else if((f_size)<(s_size))
return -1;
else
while(i<f_size && j<s_size)
if((bool)this->at(i) && !(bool)second[j])
return 1;
else if(!(bool)this->at(i) && (bool)second[j])
return -1;
else
{i++;j++;}

return 0;
}

//add to bit vector (ignore the last carry)
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::binary_add(bvector second,bvector& result,bool ignore_carry)
{
```

```
size_t res_size;
register unsigned int i1,i2,ires;
bool carry=false,res;

res_size=this->size()>second.size() ? this->size() : second.size();
result.resize(res_size);
ires=res_size;
i1=this->size();
i2=second.size();

while(i1>0 || i2>0)
{
if(i1>0 && i2<=0)
{
res=(bool)this->at(i1-1) ^ carry;
carry = ((bool)this->at(i1-1) && carry);
result[ires-1]=res;
ires--; i1--;
}
else if(i2>0 && i1<=0)
{
res=(bool)second.at(i2-1) ^ carry;
carry = ((bool)second.at(i2-1) && carry);
result[ires-1]=res;
ires--; i2--;
}
else
{
res=(bool)this->at(i1-1) ^ (bool)second.at(i2-1) ^ carry;
carry = ((bool)this->at(i1-1) && (bool)second.at(i2-1)) ||
((bool)second.at(i2-1)&& carry) ||((bool)this->at(i1-1) && carry);
result[ires-1]=res;
ires--; i1--; i2--;
}
}

if(!ignore_carry && carry)
{
result.resize(res_size+1);
for(i1=result.size()-1;i1>0;i1--)
result[i1]=result[i1-1];
result[0]=true;
}
}


//subtract two bit vector(does not work in the case
//the first bit vector is smaller than the second one)
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::binary_subtract(bvector second,bvector& result)
{
register unsigned int j=second.size(),i;
bvector nsecond,
one(1,true),
two_comp;

if(this->size() - second.size()<0)
return; /*it does not return numbers less than 0*/

i=this->size() - second.size();
nsecond.resize(this->size(),true);

/*2's complement*/

while(j>0)
{
nsecond[j+i-1]=!(bool)second[j-1];
j--;
}
nsecond.binary_add(one,two_comp,true);

/*add 2's complement to the first number*/
this->binary_add(two_comp,result,true);
}

//divide two bit vector and calculate its fraction part
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::binary_division(bvector divisor,bvector& int_result,
bvector& frac_result,size_t frac_size)
{
divisor.binary_trim(); //remove extra 0 from its begining

/*binary division using shift and subtract*/
register unsigned int id,iq;
size_t divisor_size=divisor.size(),
dividend_size=this->size();
bvector remainder;
bvector quotient(dividend_size,false);
bvector frac_quotient(frac_size,false);
bvector zero(1,false);
bvector temp;

//integer part
remainder=(*this);
```

```
remainder.resize(divisor_size-1);

id=divisor_size-1;
iq=1;
while(id<dividend_size)
{
remainder.resize(remainder.size()+1,this->at(id++));

if(remainder.binary_compare(divisor,0,0)>=0)
{
remainder.binary_subtract(divisor,temp);
remainder=temp;
quotient[iq++]=true;
}
else
quotient[iq++]=false;

}
if(iq==1) remainder=(*this);
// integer part result
quotient.resize(iq);
int_result=quotient;
int_result.binary_trim();

//fraction part
if(remainder.binary_compare(zero,0,0)==1)
{
iq=0;
while(iq<frac_size)
{
remainder.resize(remainder.size()+1,false);

if(remainder.binary_compare(divisor,0,0)>=0)
{
remainder.binary_subtract(divisor,temp);
remainder=temp;
frac_quotient[iq++]=true;
}
else
frac_quotient[iq++]=false;
}
}
else
frac_quotient.clear(); //remove all elements from the vector

//fraction part result
frac_result=frac_quotient;
```

```
}
//convert bit vector to byte array
/*+++++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::binary_multipilication(bvector int_part,
bvector frac_part,bvector &result)
{
this->binary_trim();
int_part.binary_trim();

register unsigned int j;
size_t frac_size=frac_part.size(),
int_size=int_part.size();

bvector instance_res=*this,
temp,
res(1,false),
one(1,true),
zero(1,false);


//multiply the fraction part
if((bool)frac_part[frac_size-1])
res=instance_res;

for(j=frac_size-1;j>0;j--)
{
instance_res.resize(instance_res.size()+1,false);

if((bool)frac_part[j-1])
{
res.binary_add(instance_res,temp,false);
res=temp;
}
}

//multiply the integer part
for(j=int_size;j>0;j--)
{
instance_res.resize(instance_res.size()+1,false);

if((bool)int_part[j-1])
{
res.binary_add(instance_res,temp,false);
res=temp;
}
}
```

```
//round up the result and discard the fraction part
if(res.binary_compare(zero,res.size()-frac_size,0)==1)
{
res.resize(res.size()-frac_size);
res.binary_add(one,result,false);
}
else
{
res.resize(res.size()-frac_size);
result=res;
}

result.binary_trim();
}
//convert bit vector to byte array
/*++++++++++++++++++++++++++++++++++++++++++++++++++++*/
void bvector::to_byte(unsigned char* bytes,size_t & data_size,unsigned char& end_offset)
{
size_t t_size=this->size();
unsigned char number,m;
register unsigned int i,iter,l;
register int j,k;

data_size=(size_t)ceil(t_size/8.0);
end_offset=(data_size*8)-t_size;

if (bytes==NULL) return;

for(i=0;i<data_size;i++)
{
iter=i*8;

l=t_size-iter;
if (l>=8) l=8;
number=0;
for(j=0;j<l;j++)
if(this->at(iter+j))
{
m=1;
for(k=0;k<7-j;k++)
m*=2;
number+=m;
}
bytes[i]=number;
}
}
```

array_util.h

```
#ifndef _ARRAY_UTIL_H_
#define _ARRAY_UTIL_H_

void swap(unsigned int & a, unsigned int & b);
void insertion_sort(unsigned int ** arr,  int min,  int max);
bool repeated(unsigned int *arr,  int first,  int last, unsigned int key);

#endif
```

array_util.cpp

```
//swap two element
void swap(unsigned int & a, unsigned int & b) {
unsigned int temp = a;
a = b;
b = temp;
}

//insertion sort to sort out an array
void insertion_sort(unsigned int ** arr, int min, int max) {
unsigned int temp, smallest;
smallest = min;
for (int a = min; a < max; a++) {
if ((*arr)[a] < (*arr)[smallest])
smallest = a;
}
swap((*arr)[min], (*arr)[smallest]);

for (int i = min + 1; i < max; i++) {
temp = (*arr)[i];
int j = i - 1;
while (temp < (*arr)[j]) {
(*arr)[j+1] = (*arr)[j];
j--;
}
(*arr)[j+1] = temp;
}
}

//simple linear search in an array
bool repeated(unsigned int * arr, int first, int last, unsigned int key)
```

```
{
unsigned int i;

for(i=first;i<last;i++)
if(key==arr[i])
return true;

return false;
}
```