Journal of Computer Science and Technology Studies (JCSTS))
ISSN: 2709-104X
DOI: 10.32996/jcsts
Journal Homepage: www.al-kindipublisher.com/index.php/jcsts

Parallel Implementation of RC6 Algorithm

Artan Berisha[1] ✉ and Hektor Kastrati[2]
[12]*Department of Mathematics, University of Prishtina, Faculty of Mathematics and Natural Sciences, Kosovo*
✉**Corresponding Author**:Artan Berisha, **E-mail**: artan.berisha@uni-pr.edu

ARTICLE INFORMATION

Received: February03, 2021
Accepted: June14, 2021
Volume:3
Issue: 2
DOI: 10.32996/jcsts.2021.3.2.1

KEYWORDS

Parallel implementation, RC6 algorithm, Computer Science

ABSTRACT

Data security is very important in the field of Computer Science. In this paper, the encryption algorithm called RC6 will be analyzed and its standard and parallel implementation will be done. First, the field of Cryptology is discussed in general terms, and then the classification of encryption algorithms according to operation and techniques is explained. RC6 is a symmetric block algorithm derived from the RC5 algorithm. RC6 operates on 128-bit blocks and accepts 128, 192, 256-bit keys until 2040 bytes. In the Advanced Encryption Standard (AES) competition, RC6 managed to rank among the five finalists. The structure of the RC6 algorithm will be analyzed also the encryption and decryption methods. A comparison between standard and parallel implementation will be made.

1. Introduction

When we hear the word Cryptography, the first thought is that it could be email encryption, secure access to the website, bank card payment or code-breaking during World War II, such as the attack on well-known German coder "Enigma". Cryptography seems to be closely related to modern electronic communication. However, Cryptography is an old field, with the first specimens dating back to 2000 BC, when non-standard "secret" hieroglyphs were used in ancient Egypt. Apart from Egypt, Cryptography has been used in one or many different forms in most of the ancient cultures that have developed written language. For example, there is evidence of secret writings in ancient Greece, the so-called Scytale of Sparta, or Caesar's famous coder in ancient Rome.

Cryptology is divided into two main branches:

> ***Cryptography*** is the science of secret writing with the aim of concealing the meaning of messages.
> ***Cryptanalysis*** is the science of breaking cryptosystems. Cryptanalysis plays a key role in modern cryptosystems, as without people trying to break cryptosystems, it would not be possible to know whether those cryptosystems are safe or not.

The term cryptography comes from the Greek words "kryptos" which means "hide" and "graphen" which means "writing", which means "hidden writing". Therefore, cryptography deals with the creation and analysis of protocols that prevent third parties from accessing private messages, in other words, the purpose of cryptography is to hide the meaning of the message, so the sender does not worry if his message falls into someone else's hands while is incomprehensible.

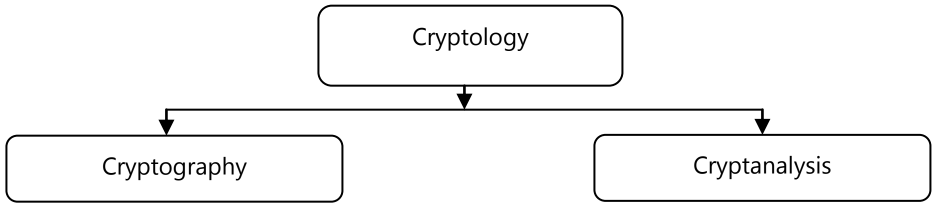

Figure 1. Cryptology

2. Terminologies

Suppose a sender wants to send a message to the recipient. Furthermore, this sender wants to send the message securely. He wants to make sure that the eavesdropper cannot read the message (Schneier, 1993) (Buchmann, 2004).

2.1 Messaging and Encryption

A message is called the base text. The process of masking a message in order to hide its substance is called encryption. An encrypted message is called encrypted text. The process of returning encrypted text to the base text is called decryption.

M or P. denotes basic text. It can be a bitstream, a text file, a bitmap file, a digitized audio stream, a digitized video, and so on. As long as it has to do with the computer, M is simply binary data. The base text may be intended for transmission or storage. In both cases, M is the message to be encrypted.

C. denotes the encrypted text. It is also binary data, sometimes of the same size as M, sometimes larger. The encryption function E operates on M to create C. Or in the mathematical notion: *E(M) = C.*

In the inverse process, the decryption function D operates on C to create M: *D(C) = M.*
Since the whole purpose of encrypting, then decrypting a message is to return the original base text, the following identity must be true: *D(E(M)) = M.*
The main purposes of Cryptography are:

- ***Confidentiality*** is the service that enables information to be used only by those who are authorized.
- ***Integrity*** is the service that addresses unauthorized data manipulation.
- ***Authentication*** is the service that is related to the identification. This function applies to both parties and also to information.
- ***Non-repudiation*** is the service that prevents parties from denying past actions.

3. Algorithms and keys

An encryption algorithm, also called encryption, is a mathematical function used for encryption and decryption. (Usually, there are two related functions: one for encryption and the other for decryption).
If the security of an algorithm is based on the fact that the internal structure of the algorithm is secret, then the algorithm is limited. Limited algorithms are of historical interest but are inadequate by today's standards. These algorithms cannot be used for a large group of users, as every time a user leaves the group, everyone has to change the algorithm. Furthermore, if anyone accidentally discovers the secret, everyone has to change their algorithm. Although limited algorithms have drawbacks in their operation, these algorithms are widely used in low-security applications. Modern cryptography solves this problem with a key; this key is denoted by K. The key can be any number of great values. The range of options for selecting this key is called the keyspace. Both encryption and decryption use this key (these actions are dependent on this and the index k denotes this fact), so the functions are now done:

$$E_K(M) = C$$
$$D_K(C) = M$$

These functions have the characteristic that:

$$D_K\ (\ E_K(M)) = M$$

Some algorithms use different encryption keys than decryption keys. So the encryption key K_1, is different from the decryption key K_2 , in this case:

$$E_{K_1}(M) = C$$
$$D_{K_2}(C) = M$$
$$D_{K_2}\left(E_{K_1}(M)\right) = M$$

All security in these algorithms is key-based (or keys), so it is not based on algorithm details. This means that algorithms can be published and analyzed. Products that use the algorithm can be mass-produced. It does not matter if the eavesdropper knows your algorithm. If the key is not known, he cannot read the messages. The cryptosystems that use the same key for encryption and decryption are called symmetric cryptosystem. If the key for encryption is different from the decryption key, then the cryptosystem is called asymmetric.

4. Security of algorithms

Different algorithms offer different degrees of security; this depends on the difficulty of breaking the algorithm. If the cost required to perform an algorithm is greater than the value of the encrypted data, then the algorithm used is safe. If the time required to break the algorithm is longer than the time that the data should be secret, then the algorithm used is safe. If the amount of data encrypted by a single key is less than the amount of data needed to break an algorithm, then the user is safe.
Lars Knudsen has classified these breaking categories of an algorithm.

- ***Total breakage***. A cryptanalyst discovers the key *K*, so that $D_K(C) = P$.
- ***Global deduction***. A cryptanalyst discovers an alternative algorithm A, equivalent to $D_K(C)$, without knowing *K*.
- ***Local deduction***. A cryptanalyst discovers the basic text of an encrypted captured text.
- ***Information deduction***. A cryptanalyst obtains some information about the key or base text. This information can be some bits of the key, some information about the form of the base text, etc.

An algorithm is totally secure if, despite how muchencrypted text a cryptanalyst has, it still does not have enough information to detect the underlying text. In fact, only the one-time pad technique is unbreakable even with unlimited resources. All other cryptosystems are vulnerable to encrypted text attack simply by trying all possible keys one by one and checking that the underlying text turns out to be meaningful. This type of attack is called a brute-force attack. Cryptography is more focused on cryptosystems that are computationally almost impossible unbreakable. An algorithm is considered computer-safe (sometimes called robust) if it cannot be broken with current or future resources.

5. RC6

RC6 is a block algorithm presented to NIST (National Institute of Standards and Technology) to be considered as a new standard for Advanced Encryption Standard (AES). RC6 is a family of fully parameterized algorithms. A version of RC6 is more precisely specified as RC6 - *w / r / b*, where the word length is *w*, the encryption consists of a nonnegative number of rounds *r* and *b,* indicating the encryption key's length in bits. RC6 - w / r / b operates in units of four words of w-bits using six basic operations for all variants. The base *2* logarithms of *w* will be defined as lgw(Menezes, Oorschot, & Vanstone, 1996).

- $a + b$ addition of modulo numbers 2^w
- $a - b$ subtraction of modulo numbers 2^w
- $a \oplus b$ XOR manipulation of word bits with length w-bits
- $a \times b$ multiplication of integer modulo 2^w
- $a <<< b$ for the sum given by lgw the least significant bits of *b* shift the words a to *b*-bits to the left.
- $a >>> b$ for the sum given by lgw the least significant bits of *b* move the words a to *b*-bits to the left.

Key generation

The key organization in RC6 - *w / r / b* is practically identical to the key organization for the RC5 - *w / r / b* algorithm. The only difference is that more words are obtained from the key given by the user for use during encryption and decryption.
The user gives a *b*-bit key. Zero enough bits are attached to give a key a length equal to a non-zero number of words. These keys are then stored in the little-endian mode in a string of w-bits of the words *L [0], ..., L [c-1]* (Rivest, Robshaw, Sidney, & Yin, The RC6 Block Cipher, 1998).

So the first bits of the key are stored as low-order bits of *L [0]*,etc .., and if necessary, *L [c-1]* is added to the high-order bits. The number of *w*-bit words to be generated by the extra rounds is *2r + 4* and these words are stored in the string *S [0,..., 2r + 3]*. The constants $P_{32} = B7E15163$ and $Q_{32} = 9E3779B9$ (hexadecimal numbers) are the same "magic constants" used in RC5 for the key body (Rivest, Robshaw, Sidney, & Yin, The RC6 Block Cipher, 1998).

The value of $P_{32}$is obtained from the binary expansion of (*e-2)*, where e is the basis of the natural logarithm function. The value Q_{32} is obtained from the binary expansion of *(Ø-1)*, where *Ø* is the Golden Ratio (Rivest, Robshaw, Sidney, & Yin, The RC6 Block Cipher, 1998). These values are in one way arbitrary and other values can be selected for "special" versions of the RC6 algorithm.
The pseudo-code for key generation is:

```
Input:     User-supplied b byte key preloaded into the c - word array L[0,...,c-1]
           Number r of rounds
Output:    w-bit round keys S[0,...,2r + 3]
Procedure:     S[0]=Pw
               for i=1 to 2r+3 do
                   S[i]=S[i-1]+Qw
               A=B=i=j=0
               v=3 x max{c, 2r+4}
               for s=1 to v do
                   {
                       A=S[i]=(S[i]+A+B)<<ε
                       B=L[j]=(L[j]+A+B)<<<(A+B)
i=(i+1) mod (2r+4)
```

```
        j=(j+1) mod c
    }
```

Encryption (Rivest, Robshaw, Sidney, & Yin)
RC6 works with 4 registers with *b* - bits *A, B, C, D,* which registers contain the input of the basic text as well as the output of the encrypted text at the end of the encryption. The first bit of the base text or encrypted text is set to the least significant bit of register *A*, the last bit of the base text or encrypted text is set to the most important bit (most -significant) of register *D*.
We use *(A, B, C, D) = (B, C, D, A),* to assign parallel values to the right of the registers on the left.
The pseudo-code for the encryption algorithm is (Rivest, Robshaw, Sidney, & Yin):

```
Input:   Plaintext stored in four w-bit input registers A,B,C,D
         Number r of rounds
w-bit round keys S[0,...,2r + 3]
Output:  Cipher text stored in A,B,C,D
Procedure:   B = B + S[0]
             D = D + S[1]
             for i = 1 to r do
             {  t = (B*(2B + 1)) <<< lg w
                u = (D*(2D + 1)) <<< lg w
                A = ((A ⊕ t) <<< u) + S[2i]
                C = ((C ⊕ u) <<< t) + S[2i + 1]
                (A, B, C, D)  =  (B, C, D, A)
             }
             A = A + S[2r + 2]
             C = C + S[2r + 3]
```

Decryption (Rivest, Robshaw, Sidney, & Yin)
Decryption in the RC6 algorithm is done in this form, based on this pseudo code.
The pseudo code for the decryption algorithm for RC6 is (Rivest, Robshaw, Sidney, & Yin):

```
Input:   Cipher text stored in four w-bit input registers A,B,C,D
         Number r of rounds
w-bit round keys S[0,..., 2r + 3]
Output:  Plaintext stored in A,B,C,D
Procedure:   C = C - S[2r + 3]
             A = A - S[2r + 2]
             for i = r downto 1 do
                {
                   (A, B, C, D) = (D, A, B, C)
                   u = (D*(2D + 1)) <<< lg w
                   t = (B*(2B + 1)) <<< lg w
                   C = ((C - S[2i + 1]) >>> t) ⊕ u
                   A = ((A - S[2i]) >>> u) ⊕ t
                }
             D = D - S[1]
             B = B - S[0]
```

6. Implementation of RC6 algorithm in parallel and sequential form in SCALA
To encrypt a long basic text with an algorithm block, there are different ways of operating. They are (KK) (William, 2017):

- Electronic Code Book mode (ECB),
- Cipher Block Chaining mode (CBC),
- Cipher Feedback mode (CFB),
- Output Feedback mode (OFB),
- Counter mode (CTR)

ECB (Electronic Codebook Mode) is the most suitable for parallel implementation. Let $e_k(x_i)$ encrypt the base text's block xi with key k using a block cipher. Let $e^{-1}{}_k(y_i)$ decrypts the text of encrypted blocks y_i with key k. Assume that the block algorithm encrypts (decrypts) blocks of size b bits. Messages exceeding b-bits are divided into blocks of b-bits. If the message length is not a multiple of b-bits, then padding must be done before encryption so that the length is a multiple of b-bits. As shown in figure 2 and figure 3, each block is encrypted (decrypted) separately in ECB mode. The blocking algorithm can be, for example, AES, 3DES or RC6.

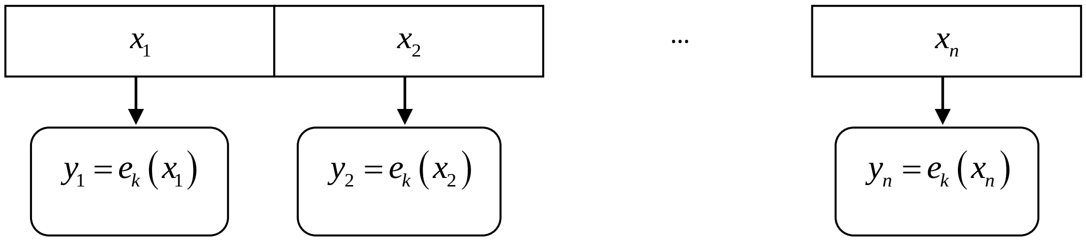

Figure 2. ECB encryption mode

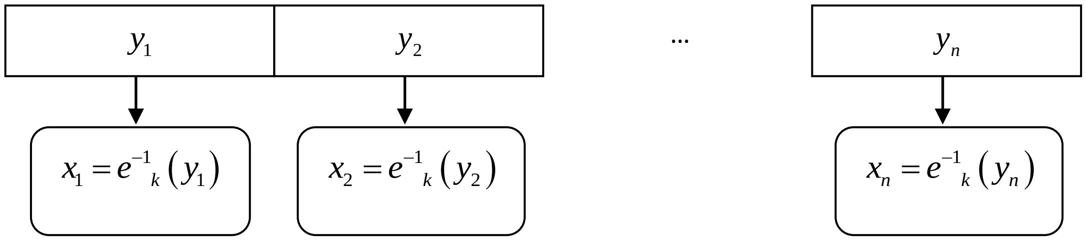

Figure 3. ECB decryption mode

Definition (Boneh & Shoup, 2017) : ECB

Let *e()* be block cipher with bock size of b bits and let x_i, $y_i$be messages of same size b.

Encryption: $y_i = e_k(x_i), \quad i \geq 1$

Decryption: $x_i = e_k^{-1}(y_i) = e_k^{-1}(e_k(x_i)), \quad i \geq 1$

The ECB way has several advantages. Synchronization of encryption and decryption blocks between Alice and Bob is not necessary. For example, if the receiver does not receive all the encrypted blocks due to transmission problems, it is possible to decrypt those received blocks. Similarly, bit errors, e.g., caused by noise in transmission channels, can damage only the corresponding blocks but not those reached blocks (Paar & Pelzl, 2011).

Also, block algorithms in ECB mode can be paralleled, e.g. one unit encrypts (or decrypts) block 1, the other unit blocks 2, and so on. This is an advantage for high-speed implementations, but many modes of operation such as CFB (Cipher Feedback) do not enable parallelization (Paar & Pelzl, 2011).

There are some weaknesses associated with the ECB mode. The main problem of the ECB is that encryption is very determinative. This means that identical base text blocks result in identical blocks of encrypted text, as long as the encryption key does not change.

Method *par* in Scala

The shift in recent years from processor manufacturers from single-core architecture to multi-core architecture (D., Paris France, May 2008) (Breshears, 2009), it is necessary that parallel programming remain a challenge. Parallel collections are included in the SCALA standard library to facilitate parallel programming while saving the user from low-level parallelization details while giving easy access to high-level abstraction. The idea was that the implicit parallelism behind abstract collections would bring parallel execution closer to access to work by developers.

The idea is simple - collections are well-understood and frequently used programming abstracts. And given the regularity, these collections can be effectively paralleled. By allowing users to "swap" sequential collections for those operating in parallel, SCALA parallel collections take a step forward by enabling parallelization to be performed as easily as possible.

We take the following example, where we perform a monad action on a large collection:

```
val list = (1 to 10000) .toList

list.map (_ + 42)
```

To perform parallel operations, the ***par*** method must be invoked on a sequential list collection. After that, a parallel collection in the same form as a sequential collection can be used. The above example can be paralleled by doing the following:

list.par.map (_ + 42)

The SCALA parallel collections library contains a number of important data including (Fédérale, 2002):

- ParArray
- ParVector
- Mutable.ParHashMap
- Mutable.ParHashSet
- immutable.ParHashMap
- immutable.ParHashSet
- ParRange

Semantics: While the abstraction of parallel collections is almost the same as normal sequential collections, it is important to note that semantics vary, especially in actions with side and non-associative effects. In order to see this issue, we visualize how actions are paralleled. Conceptually, the framework of Scala parallel collections parallels an action in a parallel collection, recursively "dividing" a given collection, applying an action to each parallel division, and recombining the results that have been achieved in parallel (Scala par method).

Parallelization of the Encrypt method
Method encrypt in RC6 algorithm:

```
defencrypt(data: Array[Byte], key1: Array[Byte]): Array[Byte] = {
val key = RC6Par.paddingKey(key1)
  S = generateSubkeys(key)
vallength = 16 - data.length % 16
val bloc = new Array[Byte](data.length + length)
valpadding = new Array[Byte](length)
padding(0) = 0x80.toByte
  (0 to (data.length + length)/16).par.foreach { blocI=>
    (1 to 16).foreach { tmpI =>
val i = Math.min(blocI*16 + tmpI, data.length + length -1)
if (i > 0 && i % 16 == 0) {
valblocHolder = BlockEncrypter(util.Arrays.copyOfRange(bloc, i - 16, i))
System.arraycopy(blocHolder, 0, bloc, i - 16, blocHolder.length)
     }
if (i <data.length) bloc(i) = data(i)
else bloc(i) = padding((i - data.length) % length)
    }
  }
valblockHolder = BlockEncrypter(util.Arrays.copyOfRange(bloc, data.length + length - 16, data.length + length))
System.arraycopy(blockHolder, 0, bloc, data.length + length - 16, blockHolder.length)
bloc}
```

In this method the base text is padded and the base text is divided into blocks.

Using the par method in Scala these blocks are encrypted in parallel by the BlockEncrypter method.

Parallelization of the Decrypt method

```
defdecrypt(data: Array[Byte], key1: Array[Byte]): Array[Byte] = {
var bloc = new Array[Byte](data.length)
val key = RC6Par.paddingKey(key1)
  S = generateSubkeys(key)
  (0 to data.length/16).par.foreach { blocI =>
    (1 to 16).foreach { tmpI =>
val i = Math.min(blocI*16 + tmpI, data.length - 1)
if (i > 0 && i % 16 == 0) {
valblockHolder = BlockDecrypter(util.Arrays.copyOfRange(bloc, i - 16, i))
System.arraycopy(blockHolder, 0, bloc, i - 16, blockHolder.length)}
if (i <data.length) bloc(i) = data(i)
      }
   }
valblocHolder = BlockDecrypter(util.Arrays.copyOfRange(bloc, data.length - 16, data.length))
System.arraycopy(blocHolder, 0, bloc, data.length - 16, blocHolder.length)
bloc = deletePadding(bloc)
bloc }
```

In this method, the encrypted text is split into blocks, then using the first Scala method, the blocks are decrypted in parallel by the BlockDecrytpter method.

7. Experiment and results

The experiment, in this case, is performed by classes, one class (Measure) measures the difference of execution time between the sequential and parallel algorithm, testing it for a random number of strings and the number of tests is determined by the user (Table 1, Figure 4, Figure 5). And the other class (Measure2) measures the time difference between the sequential and parallel algorithm for a fixed string (Table 2, Figure 6, Figure 7). Experiments and tests were performed on a computer with an Intel Core i5-3317U @ 1.7GHz processor and 6 GB RAM, with Windows 8.1 operating system.

Each set of values in these tables was executed 10 times and the average of the values obtained from the experiment was taken. Time is measured in nanoseconds then converted to seconds.

Table 1. Resuls gained with testing class Measure

Iterationnumber	Length of randomstring(n-characters)	Execution time for RC6	Execution time for RC6Par	Ratio	Speedup
10000	n = 100	0.18942s	1.225s	6.5	0.15
10000	n = 1000	0.96491s	1.9334s	2.00	0.49
10000	n = 10000	8.9246s	5.8009s	0.65	1.54
10000	n = 100000	89.787s	44.205s	0.52	1.93
10000	n = 1000000	815.89s	411.67s	0.5	1.98
(*)1000	n* = 10000000	884.25s	416.87s	0.47	2.12

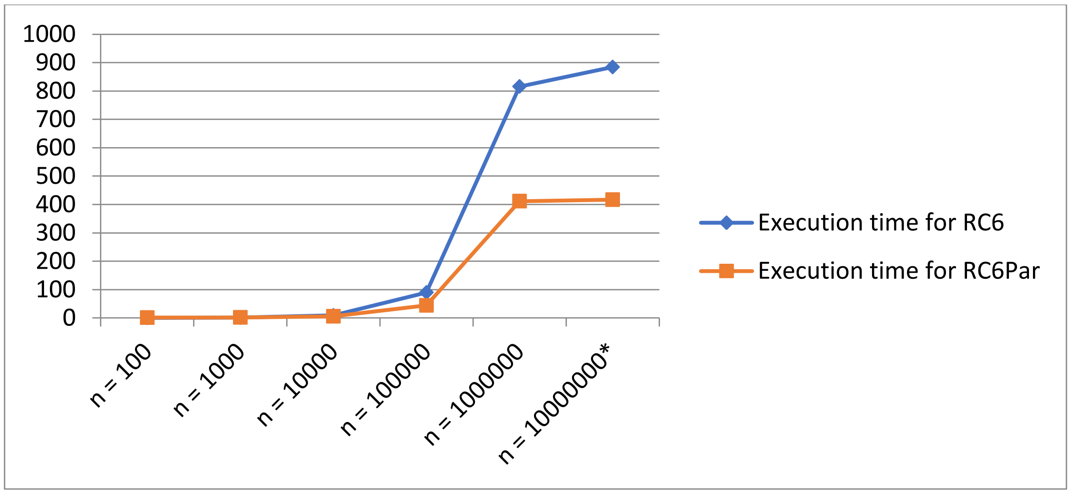

Figure 4. Execution time (time converted to seconds)

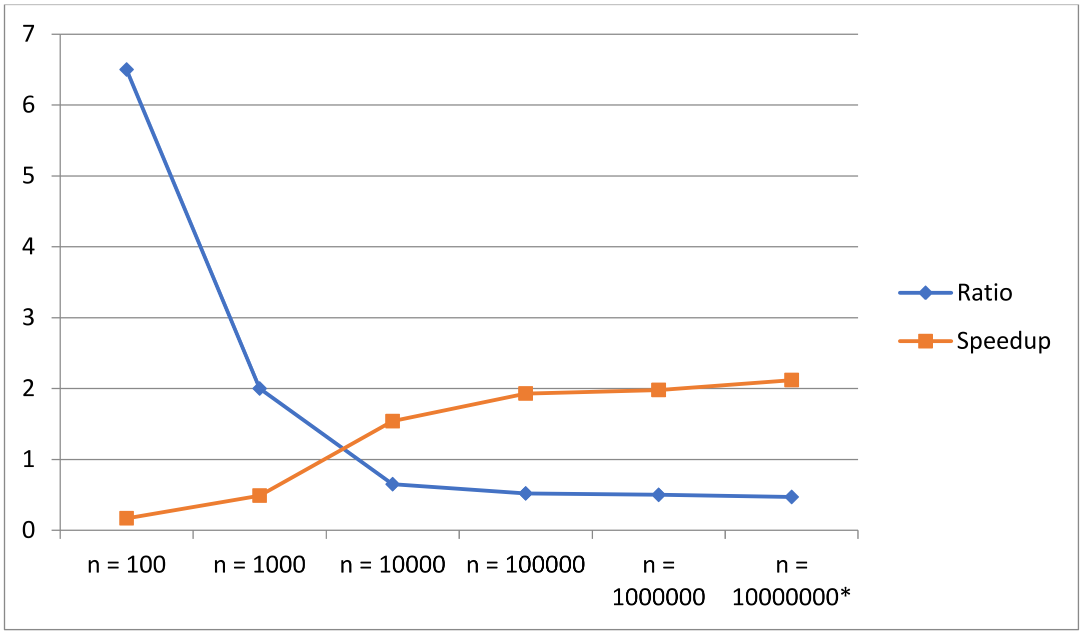

Figure 5. Speedup (time converted to seconds)

Table 2. Resuls gained with testing class Measure2

Iterationnumber	Length of randomstring(n-characters)	Execution time for RC6	Execution time for RC6Par	Ratio	Speedup
10	n=100	6.0775e-5s	0.00035298s	5.82	0.18
10	n=1000	0.0002s	0.00054395s	1.85	0.54
10	n=10000	0.00275s	0.0015509 s	0.57	1.79
10	n=100000	0.027188s	0.011799s	0.44	2.31
10	n=1000000	0.28606s	0.12008s	0.42	2.38
10	n=10000000	4.8494s	1.2047s	0.28	4.1
10	n=50000000	16.244s	7.0929s	0.44	2.29
10	n=80000000	26.113 s	10.158s	0.39	2.54

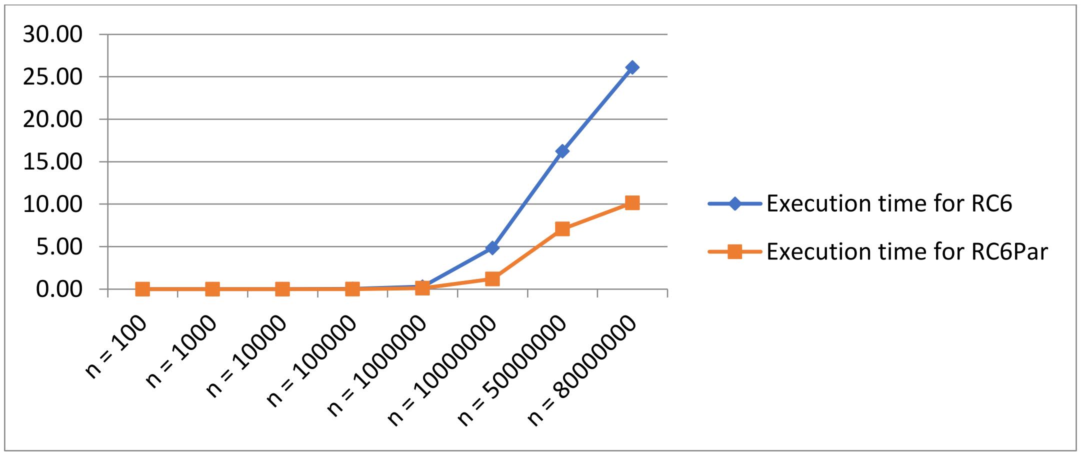

Figure 6. Execution time (time converted to seconds)

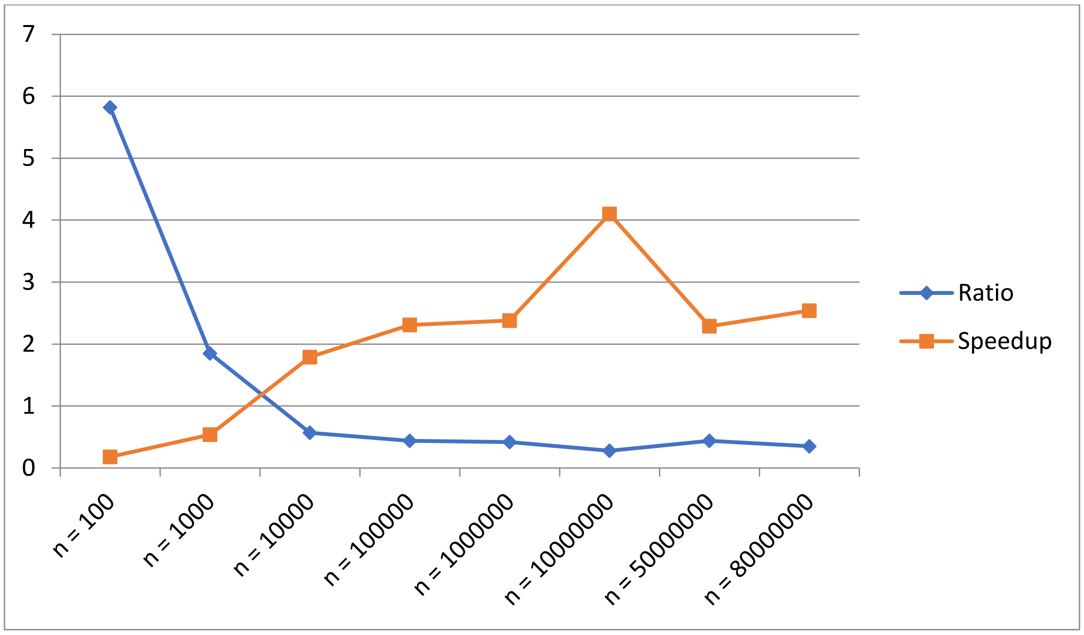

Figure 7. Speedup (time converted to seconds)

8. Conclusion

We have introduced the RC6 algorithm, this algorithm for data encryption and decryption.

S RC6 is a symmetric block algorithm derived from the RC5 algorithm. RC6 algorithm is designed by Ron Rivest, Matt Robshaw, Ray Sidney, and Yiqun Lisa Yin to meet the Advanced Encryption Standard (AES) competition requirements. The RC6 algorithm was one of the 5 finalists of that competition.

It is a secure, compact and simple algorithm and offers good performance and considerable flexibility.

The RC6 algorithm can have a block size up to 2040 bits and be parameterized to support different word lengths and rounds.

In terms of security, this algorithm turns out to be quite secure since if used in the 20-round version, the best chance of cryptanalysis to break the algorithm is to have 2^{128} basic texts.

Sequential and parallel implementation of the algorithm was done and the algorithm for words of different lengths was tested.

During the tests, it was noticed that the parallel form is faster, but also this form requires more resources. So for messages up to 2GB in length, it has not been possible to test because of the inability to access a high-performance computer.

References

[1] Boneh, D., & Shoup, V. (2017). *A graduate course in Applied Cryptography.*
[2] Breshears, C. P. (2009). *The Art of Concurrency.* O'Reilly.
[3] Buchmann, J. A. (2004). *Introduction to Cryptography.* Springer.
[4] D., L. (Paris France, May 2008). CUDA: Scalable Parallel Programming for High-Performance Scientific Computing. *In Proceedings of 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macr, ISBI 2008*, pp. 836-838.
[5] Fédérale, É. P. (2002). *The Scala Programming Language*. Retrieved 01 15, 2020, from https://www.scala-lang.org/
[6] KK. (n.d.). *Block Ciphers Modes of Operation*. Retrieved 3 9, 2020, from http://www.crypto-it.net/eng/theory/modes_of_block_ciphers.html
[7] Menezes, A. J., Oorschot, P. C., & Vanstone, S. A. (1996). *Handbook of Applied Cryptography.* CRC Press.
[8] Paar, C., & Pelzl, J. (2011). *Understanding Cryptography - A Textbook for Students and Practioners.* Springer.
[9] Rivest, R. L., Robshaw, M., Sidney, R., & Yin, Y. (n.d.). Retrieved 02 03, 2021, from https://people.csail.mit.edu/rivest/pubs/RRSY98.pdf
[10] Rivest, R. L., Robshaw, M., Sidney, R., & Yin, Y. (1998). The RC6 Block Cipher. *Inin First Advanced Encryption Standard (AES) Conference.*
[11] *Scala par method.* (n.d.). Retrieved from http://docs.scala-lang.org/overviews/parallel- collections/overview.html
[12] Schneier, B. (1993). *Applied Cryptography.* Wiley.
[13] William, S. (2017). *Cryptography and Network Security - Principles and Practice.* Pearson.