symKrypt: A General-purpose and Lightweight Symmetric-Key Cryptography

Ripon Patgiri, Senior Member, IEEE

Abstract—Symmetric-key cryptography is used widely due to its capability to provide a strong defense against diverse attacks; however, it is prone to cryptanalysis attacks. Therefore, we propose a novel and highly secure symmetric-key cryptography, symKrypt for short, to defend against diverse attacks and provide absolute security. Our proposed algorithm changes private keys in each block of communication, i.e., symKrypt uses multiple private keys to encrypt a single block of a message. Moreover, symKrypt keeps secret the bit mixing of the original message with the private keys. Also, the number of private keys is kept secret. In addition, the private keys are generated dynamically based on the initial inputs using a pseudo-random number generator which is highly unpredictable and secure. In this article, we theoretically analyze the capabilities of symKrypt and provide experimental demonstration using millions of private keys to prove its correctness. Furthermore, we demonstrate the proposed pseudo-random number generator algorithm experimentally in NIST SP 800-22 statistical test suite. Our propose pseudo-random number generator passes all 15 tests in the said test suite. symKrypt is the first model to use multiple private keys in encryption yet lightweight and powerful.

Index Terms—Security; Security Protocol; Encryption; Cryptography; Symmetric Cryptography; Diffie-Hellman Cryptography; Random Number Generator; Computer Networking.

1 INTRODUCTION

Symmetric-key cryptography is the most secure cryptography protocol. Therefore, there are diverse variants of symmetric-key cryptography, particularly, Twofish [1], Serpent, AES (Rijndael) [2], Salsa20 [3], ChaCha20 [3], Blowfish, Kuznyechik, DES, 3DES, Skipjack, and IDEA [4]. Diverse new variants are suggested by many researchers [5], [6], [7], [8], [9], [10]. Also, diverse platforms are available, which requires modification of symmetric-key cryptography techniques [11], [12], [13]. Recent analysis on symmetric-key cryptography suggests many possible attacks [14], [15], [16], [17], [18].

There are diverse attacks on symmetric-key cryptography, and therefore, it demands a symmetric-key cryptography algorithm that shows strong resistance to the attacks. It also demands general-purpose symmetric-key cryptography that can be applied in diverse domains, for instance, the Internet of Medical Things. Moreover, most modern devices require lightweight symmetric-key cryptography, particularly securing small IoT devices [19]. Moreover, Edge Computing is emerging, and therefore, there are diverse cryptographic requirements. For instance, Edge Nodes and Cloud Computing can communicate with high-sized key cryptography. But the Edge Devices are low-, mid-, and high-powered. Therefore, the large key size of the block cipher algorithm becomes expensive for low-powered powered devices. Therefore, the key size requirements range from 16-bits to 2048 bit; even more. For instance, smartphones are highly capable devices but not wearable devices. Consequently, it demands general-purpose symmetric-key cryptography, which can provide absolute security. A general-purpose symmetric key cryptography is required to suffice all those requirements. Furthermore, there are various attacks on symmetric-key cryptography due to single-keyed encryption. We propose a general-purpose symmetric-key cryptography algorithm called symKrypt, which is a straightforward solution and lightweight cryptography yet very powerful to address the above issues. symKrypt relies on Diffie-Hellman algorithm [20] for key sharing, where symKrypt computes two secret keys, namely, shared secret key and shared secret seed value (Elliptic-curve cryptography [21], [22] or Elliptic-curve Diffie-Hellman (ECDH) algorithm [23] can also be used). Once the secret keys are computed, symKrypt does not require Diffie-Hellman key exchange for the entire session. The secret keys are used to generate the private keys, and these are not used in encryption. symKrypt uses multiple private keys in each round of encryption of a block of a message. Similarly, it uses multiple private keys in the encryption of each block of a message. The private keys are computed asymmetrically by the sender and receiver; but both has to maintain the order of the private keys for each message. Therefore, the private keys play a vital role in defending diverse attacks in symmetric-key cryptography. These private keys are generated dynamically using a pseudo-random number generator algorithm that is highly unpredictable for adversaries. symKrypt performs rotation, and it never shares the information of rotation $r$ and the total number of private keys $t$. Therefore, symKrypt creates a strong deterrence against the attackers. The $t$ and $r$ are computed dynamically, and also, the value of $r$ changes in each iteration. symKrypt also protects the left or right rotation information, and thus, the adversaries do not have any clue on the types of the rotations.

The main contributions of this article are outlined below:
We propose a novel and highly secure symmetric-key cryptography algorithm, called symKrypt for short, based on dynamic private keys.

symKrypt changes its private key in each iteration of a block of message. Also, it changes the private keys in each block of a message.

The private keys are generated by a pseudo-random number generator which is highly unpredictable. The shared secret key and shared secret seed value are replaced with the private keys.

The total number of private keys is kept secret. Moreover, the rotation information is kept secret, and it is dynamically generated.

We propose a pseudo-random number generator to generate the private keys which rely on the non-cryptographic string hash function.

symKrypt demonstrates its strong resistance against many attacks, including cryptanalysis attacks.

This article demonstrates the capabilities of symKrypt theoretically and experimentally. To the best of our knowledge, symKrypt is the first of its kind to use multiple keys in encryption. The changes of the private keys create a resistance against cryptanalysis attacks. This article also demonstrates how to change its private keys and generate the dynamic private keys in each round. Moreover, we show how it helps in defending many attacks by keeping secret about the rotation information and the total number of private keys. The adversaries have no clue in gaining these information. Therefore, symKrypt can provide a strong deterrence to any possible attacks of symmetric-key cryptography yet lightweight.

This article is organized as follows- Section 2 describes the proposed system and elaborates the proposed algorithms in detail. Section 4 analyzes the proposed system theoretically and demonstrates the resistance of symKrypt against any attacks. Section 5 proves the proposed system experimentally. Finally, Section 6 concludes the article.

2 PROPOSED SYSTEMS

We propose a novel and highly secure symmetric-key cryptography algorithm, symKrypt (SYMmetric-Key CRYPTOgraphy) for short. The key objective of our proposed systems is to provide strong resistance against the possible attacks on symmetric-key cryptography. Therefore, we have a few assumptions, and these assumptions are outlined below.

- At the given time, the sender and the receiver must be active.
- Our proposed algorithm relies on the Diffie-Hellman key exchange protocol, and thus, we omit a detailed analysis of the same. Also, we assume that the symmetric-key exchange protocol is secure enough to protect against any attacks on key sharing.
- We assume that sender and receiver are valid. Therefore, the man-in-the-middle attack is out of scope. Moreover, our proposed system does not deal with DDoS attacks.

### Table 1: Parameters and their states of symKrypt

<table>
<thead>
<tr>
<th>Parameter</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared secret key SK</td>
<td>Secret &amp; Static</td>
</tr>
<tr>
<td>Shared secret seed value S</td>
<td>Secret &amp; Static</td>
</tr>
<tr>
<td>Private key P</td>
<td>Secret &amp; Dynamic</td>
</tr>
<tr>
<td>Total number of private keys t</td>
<td>Secret &amp; Dynamic</td>
</tr>
<tr>
<td>Minimum/Maximum number of public key c</td>
<td>Public</td>
</tr>
<tr>
<td>Number of circular shift rotation r</td>
<td>Secret &amp; Dynamic</td>
</tr>
<tr>
<td>Left or Right shift rotation</td>
<td>Secret &amp; Dynamic</td>
</tr>
<tr>
<td>Bit size of the key β</td>
<td>Public</td>
</tr>
</tbody>
</table>

3 PRELIMINARY

Table 1 shows the essential parameters of our proposed algorithm and their state. Most of the parameters are secret. The shared secret key and the shared secret seed value are kept secret, and used only once to generate the first bit of the first private key. The key-exchange take place only once. The private keys are changed and generated dynamically, and these are kept secret. Similarly, the r and t are kept secret, and generated dynamically. Moreover, the rotation decision is also made dynamically. However, the minimum/maximum number of public key range and bit size of the keys are made public.

3.1 Description

Let A and B be the sender and receiver, respectively. Let A and B mutually agree on a shared secret key SK and a shared secret seed value S securely using the Diffie-Hellman algorithm. The sender A divides the message m into several blocks. The blocks are encrypted and sent to the B. The SK and B are used to generate private keys for encryption and let the private keys be \( P = \{ P_1, P_2, P_3, \ldots, P_t \} \) and the bit size of the private keys be the \( \beta \). A block of a message is encrypted by all these randomly generated private keys as given in Equation (1).

\[
\begin{align*}
\zeta &= P_1 \oplus m \\
&\quad \text{rotate}(\zeta, (P_1 \mod \beta)) \\
\zeta &= P_2 \oplus \zeta \\
&\quad \text{rotate}(\zeta, (P_2 \mod \beta)) \\
\zeta &= P_3 \oplus \zeta \\
&\quad \text{rotate}(\zeta, (P_3 \mod \beta)) \\
&\quad \vdots \\
\zeta &= P_t \oplus \zeta \\
&\quad \text{rotate}(\zeta, (P_t \mod \beta))
\end{align*}
\]

The \( \zeta \) is rotated using circular shift right/left operation depending on the last bit of the private key. The total number of rotations is calculate using modulus operation as shown in Equation (1). The circular shift right/left and the total number of rotations in the circular shift right/left rotation are kept secret.

In the decryption process, the receiver B receives ciphertext \( \zeta \) and decrypts using secret private keys \( P \) as given in Equation (2). The sender’s private keys must be same as the receiver; otherwise, the decryption process fails. Firstly, the \( B \) derives the total number of rotations and performs the ciphertext exactly the opposite rotation of the encryption.
process. Secondly, the rotated ciphertext is decrypted using the private keys shown in Equation (2).

\[
\text{rotate}(m, (P_t \mod \beta)) = \begin{cases} 
\text{rotate}(m, (P_{(t-1)} \mod \beta)) \\
\text{rotate}(m, (P_{(t-2)} \mod \beta)) \\
\vdots \\
\text{rotate}(m, (P_1 \mod \beta)) \\
m = P_t \oplus \zeta
\end{cases}
\]

(2)

The decryption process is the opposite of the encryption process. Therefore, the private keys are XORed with the ciphertext in descending order.

3.2 Encryption process

Algorithm 1 Algorithm for encryption.

1: procedure \text{SYMENC}(m, SK, S, \beta) 
2: \hspace{1em} t = (SK \mod \beta) + c 
3: \hspace{1em} \zeta = m 
4: \hspace{1em} i = 1 
5: \hspace{1em} r = S \mod \beta 
6: \hspace{1em} \text{odd} = SK \land 1 
7: \hspace{1em} \text{while } i \leq t \text{ do} 
8: \hspace{2em} P_i = \text{GENPRNG}(SK, S, \beta) 
9: \hspace{2em} S = P_i 
10: \hspace{2em} r = P_i \mod \beta 
11: \hspace{2em} \zeta = \zeta \oplus P_i 
12: \hspace{2em} \text{if } P_i \land 1 = 1 \text{ then} 
13: \hspace{3em} \zeta = \text{CIRCULARROTATELEFT}(\zeta, r) 
14: \hspace{2em} \text{else} 
15: \hspace{3em} \zeta = \text{CIRCULARROTATERIGHT}(\zeta, r) 
16: \hspace{2em} \text{end if} 
17: \hspace{2em} i = i + 1 
18: \hspace{2em} \text{end while} 
19: \hspace{1em} \text{Send } \zeta \text{ to the receiver} 
20: \hspace{1em} \text{return } P 
21: \text{end procedure}

The sender A wants to send a message to receiver B, and therefore, both parties agree on the shared secret key SK and the shared secret seed value S, where SK \neq S. Algorithm 1 requires the original message m, shared secret key K, shared secret seed value S, and the bit size of encryption \beta as the input parameters. The bit size of both \beta must be equal to or greater than m, and \beta be of any size (\beta \geq m or 128 bits by default and the size of \beta can be 16 bits, 32 bits, 64 bits, 128 bits, 256 bits, 512 bits, and so on, as per the requirement of the user’s application). The bit size \beta is public. Let t and r be the total number of rounds and the total number bits’ rotation. Algorithm 1 calculates t = SK \mod \beta + c and r = P \mod \beta which are not known to adversaries. The c is a constant value which means that at least c rounds of XOR operation need to be performed, and it is made public. For instance, if c = 5, then at least five rounds of XOR operations need to be performed between the original message and the five private keys. Alternatively, the t represents the total number of private keys to be used in encryption and decryption. The private keys are generated using a pseudo-random number generator (PRNG) algorithm. Moreover, the seed value changed to the generated private key in each iteration. The generated private key \mathcal{P} is XORed with the original message m. The ciphertext \zeta is rotated by the circular shift left/right rotation operation r times in each iteration. If the shared secret key is odd, then performs circular shift left operation; otherwise, performs circular shift right operation. The adversary is unable to know whether to perform a circular shift right or left. The sender A sends the ciphertext \zeta to the B over insecure channel.

3.3 Decryption process

Algorithm 2 Algorithm for decryption.

1: procedure \text{SYMDXC}(\zeta, SK, S, \beta) 
2: \hspace{1em} t = (SK \mod \beta) + c 
3: \hspace{1em} m = \zeta 
4: \hspace{1em} r = S \mod \beta 
5: \hspace{1em} \text{odd} = SK \land 1 
6: \hspace{1em} i = 1 
7: \hspace{1em} \text{while } i \leq t \text{ do} 
8: \hspace{2em} P_i = \text{GENPRNG}(SK, S, \beta) 
9: \hspace{2em} S = P_i 
10: \hspace{2em} r = P_i \mod \beta 
11: \hspace{2em} \zeta = \zeta \oplus P_i 
12: \hspace{2em} \text{if } P_i \land 1 = 1 \text{ then} 
13: \hspace{3em} \zeta = \text{CIRCULARROTATERIGHT}(\zeta, r) 
14: \hspace{2em} \text{else} 
15: \hspace{3em} \zeta = \text{CIRCULARROTATELEFT}(\zeta, r) 
16: \hspace{2em} \text{end if} 
17: \hspace{2em} i = i + 1 
18: \hspace{2em} \text{end while} 
19: \hspace{1em} \text{Write } \zeta 
20: \hspace{1em} \text{return } P 
21: \text{end procedure}

The receiver B receives the ciphertext \zeta, and decrypts the ciphertext using Algorithm 2. Algorithm 2 is similar to Algorithm 1 except the rotation operation and its order. The circular shift rotation is performed in each iteration after the XOR operation in the encryption process. In contrast, the circular shift rotation is performed in each iteration before the XOR operation in the decryption process. Moreover, the rotation direction should be opposite to each other; for instance, if encryption performs circular shift rotate left, then decryption has to perform a circular shift right operation depending on the private key. In short, the decryption operation has to perform the reverse order of the encryption operation.
3.4 Pseudo-random Number Generator

The necessary conditions of pseudo-random number generator for symKrypt are outlined below:

- The PRNG must be able to produce a highly random, unpredictable, and cryptographically secure key.
- The PRNG must pass all the 15 tests of NIST SP 800-22.
- The generated random key must be reproducible for the correct inputs.

The PRNG must satisfy the above conditions to decrypt the encrypted message correctly; otherwise, the symKrypt fails. symKrypt heavily depends on PRNG with the abovementioned conditions. Our proposed PRNG depends on the non-cryptographic string hash functions, and the string hash function mixes the bits and produces unpredictable LSB bits. The LSB bit is extracted to form a private key. Algorithm 3 iterates β times and forms private keys of β bits. Algorithm 3 takes shared secret key, shared secret seed value, and bit information as the initial inputs. It can produce the same output for a given input set at any given time. However, the output is truly random, tested in NIST SP 800-22 statistical test suite [24], [25].

3.5 Sequence of messages’ blocks

Algorithm 4 Encryption of the blocks of messages by symKrypt.

```plaintext
1: procedure SYMKRYPTENC(ψ, SK, S, β)
2:    i = 1
3:    while i ≥ ψ do
4:        PK = SYMEDC(ψi, SK, S, β)
5:        S = S ⊕ SK
6:        SK = PK
7:        i = i + 1
8:    end while
9:    end procedure
```

Algorithm 5 Encryption of the blocks of messages by symKrypt.

```plaintext
1: procedure SYMKRYPTDEC(ψ, SK, S, β)
2:    i = 1
3:    while i ≥ ψ do
4:        PK = SYMEDC(ψi, SK, S, β)
5:        S = S ⊕ SK
6:        SK = PK
7:        i = i + 1
8:    end while
9:    end procedure
```

Algorithm 1 and Algorithm 2 demonstrate the encryption and decryption of a single block. It returns a private key, and it is required to encrypt or decrypt the next blocks of messages. Let the blocks of message be the $m_1, m_2, m_3, \ldots, m_\psi$. Algorithm 4 and Algorithm 5 demonstrate the encryption and decryption of entire blocks of messages in a communication between $A$ and $B$, respectively. In encryption or decryption, the private keys are changed in each round or iteration. Moreover, the shared secret key $SK$ and the shared secret seed value $S$ are used only once to generate the first bit of the first private key. Later, the shared secret key and shared secret seed value are replaced. Moreover, the value of $t$ and $r$ change in each block’s encryption or decryption.

4 Analysis

symKrypt is a symmetric-key cryptography algorithm that depends on symmetric key exchange algorithms. There are a few symmetric-key exchange algorithm, namely, Diffie-Hellman [20], Elliptic-key cryptography [21], [22] and ECDH [23]. We choose one of the most secure symmetric key exchange algorithm, Diffie-Hellman algorithm. The security of symKrypt depends on a pseudo-random number generator that generates private keys. Moreover, Table 1 shows that the minimal number of parameters are made public, preventing the attackers from gaining information on the ciphertext.

4.1 Time Complexity

The time complexity of a block of a message in symKrypt is constant. The time complexity of Algorithm 1 depends on the time complexity of Algorithm 3. The time complexity of Algorithm 3 depends on the hash function and the number of bit size requirements. The has function’s time complexity depends on the input string length, for instance, $l$. Algorithm 3 iterates β times, and hence, the total time complexity of the Algorithm 3 is $O(\beta \times l)$. In practical applications, the bit size ranges from 16 bits to 2048, quite a small number. Also, the string length is similar to the bit sizes. Therefore, we can easily rewrite the time complexity of Algorithm 3 intuitively, and it is $O(1)$. The total time complexity of Algorithm 1 is $O(t \times \beta \times l)$. In a practical scenario, the value of $t$ ranges from 10 to 100. Therefore, the total time complexity of Algorithm 1 and Algorithm 2 is $O(1)$. The time complexity of symKrypt depends on Algorithm 4 and Algorithm 5. Algorithm 4 and Algorithm 5 depends on the number of blocks. The total number of blocks is $\psi$. Therefore, the total time complexity is $O(\psi \beta \times l) \approx O(\psi)$. 
4.2 Correctness of symKrypt

We exploit the XOR property to perform encryption. The plaintext $m$ is XORed with several private keys. It also performs several rotation operations. XOR operation produces zero if two bits are equal; otherwise, it produces one. Similarly, XOR operation produces zero for two same keys. Therefore, symKrypt can produce zero in the encryption process to transmit to the receiver. The zero value is correct, and it can be sent to the receiver. The receiver can retrieve the original message from the zero value if the shared secret key and shared secret seed value are correct. It is also possible that symKrypt produces ‘1’ in all bit fields. Moreover, symKrypt can also produce single-digit output or two digits output in the encryption process. symKrypt can produce any output in encryption. The original message can be decrypted from the ciphertext in any condition.

The message $m$ is XORed with set of keys $P$. For example, Equation (3) encrypts the message $m$.

$$
\zeta = m \oplus P_1
$$

Equation (4) decrypts the encrypted message by Equation (3).

$$
m = \zeta \oplus P_1
$$

Let us assume that the encryption process using multiple keys as shown in Equation (5).

$$
\zeta = m \oplus P_1 \oplus P_2 \oplus P_3 \oplus \ldots \oplus P_t
$$

Equation (5) shows that encrypting a message with multiple keys cannot protect the attackers if we do not mix the message with private keys properly. Equation (5) is easy to decrypt by any novice attacker. Therefore, we address this issue by rotation in each iteration. The rotation information is kept secret, and therefore, there is no way to trace back the original message from the ciphertext for the adversaries. Exclusively, the intended user can decrypt the original message even if the encryption process produces a single or double-digit number.

Moreover, the correctness of symKrypt also depends on the rotation. Let $m$ is circular shift left rotated by $r$ times. It requires a circular shift right rotated by $r$ times to produce $m$ correctly. The value of $r$ is dynamic and changes in each iteration. In addition, the left/right rotation is also dynamic and depends on the private keys. The exact reverse order of the encryption process can decrypt the original message from the ciphertext. Therefore, the intended user can exclusively decrypt the original message. The rotation process creates a strong resistance against any attacks. Moreover, the value of $t$ also provides a good defense against attacks.

4.3 Brute-force attacks

A brute-force attack is the most famous attack; however, many symmetric-key cryptographies have already taken preventive measures to secure communication. Similarly, symKrypt provides a strong resistance to brute-force attacks. Let us assume that the $c = 0$ and $SK \mod \beta = 0$, then the value of $t$ is zero in $t = (SK \mod \beta) + c$ and there is no encryption. A raw message is sent to the receiver. In this case, symKrypt fails. Another instance, if $c = 1$ and $t = 0$, then symKrypt also fails. Therefore, we suggest the value of $c$ between 10 and 100 for the practical scenario. In the worst-case scenario, the value of $t$ is 227 where $SK \mod \beta = 127$ in 128 bits key size and $c = 100$. It implies that symKrypt has to perform encryption or decryption using 227 private keys; however, an adversary is unable to know the total number of private keys. In addition, the private keys are highly unpredictable, and thus, it also provides a strong deterrence against any attacks.

Let us assume that the adversary knows the value of $t$, which is 10. Let us also assume that the same adversary knows the shared secret key $SK$. Theoretically, the adversary has the most information of the communication; however, the adversary does not know the seed value of $S$. The probability of gaining the correct information about the seed value is $\frac{1}{2^r}$, where $\beta$ can be $2^6, 2^7, 2^8, \ldots$ bits and it is made public. Our assumption was the adversary knows the shared secret key $SK$. If the adversary does not know the $SK$, then the probability of getting correct $SK$ is $\frac{1}{2^r}$. Therefore, the total probability break of both the secret information using brute-force attack (BF) is given in Equation (6).

$$
Pr(BF) = Pr(SK) \cap Pr(S)
$$

$$
= \frac{1}{2^r} \times \frac{1}{2^\beta}
$$

The $Pr(BF)$ is the probability of breaking security using the brute-force method, and the two secret information are independent of each other. Therefore, the probability of not getting secret information is $(1 - \frac{1}{2^r})$. If the adversary knows the two secret information, it is easy to decrypt an encrypted message.

We also assumed that the adversary knows the value of $t$. If the adversary does not know the value of $t$, then it can also provide a strong deterrence against the attacks. Let us assume that the attacker can match the secret information due to collision, but the attacker is unable to produce the correct value of $t$. Let us assume that the value of $t$ ranges from 10 to 137 for the value of $c = 10$ and $\beta = 128$. Thus, the probability to get correct $t$ value is $\frac{1}{128}$ because $c$ and $\beta$ are public. Alternatively, the probability is $\frac{1}{2^{10}}$. Also, the value of $r$ is public, and therefore, the probability to know the total value of $r$ is $\frac{1}{2^{137}}$. Therefore, the total probability of breaking symKrypt using brute-force is given in Equation (7).

$$
Pr(BF) = Pr(SK) \cap Pr(S) \cap Pr(t) \cap Pr(r)
$$

$$
= \frac{1}{2^r} \times \frac{1}{2^\beta} \times \frac{1}{128} \times \frac{1}{2^{137}}
$$

$$
\approx \frac{1}{8^3 \times (\beta - 1)}
$$

Therefore, the probability of not able to break symKrypt is $(1 - \frac{1}{8^3 \times (\beta - 1)})$.

Let us assume that attacker is not interested in attacking the shared secret keys. Let also us assume that the adversary knows the value of $t$. Therefore, there are $t$ private keys used to encrypt. The probability of knowing a private key is $\frac{1}{2^r}$.  

There are \( t \) private keys; thus, the total probability break a single block of communication is \( \frac{1}{2^t} \). Moreover, there are \( \psi \) blocks in a message; thus, the total probability capture entire message is \( \frac{1}{2^t \psi} \), which almost zero. If the adversary does not know the value of \( t \) and \( \psi \), then the total probability of capturing entire message is \( \frac{1}{2^t \psi \times (\beta-1) \times 2^{(\beta-1)}} \). Moreover, the value of \( t \) and \( \psi \) change in each communication of each block. The adversary does not know whether to rotate left or right, and how many times to rotate, as shown in Table 1.

Therefore, it is easier to attack in Diffie-Hellman algorithm rather than symKrypt. Hence, symKrypt assumes that Diffie-Hellman algorithm can provide strong security. Thus, our proposed system is able to provide a strong security measurement against the attacks because there is a few public information of symKrypt as shown in Table 1. Most of the parameters are dynamically generated and kept secret.

### 4.4 Cryptanalysis attacks

A cryptanalysis attack is an attack by analyzing the ciphertext to discover the plaintext. An attacker collects many ciphertexts and performs analysis on the collected database. It performs ciphertext-only, known-plaintext, chosen-plaintext/ciphertext, adaptive chosen-plaintext/ciphertext, related-key, and differential attacks. These types of attacks can be applied in the single-keyed symmetric ciphertext. symKrypt uses \( t \) random key generated by pseudo-random number generator. The private keys change in the encryption or decryption of each block of messages. Therefore, cryptanalysis attack does not apply in symKrypt.

The dictionary attack is an attack by creating a dictionary through collecting several ciphertexts. A dictionary attack is dangerous for password-based attacks by creating a large dictionary. Moreover, there is also a birthday attack based on collision probability; however, this kind of attack does not apply to symKrypt due to encryption using several private keys. Also, symKrypt provides strong resistance against a preimage attack.

### 4.5 Unknown attacks

Let us assume that an adversary is able to break symKrypt with a probability of \( (\frac{1}{2^t \psi \times (\beta-1)}) \). The adversary may use any techniques to break the security of symKrypt, for instance, fault attack [14]. In this case, the adversary can break a particular block of message with the probability of \( (\frac{1}{2^t \psi \times (\beta-1)}) \). But there are several blocks of the messages still secured even if a block of message is compromised. It provides a strong deterrence against any possible attacks.

### 5 Experimental Results

We have conducted a series of rigorous tests to verify the correctness of our proposed system. This experimentation is two-fold- first, we experiment the encryption and decryption, and secondly, we test the pseudo-random number generator on NIST SP 800-22 statistically test suite. Our experimental environment is as follows- a) CPU is configured with Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz, b) RAM size is 8GB, HDD size is 1TB, c) operating system is Ubuntu 18.04.5 LTS, and d) programming language is GCC version 7.5.0.

5.1 Cryptography testing

Figure 1 demonstrates the time taken to encrypt and decrypt by symKrypt for various round settings. The round \( t \) is set for single message encryption with 1M~5M private keys. Algorithm 1 takes approximately equal times as Algorithm 2. Algorithm 1 and Algorithm 2 can perform 476399.83 and 479518.57 rounds per second, respectively. It implies that symKrypt can perform XOR operation between the original message and the 476399.83 and 479518.57 private keys per second. Therefore, it is quite fast to encrypt or decrypt a message using symKrypt.

Figure 2 depicts the total time taken to both encrypt and decrypt by symKrypt in \( t \) rounds (the \( t \) is the total rounds of encryption).

Figure 3 shows the time taken to encrypt and decrypt 1M~5M blocks of messages at \( t = 10 \). Here, we use ten private keys to encrypt or decrypt. symKrypt takes 20.92 and 20.99 seconds to encrypt and decrypt 1M blocks, respectively. Similarly, symKrypt takes 104.92 and 104.97 seconds to encrypt and decrypt 5M blocks, respectively.

Figure 4 demonstrates the total number of blocks per second for various \( t \) value settings. The \( t \) value ranges from...
for encryption or decryption. At \( t = 10 \) to 50 which directly translates it uses 10 to 50 private keys of \( t \) as the total number of blocks per second in various values.

Fig. 3: Time taken to encrypt or decrypt several millions of blocks in seconds at the settings of \( t = 10 \). Lower is better.

![Graph showing time taken in seconds vs number of blocks at t = 10](image)

Fig. 4: Comparison between symEnc and symDec for time as the total number of blocks per second in various values of \( t \). Higher is better.

![Graph showing number of blocks per second vs the t rounds of encryption](image)

10 to 50 which directly translates it uses 10 to 50 private keys for encryption or decryption. At \( t = 10 \), symKrypt can perform \( 47639.98 \) and \( 47551.85 \) blocks per second, respectively. Similarly, symKrypt can perform \( 9527.99 \) and \( 9510.37 \) blocks per second at \( t = 50 \), respectively.

### 5.2 Randomness testing

Algorithm 3 is experimented to test its randomness in NIST SP 800-22 statistical test suite [24], [25]. This experimental evaluation shows the randomness of the generated private keys. Table 2 demonstrates the P-value and pass rate of randomness testing in NIST SP 800-22 statistical test suite. NIST SP 800-22 provides approximation entropy, frequency, block frequency, cumulative sums, runs, longest runs, rank, FFT, non-overlapping template, overlapping template, random excursions, random excursions variant, serial, linear, and universal statistical testing of a given input. We have generated 10M random bits and input them into the test suite. The 32 bits, 64 bits, and 128 bits stream are tested in the default configuration of the NIST SP 800-22 test suite. Table 2 proves that the generated private keys are highly unpredictable and random. Therefore, it is difficult to guess the private keys by the adversaries.

The P-value (\( \geq 0.01 \)) is important in deciding the randomness and the pass rate. Table 2 shows the P-values and these P-values are greater than minimum P-value (0.01). The maximum P-value of 32 bits, 64 bits, and 128 bits stream are 0.991468, 0.976060, and 0.941144, respectively. The minimum P-value of 32 bits, 64 bits, and 128 bits stream is 0.100508, 0.016990, and 0.028181, respectively. The maximum pass rate is 1 for all. The minimum pass rate of 32 bits, 64 bits, and 128 bits stream is 0.9375, 0.96875, and 0.96875, respectively. Thus, Algorithm 3 proves its capability of generating a truly random number that can be used to generate the private keys for symKrypt.

### 6 Conclusion

This article demonstrates our proposed symmetric-key cryptography algorithm, symKrypt, which is the first of its kind. Our proposed algorithm is simple and straightforward yet powerful. It can be used on any platform to secure symmetric communication. symKrypt depends on multiple private keys, which are generated dynamically and kept secret. Our experimental results show that the proposed pseudo-random number generator algorithm to generate private keys are unpredictable and secure. It is tested in NIST SP 800-22 statistical test suite. Moreover, the symKrypt uses two shared secret keys, namely, shared secret key and shared secret seed value computed by the Diffie-Hellman algorithm. These two secret keys are used to generate private keys but are not used to encrypt the messages. In addition, symKrypt changes its private key for the encryption or decryption process in each iteration of a block of message. Also, it changes the private keys in each block of a message. symKrypt is the first variant to use multiple private keys without using extra communication for the private keys. The sender and receiver do not exchange the private keys but compute the private keys independently without communication.

Furthermore, the total private keys \( t \) and the rotation information \( r \) are kept secret. These are calculated dynamically. Besides, the left or right rotation is kept secret, which is also computed dynamically. Thus, symKrypt has two public information: the bit size information \( \beta \) and the maximum/minimum private key ranges.

We have demonstrated the value of \( t \geq 10 \) ranging from 1M to 5M experimentally and validated its correctness. It shows the correctness of our proposed algorithm that it works on a very large set of private keys. The performance of encryption and decryption is quite fast, as shown in the experimental section. Moreover, the bit size can be defined by the user, and it can be any size as per the requirement of the user’s application. However, the condition is \( \beta \geq m \) where \( m \) is the block of a message. There is no restriction on bit size, unlike conventional symmetric-key cryptography. We also analyzed the time complexity, which is \( O(m) \) where \( m \) is the total block of messages. Moreover, we discuss the correctness of our proposed algorithm theoretically and experimentally.

symKrypt provides strong resistance against any attacks except DDoS and MITM attacks. We illustrate the resistance of the symKrypt for any attacks. The probability of attacking symKrypt is too small, and it is almost negligible. Our proposed algorithm can defend any possible symmetric-key cryptography attack due to various reasons, particularly...
<table>
<thead>
<tr>
<th>Test name</th>
<th>32 bits</th>
<th>64 bits</th>
<th>128 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P-value</td>
<td>Pass rate</td>
<td>P-value</td>
</tr>
<tr>
<td>Approximate Entropy</td>
<td>0.299251</td>
<td>32/32</td>
<td>0.350485</td>
</tr>
<tr>
<td>Frequency</td>
<td>0.299251</td>
<td>32/32</td>
<td>0.253551</td>
</tr>
<tr>
<td>Block Frequency</td>
<td>0.299251</td>
<td>32/32</td>
<td>0.534146</td>
</tr>
<tr>
<td>Cumulative sums</td>
<td>0.299251</td>
<td>32/32</td>
<td>0.509934</td>
</tr>
<tr>
<td>Runs</td>
<td>0.739918</td>
<td>31/32</td>
<td>0.637119</td>
</tr>
<tr>
<td>Longest runs</td>
<td>0.190808</td>
<td>32/32</td>
<td>0.568055</td>
</tr>
<tr>
<td>Rank</td>
<td>0.531146</td>
<td>31/32</td>
<td>0.949602</td>
</tr>
<tr>
<td>TPR</td>
<td>0.407091</td>
<td>30/32</td>
<td>0.169901</td>
</tr>
<tr>
<td>Non-overlapping template</td>
<td>0.991468</td>
<td>32/32</td>
<td>0.976060</td>
</tr>
<tr>
<td>Overlapping template</td>
<td>0.213409</td>
<td>32/32</td>
<td>0.407091</td>
</tr>
<tr>
<td>Random Excursions</td>
<td>0.637119</td>
<td>13/13</td>
<td>0.122325</td>
</tr>
<tr>
<td>Random Excursions Variant</td>
<td>0.637119</td>
<td>13/13</td>
<td>0.213309</td>
</tr>
<tr>
<td>Serial</td>
<td>0.602458</td>
<td>32/32</td>
<td>0.862344</td>
</tr>
<tr>
<td>Linear complexity</td>
<td>0.299251</td>
<td>31/32</td>
<td>0.862344</td>
</tr>
<tr>
<td>Universal</td>
<td>0.350485</td>
<td>32/32</td>
<td>0.772690</td>
</tr>
</tbody>
</table>

a) minimal public key, b) secret information is dynamic in nature, and c) multiple private keys.

**References**


