Abstract
Symmetric key cryptography is the most commonly used cryptographic primitive in domain encryption that meets the requirement devices which are mostly used today. In the recent past a few such algorithms have been implemented to secure communication channels. In this paper, a domain encryption algorithm SAR 256 (a symmetric key cipher) with probabilistic slice and shift operation has been implemented. Developing a software-based domain encryption algorithm SAR 256 seems to be very beneficial for short length data encryption like node of wireless sensor networks (WSN), SSN, iNode of a file system, password, ATM PIN and other electronic data. The cipher SAR 256 is software-based synchronous ultra-lightweight cryptosystem which mainly design for resource-constrained devices such as Radio Frequency IDentification Devices (RFID) Tags, WSN, iNode, blockchain and other such platform with limited processing capabilities. The cipher SAR 256 inherits the hybrid feature of some well-known cryptographic primitives like Blowfish, Twofish, DES, StegoCrypt3D and IDEA. The cipher SAR 256 uses a variant length keystream ranging from $2^8 \rightarrow 560$ bit long which is known to be safe and can withstand against several cryptographic attacks. It also uses a dynamic $(16, 32)$ stage slice and shift operation with an average to worst-case probabilistic vulnerability of $\approx 0.000125$ with the average plaintext length $[16.5]$. However, in the best and the worst case, the state update function yields a probability of guess and determines attack as $\approx 0.005$ and $\approx 0.00000329$ respectively. The cryptosystem SAR 256 is implemented in 'C' with GCC Version 10.0.1 and has been tested under Xilinx ISE 14.2 with 1.8 GHz 8-core processor.

Keywords: e-signing, slice and shift, PRNG, state update, grid permutation, FRAME_3X, intermediate key, initialization vector and boolean gear.

1. Introduction
Despite an increase in overall security investment over the last two decades, breaches continue to haunt enterprises, academia and corporate sectors. Furthermore, we have seen that many of the cyberattacks that lead to breaches make use of encryption in some way. Unauthorized users may gain access to data without the approval or knowledge of the intender participant. Even though access-level controls should have prevented this, recent experience indicates that human errors happen all the time. With encryption, the situation changes dramatically even if human errors such as phishing, having same password for several accounts, unconscious share of OTP have taken place. In order to overcome with such serious issues, we have proposed a symmetric key cryptographic algorithm SAR 256. The algorithm SAR 256 is based on the pre-computation slice and shift operation. Despite the fact that precomputation vectors are part of a larger class of designs known as combinatorial designs, which are often difficult to find, a proper match of the original vectors. The encryption function is based on the mix huge precomputation that results an array of fixed length ASCII characters to get a vector of padded hexa. The cipher is identical to the Hill cipher. The cipher mainly designs for domain encryption and is much suitable for resource-
constrained devices. It uses the principle of confusion and diffusion to create an intermediate vector that further goes under padding, swapping and shuffling. At each pass of the encryption, the cipher enters into a state update operation that makes the cipher strong against many cryptographic attacks. The state update function 'doSplitToShuffle' is based on bounded PRNG, segmentation, rounding a slice and boolean gear that makes the precomputation complex. Reverse engineering of the encryption is also performed in the inverse fashion of the encryption procedure. Section 1 describes basic terminology associated with cryptography and cryptanalysis. Section 3 and 4 represent the associated problems and related work of the proposed model SAR 256. Implementation of the cipher is done in section 5 while the driver function and entry point of the cipher is presented in section 6 and 7 respectively. After implementing and testing the cipher SAR 256 under Xilinx 14.2, the test vector of the simulation is presented in section 8. The conclusion and future work of the model is described in section 9.

2. Basic Terminology

2.1 Cryptology

Cryptology is a branch of mathematics which comprise cryptography and the art of cryptanalysis such as numerical theory and application of formulas and algorithm. Since the concept for cryptanalysis is highly specialized and complex, here we only focus on some of the key conceptual mathematics behind cryptography. It encompasses both cryptography and cryptanalysis [1].

2.2 Plaintext and Ciphertext

Plaintext (PT) is a term used in cryptography that refers to a message before encryption or after decryption or the information that can be directly read by humans or a machine. Whereas a ciphertext (CT) is the encrypted text transformed from the plaintext by using an encryption algorithm? The term "cipher" is sometimes used as an alternative term for ciphertext. The ciphertext is not understandable until it has been converted into plaintext using a key [1, 2].

2.3 Key and Cryptographic Systems

A cryptographic key ‘Key or (K)’ is the set of bits that may present in binary, decimal, or hexadecimal is mainly used for the encryption of a message (plaintext) and the decryption of a secret message (ciphertext). Key is generally serving as a transformation parameter of the cryptographic algorithm and makes a cryptosystem public or private. If the key is shared (or common) for both encryption and decryption processes, the cryptosystem is referred to as a symmetric key or private key cryptosystem. Blowfish, AES, RC4, DES, RC5, and RC6 are examples of some symmetric key cryptosystems [3, 4, 5, 6]. On the other hand, if the key is not shared in the encryption and decryption process, the cryptosystem is referred to as an asymmetric key or public-key cryptosystem. Some examples of asymmetric key cryptography are RSA, Diffie-Hellman Key Exchange, Elliptic Key Cryptography (ECC). According to Kerckhoff’s Principle, key complexity is an important part of the algorithm design and reliability of a cryptographic algorithm. The security of a key is mainly depending on [7, 8, 9].

- Key confidentiality or key secrecy
- Key authenticity or verification of key sender identity
- Authorized use of the key or permissible use of the key

2.4 Encryption | e(PT, Key)

Encryption is the process of encoding data into an unreadable form. During this process, the original text (or the message is called plaintext) is transformed into an alternative form is known as ciphertext. Only the approved parties are allowed to decode the ciphertext back to get the original plaintext. Mathematically an encryption algorithm is represented as $e(PT, Key)$ [1].

2.5 Decryption | d(CT, Key)

Decryption is the process by which the encrypted data (ciphertext) is converted back into its original form (plaintext). It is noted that encryption without the correct key is very difficult for all practical purposes, and in some cases it’s impossible. Mathematically a decryption algorithm is represented as $d(CT, Key)$ [1].

2.6 The Cipher

A cipher is an algorithm mainly design for encryption or decryption. It contains a sequence of well-defined procedure that can be followed. Shift ciphers are one of the earliest cryptosystems and the simplest. There are various types of ciphers are there such as shift cipher, caesar cipher, substitution ciphers, transposition ciphers, polyalphabetic ciphers, nomenclator ciphers, Permutation Cipher and many more.
Cryptography can be defined as the art and science of cipher creation. It is the oldest of techniques for secure communication and constructing and analyzing protocols that prevent third parties or the public from reading private messages. The goal of cryptography is to maintain and achieve integrity, authentication, confidentiality and non-repudiation. There are three types of cryptographic techniques that are used, they are:

- Public key cryptography (asymmetric)
- Private key cryptography (symmetric)
- The hash functions

Cryptology has classified into two parts: (i) cryptography and (ii) cryptanalysis. Cryptography focuses on the construction of ciphers, secret codes and encryption/decryption functions. Whereas cryptanalysis is the study of ciphers, ciphertexts and a cryptosystem, while cryptanalysts attempt to decode a ciphertext without knowing the actual plaintext, encrypted key and algorithm used to encrypt the plaintext. Cryptanalysis entails a thorough examination of the cryptographic technique and its cracking. Some well-known cryptanalysis techniques and cryptanalytic attacks are Known-Plaintext Analysis, Chosen-Plaintext Analysis, Ciphertext-Only Analysis, Man-In-The-Middle or MITM attack and Adaptive Chosen-Plaintext Analysis or ACPA. CrypTool, EverCrack and Cryptol are some important tools used for Cryptanalysis.

A cryptosystem consists of a series of algorithms that converts plaintext into ciphertext and ciphertext to plaintext conversely. A full suite of encryption, decryption and key generation algorithms and protocols are the members of a cryptosystem. Examples of some well-known cryptosystems are Rabin, Cramer-Shoup, the Benaloh, AES and RSA [2, 3].

Applications store their sensitive information like credentials in a tabular format that contains a username and corresponding password in the database. When a server receives a payload authentication request to authenticate a particular credential, an eavesdropper may take an attempt to break the security of the communication channel (or even the database) and can access that sensitive information [10, 11]. If an attacker broke into the database and stolen the entire credential, then every user’s account could be accessed and gets hacked. To overcome from such serious issue, this paper presents a software-oriented domain encryption algorithm: SAR 256. The SAR 256 has been developed to protect sensitive data such as bank details, social security numbers, biometric information, credentials, medical data and personally identifiable financial information from the unauthorized access. The proposed algorithm has been designed in such a way that can encrypt data of length varies from 80-256 bit long and is claimed to be cryptographically secure against several cryptographic attacks like Linear Masking Attack, Guess and Determine Attack, Correlation Attack, Key Recovery Attack and Timing Attack. The proposed model SAR 256 is based on the concept of binary precomputation, shift permutation, shadow password and principle of confusion and diffusion [12, 13].

The Data Encryption Standard (DES) is a symmetric-key block cipher that was first published by National Institute of Standards and Technology (NIST) in 1975. It is based on 16 round permutations that use 64-bit block format and based on the Feistel network with a key length is 64-bit [14]. DES has an efficient 56-bit key length as the encryption algorithm (only check bits function) does not use 8 of the 64-bit key. The main problem associated with the DES is that it can break using linear cryptanalysis. This problem gets resolved in 3DES by increasing other performance metrics such as gate equivalent (GE), ROUND and key size [15].

Triple DES is a symmetric-key block cipher that applies the DES cipher algorithm three times to each data block by uses using the single keys of DES (of 56 bits length) at each time. It was the most commonly used symmetric algorithm in the industry and recommended standard. Although, the total key length contributes to 168 bits and [16, 17] claim that 112 bits are more similar with key strength. Triple DES can still provide robust hardware encryption [18].

4.1 Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric-key block cipher that was first published by National Institute of Standards and Technology (NIST) in 1975. It is based on 16 round permutations that use 64-bit block format and based on the Feistel network with a key length is 64-bit [14]. DES has an efficient 56-bit key length as the encryption algorithm (only check bits function) does not use 8 of the 64-bit key. The main problem associated with the DES is that it can break using linear cryptanalysis. This problem gets resolved in 3DES by increasing other performance metrics such as gate equivalent (GE), ROUND and key size [15].

4.2 Triple DES

Triple DES is a symmetric-key block cipher that applies the DES cipher algorithm three times to each data block by uses using the single keys of DES (of 56 bits length) at each time. It was the most commonly used symmetric algorithm in the industry and recommended standard. Although, the total key length contributes to 168 bits and [16, 17] claim that 112 bits are more similar with key strength. Triple DES can still provide robust hardware encryption [18].

DOI : 10.21817/indjcse/2021/v12i4/211204022  Vol. 12 No. 4 Jul-Aug 2021  800
4.3 Blowfish

Blowfish is another variant of the DES replacement algorithm. This symmetric cipher divides and encrypts messages into 64-bit blocks. Blowfish is a symmetric block cipher that was first published in 1993 by Bruce Schneier. It uses 32-448 bits key size and 64-bit block size that is permuted in 16 rounds to get a ciphertext [3]. Blowfish provides an optimal rate of encryption and yield secure ciphertext. Till date no cryptanalysis report has been reported against this cipher. It is based on the Feistel network structure and features included key-dependent substitution boxes (S-boxes) and a very high key generation schedule. Sometimes it is also called 16-round Feistel cipher [4, 19]. Use of 64 bits block makes Blowfish vulnerable against birthday attack when it encrypts data in HTTPS protocol and is also known to be susceptible to known-plaintext attacks [20]. In 2016, the SWEET-32 attack has vulgarized the cipher and recover the plaintext successfully.

4.4 Twofish

Blowfish and his predecessor Twofish are the key figures of computer security expert Bruce Schneier. The algorithms use a secret key of length up to 256 long and only one key is needed as a symmetric cryptographic system [21]. Twofish is also known to be one of the fastest and ideal for both software and hardware implementation. Due to the unique and complex design, it is considered the best choice among all AES candidates [22].

4.5 The AES

The Advanced Encryption Standard (AES) is an algorithm that transforms a fixed-length plaintext string into a new bitstream of the same length through a series of complicated operations was first published in 1998. It uses 192-bit and 256-bit keys for heavy-duty encryption, but it is extremely successful in 128-bit key size [6, 23, 24]. It is known to be completely reliable to any attack except for brute force which is attempting to decode the messages using all possible 128-192 bits or 256-bit cipher combinations. However, due to the simple algebraic structure, similar block encryption and complex counter mode the AES is not suitable for short data encryption [25, 26].

4.6 Bcrypt

The Bcrypt function, based on a blowfish cipher presented at USANIX in 1999, is designed by Niels Provos and David Mazières [27]. As well as using salt in the blowfish table, over time the iteration count can be extended to make it slower, thus remaining resistant to brute-force attacks even with the increasing number of computers. Besides Bcrypt, several Java secure hash functions such as MD5, SHA256, SHA512, PBKDF2 and Scrypt are also available for domain and credential encryption [28, 29].

5. Implementation of SAR 256

Pseudo slack low bit(s) PSLB act/serves as a pad for create blocks of fixed length [30].

```c
char PSLB_1[] = "0";
char PSLB_2[] = "00";
static int TRUE = 1;
```

5.1 int getLength(int)

The function getLength expect a positive integer and return number of digits does it have.

```c
int getLength(int N) {
    int i=0;
    while(N) {
        i++;
        N=N/10;
    }
    return i;
}
```

5.2 char * getHexa(int, int)

The function getHexa is implemented to convert a decimal value into hexadecimal. It returns a hexadecimal value PADDED_hexa of length 4, if boolean flag is set to 0 else return a 3-digit hexadecimal value when boolean flag is set to 1.

```c
char* getHexa(int DEC, int flag) {
    if(DEC==0 && flag==0) return "0000";
```
char RAW_Decimal[100];
static char PADDED_Hexa[100];
int i=0;
while(DEC) {
   int temp=DEC%16;
   if(temp<10) {
      RAW_Decimal[i++]= temp + 48;
   } else {
      RAW_Decimal[i++]= temp + 87;
   }
   DEC/=16;
}
int z=0;
if(flag==0 && i!=4) {
   PADDED_Hexa[0]= '0';
   z++;
}
if(i==1) {
   PADDED_Hexa[z++]='0';
   PADDED_Hexa[z++]='0';
}
if(i==2) {
   PADDED_Hexa[z++]='0';
}
for (int j = i - 1; j >= 0 ; j--){
   PADDED_Hexa[z++]=RAW_Decimal[j];
}
PADDED_Hexa[z]= '\0';
return PADDED_Hexa;
}

5.3 char * getDecimal(char RAW_Hexa[],int OLF) {
   if(RAW_Hexa[0]=='0' && RAW_Hexa[1]=='0' && RAW_Hexa[2]=='0' && RAW_Hexa[3]=='0' && OLF==0)
      return "0000";
   if(RAW_Hexa[0]=='0' && RAW_Hexa[1]=='0' && RAW_Hexa[2]=='0' && OLF!=0)
      return "000";
   int DECIMAL_STORE = 0, BASE = 1, L = strlen(RAW_Hexa);
   static char i_Key[5];

   for (int i = L - 1; i >= 0 ; i--){
      if (RAW_Hexa[i]>='0' && RAW_Hexa[i]<='9') {
         DECIMAL_STORE += (RAW_Hexa[i] - 48)*BASE;
         BASE = BASE * 16;
      } else if (RAW_Hexa[i] >= 'a' && RAW_Hexa[i] <= 'f') {
         DECIMAL_STORE += (RAW_Hexa[i] - 87)*BASE;
         BASE = BASE * 16;
      }
   }
   return DECIMAL_STORE;
}

The function getDecimal expecting two formal parameters, the first is a collection of hexadecimals as RAW_hexa and the second is Output Length Flag (OLF) which determines the number of bits in the list i_Key. It converts the expected hexadecimal value from the collection RAW_hexa and converts each value into its equivalent decimal of length 4 (when OLF set to 0) or of length 3 (when OLF is set to 1) [30].
```c
int k=0;
if(OLF==0 && getLength(DECIMAL_STORE) != 4) {
    i_Key[k++]=’0’;
}
if(getLength(DECIMAL_STORE) == 1) {
    i_Key[k++]=’0’;
    i_Key[k++]=’0’;
    i_Key[k++]= (char)(DECIMAL_STORE + 48);
}
if ( getLength(DECIMAL_STORE) == 2){
    i_Key[k++]=’0’;
    i_Key[k++]= (char)(DECIMAL_STORE / 10 + 48);
    i_Key[k++]= (char)(DECIMAL_STORE % 10 + 48);
}
if ( getLength(DECIMAL_STORE) == 3){
    i_Key[k++]= (char)(DECIMAL_STORE /100 + 48);
    i_Key[k++]= (char)({DECIMAL_STORE/10} %10 + 48);
    i_Key[k++]= (char)(DECIMAL_STORE % 10 + 48);
}
if ( getLength(DECIMAL_STORE) == 4){
    i_Key[k++]= (char)(DECIMAL_STORE/1000 + 48);
    i_Key[k++]= (char)({DECIMAL_STORE%1000}/100 + 48);
    i_Key[k++]= (char)({DECIMAL_STORE%100}/10 + 48);
    i_Key[k++]= (char)(DECIMAL_STORE % 10 + 48);
}
    i_Key[k]=’\0’;
return i_Key;
}

5.4 char * mergeList(char *, char *)

The function `mergeList` expect two pointers to characters as `list_a` and `list_b`. It appends the content of string `list_b` at the end of the string `list_a` and return the whole as a collection.

```c
char* mergeList(char* List_a, char* List_b) {
    char* COLLECTION = (char*)malloc((strlen(List_a)+strlen(List_b)+1)*sizeof(char));
    for(int i = 0; i < strlen(List_a); i++) {
        COLLECTION[i]=List_a[i];
    }
    for(int i = 0; i < strlen(List_b); i++) {
        COLLECTION[i+strlen(List_a)]=List_b[i];
    }
    COLLECTION[strlen(List_a)+strlen(List_b)]='\0';
    return COLLECTION;
}
```

5.5 int getPRNG(int, int, int)

Function `getPRNG` (Pseudo Random Number Generator) expects three integral parameters ‘FROM’ , ‘TO’ and ‘except’ and returns a random number lie in between these two intervals (FROM, TO] excluding FROM and including TO. The third parameter ‘except’ exclude a number if it occurs in between these intervals. It return a random value within the range { (FROM, TO] – except } [31, 32].

```c
int getPRNG(int FROM, int TO, int except) {
    FROM++;
    int RAND = rand() % (TO - FROM + 1) + FROM;
    if(except == -1) {
        return RAND;
    }
    else {
```
while(TRUE) {
    if(RAND != except)
        break;
    RAND = rand() % (TO – FROM + 1) + FROM;
}
return RAND;

5.6 char * forwardPAS(char **, char *, int)

The function forwardPAS (Forward Padding and Swapping) expect three formal parameters, they are:
- GRID, a pointer to pointer to char
- PlainText, a pointer to char
- PRNG, a random integer

As function gets called, we pass the value NULL value as an actual parameter to GRID a message plaintext and a random number PRNG as actual parameters. When control enters this method, the length of the plaintext (message to be encrypted) gets computed and is assigned to variable ‘L’ then determines the ‘ASCII’ weight of each character belonging to the instance of plaintext PlainText and multiply it with the random integer PRNG.

The final multiplication is kept on variable ASCII (a variable of type int). In this function we have only two possible values of PRNG (2 or 3), the possible range of ASCII of plaintext characters could vary from 0 to 255. Thus, in the worst case, the variable ASCII may contain value 2 (when the weight of ASCII = 1) and in the best case, it can contain value 765 (when the weight of ASCII = 255). Since the number of digits of ASCII is not fixed in this range, so there is a need to pad some slake digits to each of them. After padding some Pseudo Slack Low Bit (PSLB) to each ASCII, number of bits turn to fix so that the number of digits in ASCII can have the same length, so the vector GRID may contain elements of length 3 or 24 bits exactly. Next, the algorithm generates a random integer ROUND within the range (10, 20] such that |ROUND| = 10 and place the digits of ROUND in first two consecutive cell of another vector MUX. Now it generates two Mutually Exclusive True Random Number (MITRN) Lx_i and Lx_f such that 0 ≤ Lx_i < L and 0 ≤ Lx_f < L – {Lx_i} to swap the block of GRID located at index Lx_f and Lx_f up to ROUND times as shown in Fig. 1.

![Fig. 1: Huge GRID precomputation of SAR 256](image)

Since the value of PRNG is non-deterministic (but bounded) within interval 10 and also the value attends by ROUND is MITRN in nature and is bounded within the same interval 10. So, the probability to determine a correct sequence of PRNG and ROUND becomes $\frac{1}{10^2} = 0.01$. At each pass of the ROUND, the probability to determine a correct Lx_i w.r.t. its corresponding Lx_f is $\frac{1}{L(L-1)}$. So, the exact probability to determine a correct sequence of PRNG, ROUND and their associated Lx_i w.r.t. its corresponding Lx_f becomes $\frac{0.01}{L(L-1)}$. Since the average length of the plaintext is 16.5 (because $\frac{32 \times 16}{32} \cong 16.5$), thus in average case the probability to determine a correct sequence of PRNG, ROUND and their associated Lx_i w.r.t. its corresponding Lx_f is $\frac{0.01}{16.5(15.5)}$ which
is equivalent to \( 0.0000391 \). In the same way, the best case and the worst-case probability of determining the above sequence is obtained as \( 0.01 \) and \( 0.00001085 \). At each pass of the ROUND, we have saved the doublet \( Lx_i \) w.r.t. its corresponding \( Lx_f \) to update the secret key for reverse engineering of process forwardPAS and return it as a MUX. Here MUX act as an intermediate key ‘iKEY’ of the whole secret key ‘KEY’ such that \( iKEY \subseteq KEY \).

```c
char* forwardPAS(char **GRID, char* PlainText, int PRNG) {
    int L = strlen(PlainText);
    for(int i = 0 ; i < strlen(PlainText); i++) {
        int ASCII = (int)PlainText[i] * PRNG;
        char TEMP[6];
        sprintf(TEMP, "%d", ASCII);
        if(getLength(ASCII) == 1) {
            GRID[i]=mergeList(PSLB_2, TEMP);
        }
        if(getLength(ASCII) == 2) {
            GRID[i]=mergeList(PSLB_1, TEMP);
        }
        if(getLength(ASCII) == 3) {
            GRID[i]=mergeList("", TEMP);
        }
    }
    int ROUND = getPRNG(10, 20, -1);
    static char MUX[82];
    MUX[0] = (char)(ROUND / 10 + 48);
    MUX[1] = (char)(ROUND % 10 + 48);
    for (int i = 0; i < ROUND; ++i){
        int lx_i = getPRNG(0, L-1, -1);
        int lx_f = getPRNG(0, L-1, lx_i);
        char *TEMP = GRID[lx_i];
        GRID[lx_i] = GRID[lx_f];
        GRID[lx_f] = TEMP;
        MUX[4*i + 2] = (char)(lx_i / 10 + 48);
        MUX[4*i + 3] = (char)(lx_i % 10 + 48);
        MUX[4*i + 4] = (char)(lx_f / 10 + 48);
        MUX[4*i + 5] = (char)(lx_f % 10 + 48);
    }
    return MUX;
}
```

5.7 int* doSplitTriplet(char*, int)

Function doSplitTriplet expect a pointer to char VECTOR_imdt and an integer PTXt as length of intermediate plaintext. The algorithm simply split the vector at the mid and join them from rest two extreme. Vector FRAME_3X makes a frame of consecutive 3 elements of VECTOR_imdt and return it to the caller function.

```c
int* doSplitTriplet(char* VECTOR_imdt, int PTXt) {
    int dx = 0;
    char *VECTOR_split = (char*)malloc(PTXt * 3 * sizeof(char));
    for (int i = (PTXt * 3) / 2 - 1; i >= 0 ; i--){
        VECTOR_split[dx++] = VECTOR_imdt[i];
    }
    for (int i = PTXt * 3 - 1; i >= (PTXt * 3) / 2; i--){
```
VECTOR_split[dx++] = VECTOR_imdt[i];
}
for (int i = 0; i < PTXt * 3; ++i){
    VECTOR_imdt[i] = VECTOR_split[i];
}

int *FRAME_3x = (int*)malloc(PTXt * sizeof(int));
for (int i = 0; i < PTXt; ++i){
    FRAME_3x[i] = ((int)VECTOR_split[3 * i] - 48) * 100 +
        ((int)VECTOR_split[3 * i + 1] - 48) * 10 +
        ((int)VECTOR_split[3 * i + 2] - 48);
}
return FRAME_3x;

5.8 char * listToChar(char **GRID, int L_PTXt)

The function listToChar expects two parameters, a pointer to pointer to char (**char**) GRID and the length of plaintext PTXt. Vector GRID returned by the function forwardPAS is passed to this function along with the length of the original plaintext. It simply linearizes (row-wise) the elements of GRID and returns them as vector VECTOR_imdt.

char* listToChar(char **GRID, int L_PTXt) {
    int index = 0;
    char *VECTOR_imdt = (char*)malloc(3 * L_PTXt * sizeof(char));
    
    for (int i = 0; i < L_PTXt; ++i) {
        for(int j = 0; j < strlen(GRID[i]); j++)
            VECTOR_imdt[index++] = GRID[i][j];
    }
    return VECTOR_imdt;
}

5.9 char * doSplitToShuffle(char *, int)

Function doSplitToShuffle expect VECTOR_imdt returned by the function listToChar and length of original plaintext PTXt. It generates a random number within the range (16, 32) and stores the digits of segment_ROUND at the first two consecutive cells of MUX. It generates two indices doublet_i ∀ 0 ≤ doublet_i ≤ 3L and doublet_k ∀ 0 ≤ doublet_k < 3L − (doublet_i) such that doublet_k > doublet_i, make a slice (S) of elements ranging from doublet_i to doublet_k inclusive. After cutting a slice (S) of length |doublet_k − doublet_i + 1| it places the rest two segments (the left sub-segment ranging from 0 to (doublet_i − 1) and the right sub-segment ranging from (doublet_k + 1) to CTXt-1 of VECTOR_imdt in vector STATE_UPDATE and attach the segmented slice (S) on the vector STATE_UPDATE at the beginning (when GEAR = 1) or at the end (when GEAR = 0) as per the value of boolean GEAR 1 or 0 are shown in Fig. 2 and Fig. 3 respectively.

After shifting a slice into the appropriate position, we have updated the state of vector VECTOR_imdt with the current vector STATE_UPDATE. Each doublet_i and doublet_k assign to MUX for further reverse engineering and return to the called function. Since the operation slice and shift (state update) is repeated up to segment_ROUND times, so the probability to determine a correct segment_ROUND is $\frac{1}{16}$ itself.

At each pass of the shift, the only probability to determine the correct sequence of doublet_k with respect to its corresponding doublet_i is $\frac{1}{3L(3L-1)}$, so the probability to get a correct sequence of segment_ROUND and doublets (doublet_k w.r.t its corresponding doublet_i) becomes $\frac{0.020}{L(3L-1)}$. Since at each pass it attaches the slice (S) as per the value of boolean GEAR, so the final probability to determine the correct sequence of segment_ROUND, doublets and boolean_GEAR of vector STATE_UPDATE become $\frac{0.020}{L(3L-1)}\approx \frac{0.010}{L(3L-1)}$. The average length of the plaintext is 16.5, thus, in the average case the probability to determine the above sequence is $\frac{0.010}{16.5(48.5)}$ which is equivalent to $\approx 0.0000125$. In the same way, the best case and the worst-case probability of determining the above-said sequence are found as $\approx 0.005$ and $\approx 0.00000329$ respectively.
After shifting a slice into the appropriate position, we have updated the state of vector `VECTOR_imdt` with the current vector `STATE_UPDATE`. Each `doublet_i` and `doublet_k` assign to MUX for further reverse engineering and return to the called function. Since the operation slice and shift (state update) is repeated up to `segment_ROUND` times, so the probability to determine a correct `segment_ROUND` is $\frac{1}{16}$ itself.

At each pass of the shift, the only probability to determine the correct sequence of `doublet_k` with respect to its corresponding `doublet_i` is $\frac{1}{3L(3L-1)}$, so the probability to get a correct sequence of `segment_ROUND` and doublets (`doublet_k` w.r.t its corresponding `doublet_i`) becomes $\frac{0.020}{L(3L-1)}$. Since at each pass it attaches the slice ($S$) as per the value of boolean `GEAR`, so the final probability to determine the correct sequence of `segment_ROUND`, `doublets` and `boolean_GEAR` of vector `STATE_UPDATE` become $\frac{0.020}{L(3L-1)GEAR} \approx \frac{0.010}{L(3L-1)}$

The average length of the plaintext is 16.5, thus, in the average case the probability to determine the above
sequence is $\frac{0.010}{16.5(48.5)}$ which is equivalent to $\approx 0.000125$. In the same way, the best case and the worst-case probability of determining the above-said sequence are found as $\approx 0.005$ and $\approx 0.0000329$ respectively.

```c
char* doSplitToShuffle(char* VECTOR_imdt, int L_PTXt) {
    int segment_ROUND = getPRNG(16, 32, -1);

    static char MUX[192];
    MUX[0] = (char)(segment_ROUND / 10 + 48);
    MUX[1] = (char)(segment_ROUND % 10 + 48);

    int index = 0;
    while(index < segment_ROUND) {
        int doublet_i = getPRNG(0, L_PTXt * 3 - 1, -1);
        int doublet_j = getPRNG(0, L_PTXt * 3 - 1, doublet_i);
        if(doublet_i > doublet_j) {
            int TEMP = doublet_i;
            doublet_i = doublet_j;
            doublet_j = TEMP;
        }
        int boolean_GEAR = getPRNG(0, 1, -1);
        char *STATE_imdt = (char*)malloc(L_PTXt * 3 * sizeof(char));
        char *SLICE = (char*)malloc((doublet_j - doublet_i + 1) * sizeof(char));
        int dx = 0;
        for (int i = doublet_i; i <= doublet_j; ++i) {
            SLICE[dx++] = VECTOR_imdt[i];
        }
        if(boolean_GEAR == 0) {
            int j = 0;
            for (int i = 0; i < doublet_i; ++i) {
                if(i == doublet_i && doublet_j == L_PTXt * 3 - 1)
                    break;
                STATE_imdt[j++] = VECTOR_imdt[i];
            }
            for (int i = 0; i < dx; ++i) {
                STATE_imdt[j++] = SLICE[i];
            }
        } else {
            for (int i = 0; i < dx; ++i) {
                STATE_imdt[i] = SLICE[i];
            }
            for (int i = 0; i < L_PTXt * 3; ++i) {
                if(i == doublet_i && doublet_j == L_PTXt * 3 - 1)
                    break;
                if(i == doublet_i) i = doublet_j + 1;
                STATE_imdt[dx++] = VECTOR_imdt[i];
            }
        }
    }
    for (int i = 0; i < L_PTXt * 3; ++i) {
        VECTOR_imdt[i] = STATE_imdt[i];
    }

    MUX[6*index + 2] = (char)(doublet_i / 10 + 48);
}
```
MUX[6*index + 3] = (char)(doublet_i % 10 + 48);
MUX[6*index + 4] = (char)(doublet_j / 10 + 48);
MUX[6*index + 5] = (char)(doublet_j % 10 + 48);
MUX[6*index + 6] = (char)(boolean_GEAR / 10 + 48);
MUX[6*index + 7] = (char)(boolean_GEAR % 10 + 48);
index++;
}
return MUX;
}

5.10 void backwardSplitShuffler(char *, int * [], int, int)

Function backwardSplitShuffler expect a pointer to char as VECTOR_imdt, MUX as a matrix of KEY, segment_ROUND and length of ciphertext CTXt as formal parameters. It performs the reverse engineering of VECTOR_imdt to decipher the encrypted ciphertext. In other words, this function performs the inverse of each operation performed by the function forwardSplitShuffler.

void backwardSplitShuffler(char* VECTOR_imdt, int MUX[][3], int segment_ROUND, int L_CTXt) {
    char BUFFER[1024];
    int COUNTER_=segment_ROUND-1;
    while(COUNTER_ >= 0) {
        if(MUX[COUNTER_][2]==0) {
            int k=0;
            int d=MUX[COUNTER_][1]-MUX[COUNTER_][0]+1;
            if(MUX[COUNTER_][1]==L_CTXt-1) {
                for (int i = 0; i < L_CTXt; ++i){
                    BUFFER[k++]=VECTOR_imdt[i];
                }
            } else {
                int i=0;
                while(i < MUX[COUNTER_][0]) {
                    BUFFER[k++]=VECTOR_imdt[i];
                    i++;
                }
                for (int i = L_CTXt - d; i < L_CTXt; ++i){
                    BUFFER[k++]=VECTOR_imdt[i];
                }
                i=MUX[COUNTER_][0];
                while(i < L_CTXt-d) {
                    BUFFER[k++]=VECTOR_imdt[i];
                    i++;
                }
            }
        } else {
            int k=0;
            int d=MUX[COUNTER_][1]-MUX[COUNTER_][0]+1;
            if(MUX[COUNTER_][0]==0) {
                for (int i = 0; i < L_CTXt; ++i){
                    BUFFER[k++]=VECTOR_imdt[i];
                }
            }
        }
    } else {
        for (int i = d; i < d + MUX[COUNTER_][0]; ++i){
            BUFFER[k++]=VECTOR_imdt[i];
        }
        for (int i = 0; i < d; ++i){
            BUFFER[k++]=VECTOR_imdt[i];
        }
    }
}
BUFFER[k++]=VECTOR_imdt[i];
}

int i = d+MUX[COUNTER_][0];
while(i < L_CTXt) {
    BUFFER[k++]=VECTOR_imdt[i];
i++;
}

for (int i = 0; i < L_CTXt; ++i){
    VECTOR_imdt[i]=BUFFER[i];
}

COUNTER_--;
for (int i = 0; i < L_CTXt; ++i){
    VECTOR_imdt[i]=BUFFER[i];
}

5.11 char * backwardPAS(char *, int, int, int, int)

The function backwardPAS (backward Padding and Shuffling) expect a pointer to character (*char) as VECTOR_imdt updated by the function backwardSplitShuffler, MUX as a key matrix, ROUND, PRNG and length of ciphertext CTxt as formal parameters. Basically, it performs the next level reverse engineering of VECTOR_imdt to decipher the ciphertext CipherTEXT. In general, we can say that, this function is mathematically inverse to the function forwardPAS and can be written as:

```c
char* backwardPAS(char* VECTOR_imdt, int MUX[][2], int ROUND, int PRNG, int L_CTXt) {
    char** GRID=(char**)malloc((L_CTXt/3)*sizeof(char*));
    int k=0;
    for (int i = 0; i < L_CTXt / 3; ++i) {
        char TEMP[4];
        int j=0;
        for (j = 0; j < 3; j++)
        {
            TEMP[j]=VECTOR_imdt[k++];
        }
        TEMP[j]=‘\0’;
        GRID[i]=mergeList(TEMP,””);
    }
    int _COUNTER=ROUND-1;
    while(_COUNTER >= 0) {
        int lx_i = MUX[_COUNTER][0];
        int lx_f = MUX[_COUNTER][1];
        char *temp = GRID[lx_i];
        GRID[lx_i] = GRID[lx_f];
        GRID[lx_f] = _temp;
        _COUNTER--;
    }
    int ASCII_pt[32];
    static char PLAINtext[33];
    for (int i = 0; i < L_CTXt / 3; ++i)
    {
        ASCII_pt[i]=atoi(GRID[i]);
        ASCII_pt[i]/=PRNG;
    }
    return ASCII_pt;
}
```
6. Driver Functions of SAR 256

6.1 char * encryptPT(char *, char *)

The function encryptPT expect two (*char) as concrete plaintext and secret key KEY (KEY initially set to NULL), defines a GRID to update the secret key KEY_Secret (of size vary from 288-560 bits) and an integer PRNG. First, it invokes the function forwardPAS by passing arguments GRID as a NULL, a concrete plaintext as PlainTEXT and an integer PRNG within the range [1, 3]. In this process, the intermediate key is generated in MUX and is return to the called function that is further saved to KEY_buffer_1. At the next phase when function gridToList gets called by passing the concrete intermediate GRID and size of plaintext PTXt, it linearizes the elements of GRID (into a group of 3 each of 8 bits exactly) into a list and returns to a pointer to char VECTOR_imdt. On invoking the function forwardSplitShuffler by passing the intermediate vector VECTOR_imdt (returned by the function call gridToList) and size of plaintext PTXt as formal parameters. It split the vector and shuffle the slice up to ROUND times and return the next intermediate key as MUX and is stored in KEY_buffer_2. After that, it performs the intermediate key extraction of MUX from KEY_buffer_1, KEY_buffer_2 and arranges them sequentially to design the symmetric key KEY_Secret and finally assign it to the variable KEY (formal parameter of function encryptPT) [33]. At the end of this encryption process, it calls to function getFrame3X to frame the VECTOR_imdt into a sequence of frames of size 3 and arrange them sequentially into a static char CipherTEXT. After placing a null delimiter at the end of the ciphertext, the function returns the encrypted PlainTEXT as CipherTEXT.

```c
char* encryptPT(char* PLAINtext, char *KEY) {
    char** GRID = (char**)malloc(strlen(PLAINtext) * sizeof(char*));
    char KEY_Private[512];

    int PRNG = getPRNG(2, 3, -1);
    char* KEY_buffer_1 = forwardPAS(GRID, PLAINtext, PRNG);
    char* VECTOR_imdt = listToChar(GRID, strlen(PLAINtext));
    char* KEY_buffer_2 = doSplitToShuffle(VECTOR_imdt, strlen(PLAINtext));

    KEY_Private[0] = '0';
    KEY_Private[1] = (char)PRNG + 48;

    int index = 2;
    int ROUND = ((int)KEY_buffer_1[0] - 48) * 10 + (int)KEY_buffer_1[1] -48;

    for (int i = 0; i < ROUND * 4 + 2; ++i){
        KEY_Private[index++] = KEY_buffer_1[i];
    }

    int segment_ROUND = ((int)KEY_buffer_2[0] - 48) * 10 + (int)KEY_buffer_2[1] -48;

    for (int i = 0; i < segment_ROUND * 6 + 2; ++i){
        KEY_Private[index++] = KEY_buffer_2[i];
    }

    if(4-index%4==1) {
        KEY_Private[index] = '0';
        index++;
    }
    if(4-index%4==2) {
        KEY_Private[index]=KEY_Private[index+1]='0';
        index=index+2;
    }

    return PLAINtext;
```

Shabbir Hassan et al. / Indian Journal of Computer Science and Engineering (IJCSE)
if(4-index%4==3) {
    KEY_Private[index]=KEY_Private[index+1]=KEY_Private[index+2]='0';
    index=index+3;
}
KEY_Private[index]='\0';

int k=0;
for (int i = 0; i < index / 4; i++){
    char* test=getHexa(((int)KEY_Private[4 * i] - 48) * 1000 +
    ((int)KEY_Private[4 * i + 1] - 48) * 100 +
    ((int)KEY_Private[4 * i + 2] - 48) * 10 +
    ((int)KEY_Private[4 * i + 3] - 48),0);
    for(int j = 0; j < 4; j++) {
        KEY[k++]=test[j];
    }
}
int* BUFFER = doSplitTriplet(VECTOR_imdt, strlen(PLAINtext));

static char CIPHERtext[1024];
k=0;
for(int i = 0; i < strlen(PLAINtext); i++) {
    char* hex=getHexa(BUFFER[i],1);
    for (int index = 0; index < 3; ++index){
        CIPHERtext[k++]=hex[index];
    }
}
CIPHERtext[k]='\0';
return CIPHERtext;

6.2 char * decryptCT(char *, char *)

The function decryptCT expecting two formal parameters the ciphertext (the encrypted message of 256 bit long) and the secret key, in hexadecimal form. Matrix KEY_BUFFER_1 and KEY_BUFFER_2 same the intermediate state of the extracted secret key. An array of character “Buffer” (of size 1024) consists the decimal value equivalent to each first three consecutive triplets of the hexadecimal cipher text. All these conversions are done within the scope of a loop with condition index < L_CTXt. After converting each three consecutive hexadecimal value of the ciphertext the buffer is terminated by ‘\0’. Similarly, first four consecutive character of the secret key ‘KEY’ gets extracted and converted into equivalent decimal in KEY_imdt. The value of PRNG and ROUND (during encryption phase) are extracted and used to extract MITRN Lx_i and Lx_j and is saved into KEY_buffer_1. After finding the SEGMENT_ROUND and boolean GEAR to perform the reverse engineering of operation slice and shift. On invoking function getFrame3x, the buffer has splitted into mid, and merge from the two opposite extremes [34]. On calling the function backwardSplitShuffle we achieved the reverse action of function forwardSplitShuffler, at the end of the decryption phase, backwardPAS function gets called that performs the reverse action of the function forwardPAS and return the expected plaintext [35].

char* decryptCT(char *CIPHERtext, char *KEY) {
    int PRNG, ROUND, segment_ROUND;
    int KEY_buffer_1[100][2], KEY_buffer_2[100][3], L_CTXt=strlen(CIPHERtext);
    char BUFFER[1024];
    int i = 0, index = 0;

    while(index < L_CTXt) {
        char _TEMP[4];
        for (int j = 0; j < 3; ++j){
            _TEMP[j] = CIPHERtext[j + i];
        }
        _TEMP[3] = '\0';

        char *DECimal = getDecimal(_TEMP,1);
        }

return CIPHERtext;
}
for (int j = 0; j < 3; ++j)
    BUFFER[index++] = DECimal[j];

i = i + 3;
}
BUFFER[index] = '\0';

static char KEY_imdt[1024];
int _COUNTER = 0; i = 0, index = 0;
while (_COUNTER < 70) {
    char _TEMP[5];
    for (int j = 0; j < 4; ++j){
        _TEMP[j] = KEY[j + i];
    }
    _TEMP[4] = '\0';
    char *DECimal = getDecimal(_TEMP, 0);
    for (int j = 0; j < 4; ++j){
        KEY_imdt[index++] = DECimal[j];
    }
    i = i + 4;
    _COUNTER++;
}

index = 1;
PRNG = (int)KEY_imdt[index++] - 48;
ROUND = (int)KEY_imdt[index++] - 48;
ROUND = ROUND * 10 + (int)KEY_imdt[index++] - 48;

for (int j = 0; j < ROUND; j++){
    KEY_buffer_1[j][0] = (int)KEY_imdt[index++] - 48;
    KEY_buffer_1[j][0] = KEY_buffer_1[j][0] * 10 + (int)KEY_imdt[index++] - 48;
    KEY_buffer_1[j][1] = (int)KEY_imdt[index++] - 48;
    KEY_buffer_1[j][1] = KEY_buffer_1[j][1] * 10 + (int)KEY_imdt[index++] - 48;
}

segment_ROUND = (int)KEY_imdt[index++] - 48;
segment_ROUND = segment_ROUND * 10 + (int)KEY_imdt[index++] - 48;

for (int j = 0; j < segment_ROUND; j++){
    KEY_buffer_2[j][0] = (int)KEY_imdt[index++] - 48;
    KEY_buffer_2[j][0] = KEY_buffer_2[j][0] * 10 + (int)KEY_imdt[index++] - 48;
    KEY_buffer_2[j][1] = (int)KEY_imdt[index++] - 48;
}
KEY_buffer_2[j][1] = KEY_buffer_2[j][1] * 10 + (int)KEY_imdt[index++]

- 48;

KEY_buffer_2[j][2] = (int)KEY_imdt[index++] - 48;

KEY_buffer_2[j][2] = KEY_buffer_2[j][2] * 10 + (int)KEY_imdt[index++]

- 48;

} doSplitTriplet(BUFFER,L_CTXt/3);

backwardSplitShuffle(BUFFER,KEY_buffer_2,segment_ROUND,L_CTXt);

return backwardPAS(BUFFER, KEY_buffer_1, ROUND,PRNG, L_CTXt);

7. Entry Point of SAR 256

An entry point is a place in a programme where the execution of the programme begins and the code has access to command line parameters. This function act as an entry point of two functions encryptPT and decryptCT. The complete signature of this function is char * getEntryPoint(char *, char, char *)) that expect a pointer to char as message (either plaintext or ciphertext), an action flag 'e' for encryption and return ciphertext else decryption and return plaintext, and return (pass by reference) the secret key for the desirable function call.

char* getEntryPoint(char *message,char action,char *KEY) {

if(action=='e')
    return encryptPT(message, KEY);
else
    return decryptCT(message, KEY);

}

8. Test Vectors of SAR 256

Based on the test vector as key and plaintext, the model SAR 256 has been tested for encrypting the same message (plaintext) several times with different key with the same plaintext, each time different ciphertext has obtained. Both keystream and output ciphertext is represented in Hexadecimal and Unicode characters are given below in Table 1.

<table>
<thead>
<tr>
<th>#</th>
<th>( P_T )</th>
<th>Generated Ciphertext (C(_T)) with Unicodes</th>
<th>Private Key (K)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>plaintext</td>
<td>13F13419327914D1B119D3540CE</td>
<td>5DFA1F60198260032100080260000202BD0259000201960710070900CF00010321007706703E803FA06F07D105F2007890C403F600110835039400110760D26A00710641025F0009057900E00000</td>
</tr>
<tr>
<td>15D0D40DE1400D40027008E016</td>
<td>00D501FB02C006601910325000700C902580010197013012D0007090C4052A0060476C045F006400C906800160FC091607507805260068051405900150961013506C076C03F9006805DC039BD006802BD02D00D687804BF007D0A28091406A07D0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1801491AF17C14E15E132146085</td>
<td>013701F60198260032100080260000202BD02590002019609DB096002D400002076D03F900000641026B0012076D04C0006E04B102C5007709C4052A004076C045F006400C90068011608FC091607507805260068051405900150961013506C076C03F9006805DC039BD006802BD02D00D687804BF007D0A28091406A07D0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Average encryption time in GNU C compiler is ≈ 0.0134 sec. The code is implemented in macOS Mojave version 10.14.6 Intel Core i5 with following Run Time Environment (RTE).
The cipher SAR 256 has tested on Xilinx ISE 14.2 under CodeLite environment in macOS Mojave with the above parameters.

9. Conclusion and Future Work

In this paper a domain encryption algorithm SAR 256 is proposed. The algorithm is implemented in 'C' and has tested under Xilinx 14.2 environment and found to be safe & secure against some well-known cryptographic attacks. Since it has a fixed length ciphertext output w.r.t. a plaintext input, thus the implementation of SAR 256 can also be substituted as a hash. It has observed that the ciphertext produced by SAR 256 is cryptographically secure against various cryptographic attacks like timing attack, side channel attack, cube attack, linear masking attack and correlation attack. The cipher has been tested against domain encryption parameters and has found to be unbiased to meet the encryption standard defined by the NIST to encrypt sensitive data such as cookies, e-signing key, login credentials, ATM PIN, iNode, SSN, WSN’s node etc. The cipher is mainly suitable for resource-constrained devices due to their less computation time, low hardware requirement and hence it found to be more optimal as compared to other domain encryption algorithms. This cipher SAR 256 can be implemented to encrypt files/data on a server, cloud server, blockchain, wireless sensor networks and other such framework. The only limitation is that it has quite a large key length for each plaintext length which can be resolve in future work.

10. References


Authors Profile

Dr. Shabbir Hassan is a “Sun Certified Java Programmer (SCJP)” currently working as Assistant Professor at Centre for Distance and Online Education, Aligarh Muslim University, Aligarh. He holds Master in Computer Science and Applications (MCA) and Ph.D. at Department of Computer Science, Aligarh Muslim University. His thrust area is “Analysis and Design of Lightweight Stream Cipher” and area of interest includes Applied Mathematics, Analysis and Design of Algorithms, Dynamic Programming, Network Security and Cryptography. He has qualified UGC-National Eligibility Test (NET) and has availed Junior Research Fellowship (JRF) during the Research Work. Throughout his career, he has been involved in innovative Software Development and Academic Teaching of Computer Science subjects like C, JAVA, Python, Data Structure, Operating System, Automata Theory and Computer Networks. He has presented his research work in several National and International IEEE Conferences and marked his active participation in many Conferences, Workshop and Symposia. His research papers have published in many reputed peer-reviewed Journals of International repute like Springer, Elsevier, IASR, InderScience and Scopus Indexed Database. Apart from the Academic Research and Software Development, he is enriched with the passion of poetry and philosophy and engages himself in social works.

Dr. Arshad Iqbal has been awarded “Best Research Award” (NESIN-2020 Awards) for his contribution and achievement in innovative research. He has been working in the area of machine learning and image processing. He received his Ph.D. and Master of Computer Science and Applications (MCA) degree from Aligarh Muslim University, Aligarh. Dr. Iqbal started his career as software professional and worked in IT industry for four years. Later he joined as Information Scientist at Aligarh Muslim University, Aligarh. He is currently working as Assistant Professor in Computer Science at K. A. Nizami Centre for Quranic Studies, AMU, Aligarh since July 2011 and is actively involved in teaching and research activities. Dr. Iqbal has his active participation on managing and running Language Lab and website of their Center.

Mr. Rehan Raza received his degree in Bachelor of Computer Application (BCA) from Vinoba Bhave University, Hazaribag and currently pursuing Master of Computer Application (MCA) from National Institute of Technology (NIT), Calicut, Kerala, India. Beside Academic performance and Software Development, he is enriched by his passion for computer programming and creates abstract model for a concrete problem. His areas of Interests include Problem Solving, Design and Analysis of Algorithm, Computer Network, Database Management System and Automata Theory. His current research interests include Cryptography and machine learning (face and pattern recognition). Beside from Automatic Attendance System using Face Recognition techniques, Mr. Raza has developed a secure round-based credential encryption system, license key generator and validator (by using C) which can protect applications from unauthentic user and software counterfeit at the age of 18. Other web-based applications for user blog interaction and text response have also been developed hosted freely. Beside it, Raza has also developed android based application like DC Slot/Project Evaluation, Slot Booking System and other console-based applications.