Int. j. inf. tecnol. (October 2023) 15(7):3555–3562
https://doi.org/10.1007/s41870-023-01407-3

ORIGINAL RESEARCH

Lightweight symmetric key encryption for text using XOR operation and permutation matrix

Maroti Deshmukh[1] · **Arjun Singh Rawat**[1]

Received: 18 January 2023 / Accepted: 31 July 2023 / Published online: 14 August 2023
© The Author(s), under exclusive licence to Bharati Vidyapeeth's Institute of Computer Applications and Management 2023

Abstract Traditionally, symmetric key encryption has been computationally intensive, and the balance between security and computation costs has necessitated the use of systems with simpler computations and variable-sized keys. This paper introduces a lightweight technique for symmetric key encryption of text, which utilizes less complex operations such as XOR and permutation matrices. The proposed method encrypts and decrypts text based on the ASCII values of plain text characters, using two symmetric keys to provide security. The use of lightweight and computationally efficient operations like XOR and permutation matrices ensures efficient encryption. One of the advantages of this approach is that it allows for the use of any key size without increasing computational complexity, while also enhancing security. Our statistical findings demonstrate that while increasing input size increases computation complexity, increasing key size does not affect it. Thus, any key size can be used for greater security. In summary, our proposed technique provides a simple, lightweight, and efficient method for data encryption and decryption with enhanced security using variable-sized keys.

Keywords Encryption · Decryption · ASCII · Symmetric encryption · Plain text · Cipher text · XOR matrix · Permutation matrix · Variable-sized key

✉ Maroti Deshmukh
marotideshmukh@nituk.ac.in

Arjun Singh Rawat
arjunsinghrawat005@gmail.com

1 Department of Computer Science and Engineering, National Institute of Technology, Uttarakhand, Srinagar, India

1 Introduction

Cryptography produces ciphers to secure communication between a sender and recipient, with objectives such as privacy, integrity, and access control. Encryption scrambles messages and decryption transforms them back to plain text. Cryptography algorithms are used to encrypt and decrypt data, with the primary objective of making it difficult to decode without the key. Testing each key combination is the most effective technique, but time-consuming. Cryptography algorithms are usually classified into two groups: symmetric key encryption and asymmetric key encryption.

1. **Symmetric key encryption** [1] uses a single key for both the encryption and decryption processes, as shown in Fig. 1. The key is first sent via a secure channel to the sender and receiver, and the strength of the encryption depends on the length of the key (in bits). Various techniques are used and researched in different fields for key distribution between the sender and receiver. Examples of symmetric key encryption algorithms include RC2, DES, 3DES, RC5, Blowfish, AES [2–5], and S-DES.
2. **Asymmetric key encryption** resolves the issue of key distribution in symmetric algorithms by using different keys for encryption and decryption, as shown in Fig. 2. This method employs two types of keys: private keys and public keys. The original data or plain text is encrypted using the public key to produce a cipher text. When the receiver receives this cipher text, they use their own private key to decrypt it. A private key is also referred to as a secret key since only the intended recipient or an authorized person is aware of it. However, public keys can be kept in open databases where anyone can access them. Examples of asymmetric key encryption methods

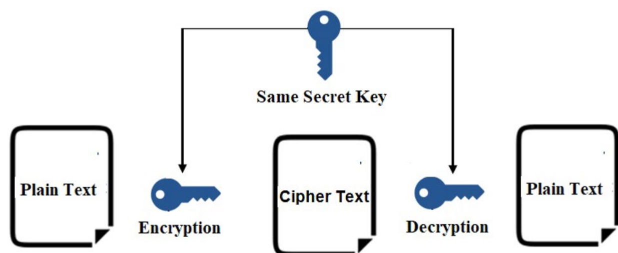

Fig. 1 Symmetric key encryption

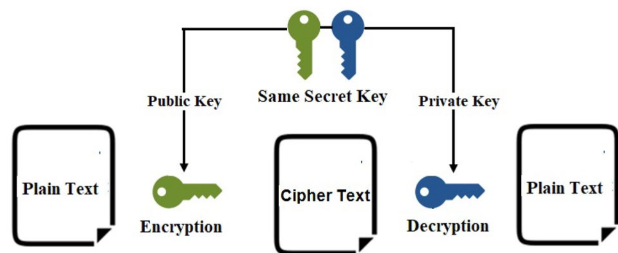

Fig. 2 Asymmetric key encryption

include RSA [6, 7], Digital Signatures [8], and others [9–17].

It is assumed that the encryption and decryption algorithm of a certain cryptography system is known to everyone. The level of security of the cipher text is determined by the secrecy of the key used for encryption and decryption.

The paper is structured as follows: the background work of symmetric and asymmetric key encryption schemes is discussed in Sect. 2, the proposed method is presented in Sect. 3, the performance of the proposed scheme is evaluated in Sect. 4, the security analysis of the proposed scheme is presented in Sect. 5, and the paper is concluded with potential future work in Sect. 6.

2 Background work

Symmetric key encryption has been an important technique for ensuring data security for a long time. Traditional symmetric encryption algorithms such as AES, DES, and 3DES have been widely used, performance of these algorithms discussed by Patel et al. [18]. However, these techniques are computationally intensive and have certain limitations in terms of the key size and complexity. Researchers have developed various lightweight and efficient symmetric encryption schemes to address these limitations. One of the common approaches to achieving lightweight encryption is to use simple operations such as XOR and permutation matrices, which are computationally efficient. For example, the Lightweight Block Cipher family (LBlock, LEA, and Piccolo) are lightweight ciphers that use simple operations such as bitwise XOR and permutation. These ciphers have shown good performance in terms of security and efficiency, making them suitable for resource-constrained devices. Another approach to lightweight symmetric encryption is to use variable-sized keys. The RC5 algorithm is an example of a symmetric encryption scheme that allows for variable key sizes. This technique enhances the security of the cipher by allowing for the use of larger key sizes while keeping computational complexity low. Table 1 shows the variety of symmetric key encryption with their advantages, disadvantages, or limitations.

The proposed technique of symmetric key encryption using lightweight operations such as XOR and permutation matrices, and the use of variable-sized keys is a promising approach for achieving efficient and secure data encryption. The use of ASCII values for encryption and decryption is an added advantage, making it suitable for resource-constrained devices. Various other techniques such as LBlock, LEA, and RC5 have also shown good performance in terms of security and efficiency, making them suitable for use in different applications.

3 Proposed model

The proposed approach begins by transforming characters into their ASCII values before encrypting the plaintext into ciphertext through a series of steps. It utilizes two keys, k_1 and k_2, as symmetric keys for both encryption and decryption. Although there is no limit on key length, a larger value is preferred for improved security. This algorithm covers the entire ASCII character set. The encryption and decryption methods of the proposed algorithm are discussed in detail in the subsequent steps of the algorithm. The detailed steps for plain text encryption are discussed in Algorithm 1 in the following sections. In Algorithm 1(i), the ASCII characters of a plain text encoded representation are shown. First, the plain text characters $PT_1, PT_2, \dots, PT_l$ are converted into ASCII decimal values. Next, the encoded numbers $N_1, N_2, \dots, N_l$ are generated by adding the value 1000 to these converted

Table 1 Literature survey of symmetric key encryption schemes

Year	Scheme	Advantages	Disadvantages/limitations
1977	DES [19]	Strong security Widely adopted Efficient implementation	Key size too small Vulnerable to brute force attacks Outdated algorithm
1999	AES [20]	Strong security Efficient implementation Large key sizes Used by US government	Can be vulnerable to side channel attacks Possible weak key attacks Vulnerable to brute force attacks with small key sizes Limited block sizes
2005	Blowfish [21]	Strong security Simple implementation Fast encryption and decryption Large key sizes	Vulnerable to side channel attacks No longer widely adopted Limited block sizes
1998	Twofish [22]	Strong security Large key sizes High flexibility	Slow in software implementation Limited block sizes
1998	Serpent [23]	Strong security High flexibility No known attacks	Slow in software implementation Limited block sizes Not widely adopted
2011	LBlock [24]	High performance in hardware implementation	Not widely adopted in software implementation
2014	LEA [25]	High security with good performance	Limited block size
2014	Piccolo [26]	High performance with low resource utilization	Limited key size
2005	RC5 [27]	Variable block and key size Efficient implementation Strong security Suitable for low-resource environments	No longer widely adopted Limited support for hardware acceleration Vulnerable to related key attacks

ASCII decimal values. To create the combined number CN, all the encoded numbers undergo a string concatenation operation. In Algorithm 1(ii), the combined number CN is used as a row matrix along with the encryption key K_1 to generate the encoded ASCII matrix α. Here, k_1 defines the number of columns in each row. The number of rows in the α matrix is equal to $len(CN) \mod K_1$, where $len(CN)$ represents the length of CN. Any unfilled columns are filled with bogus values. In Algorithm 1(iii), the β matrix is generated with the help of the private key pair K_1, K_2. In Algorithm 1(iv), the XOR operation is performed between the α and β matrices, and the resulting matrix is added with the value 1000 to generate the γ matrix. In Algorithm 1(v), row-wise permutation is performed on the γ matrix using the θ matrix, which generates the γ_r matrix. The θ matrix contains the row permutation sequences for the row-wise permutation operation on the γ matrix. In Algorithm 1(vi), column-wise permutation is performed on the γ_r matrix using the θ' matrix, which generates the γ_c matrix. The θ' matrix contains the column permutation sequences for the column-wise permutation operation on the γ_r matrix. In Algorithm 1(vii), the ciphertext is generated by first subtracting the value 1000 from each element of the γ_c matrix. The resulting values are then converted into equivalent ASCII characters, which are assigned to ciphers $C_1, C_2, \dots, C_{p-1}, C_p$, where p represents the number of rows in the γ_c matrix and $q = K_1$ represents the number of columns. Finally, all the ciphers are concatenated to generate the ciphertext CT.

Algorithm 1: Encryption Process:

(i) ASCII Characters Encoded Representation::
Input: Plain Text PT
Output: Combined Number CN
$N_1 \leftarrow ASCIIVal(PT_1) + 1000$
$N_2 \leftarrow ASCIIVal(PT_2) + 1000$
...
$N_{l-1} \leftarrow ASCIIVal(PT_{l-1}) + 1000$
$N_l \leftarrow ASCIIVal(PT_l) + 1000$
$CN \leftarrow Concat(N_1, N_2, ..., N_{l-1}, N_l)$

(ii) Encoded ASCII Matrix Generation::
Input: Combined Number CN, Encryption Key pair (K_1, K_2)
Output: Encoded ASCII Matrix α

$$\alpha := \begin{bmatrix} CN_1 & CN_2 & CN_3 & --- & CN_{k_1} \\ CN_{k_1}+1 & CN_{k_1}+2 & CN_{k_1}+3 & --- & CN_{2k_1} \\ - & - & - & --- & - \\ CN_{l-1} & CN_l & B_1 & --- & B_{k_1} \end{bmatrix}$$

(iii) Keys Encoded Matrix Generation::
Input: Encryption Key pair (K_1, K_2)
Output: Keys Encoded Matrix β
$\beta(a_{ij}) \leftarrow (((i \times j + i + j)^{K_2}) \mod K_2) \mod 256$

(iv) XOR operation::
Input: Matrics α and β
Output: XOR Matrix γ
$\gamma \leftarrow (\alpha \oplus \beta) + 1000$

(v) Row-Wise Permutation::
Input: XOR Matrix γ, ROW Matrix θ
Output: Row-Wise Permuted XOR Matrix γ_r
$\gamma_r \leftarrow RP(\gamma, \theta)$

(vi) Column-Wise Permutation::
Input: Row-Wise Permuted XOR Matrix γ_r, Column Matrix θ'
Output: Column-Wise Permuted Matrix γ_c
$\gamma_c \leftarrow CP(\gamma_r, \theta')$

(vii) Cipher Text Generation::
Input: Column-Wise Permuted Matrix γ_c
Output: Cipher Text CT
$CT_1 \leftarrow ASCIIChar(\gamma_{c_{1,1}} - 1000)$
$CT_2 \leftarrow ASCIIChar(\gamma_{c_{1,2}} - 1000)$
...
$CT_p \leftarrow ASCIIChar(\gamma_{c_{p,q}} - 1000)$
$CT \leftarrow Concat(CT_1, CT_2, ..., CT_{p,q})$

The detailed steps for decrypting the ciphertext are discussed in Algorithm 2 in the following sections. In Algorithm 2(i), the ASCII characters of the decoded ciphertext representation are shown. Here, the ciphertext characters $CT_1, CT_2, \ldots, CT_l$ are first converted into ASCII decimal values. The resulting numbers $CTN_1, CTN_2, \ldots, CTN_l$ are generated by adding the value 1000 to these converted ASCII decimal values. To create the combined ciphertext number CCN, all the encoded numbers undergo a string concatenation operation. In Algorithm 2(ii), the combined number CCN is used as a row matrix along with the encryption key K_1 to reconstruct the decoded ASCII matrix γ_r. Here, K_1 defines the number of columns in each row. The number of rows are equal to $len(CCN) \mod K_1$, where $len(CCN)$ represents the length of the CCN. In Algorithm 2(iii), column-wise permutation is performed on the γ_r matrix using θ' matrix, which reconstructs the γ_c matrix. The θ' matrix contains the column permutation sequences for the column-wise permutation operation on the γ_r matrix. In Algorithm 2(iv), row-wise permutation is performed on the γ_c matrix using θ matrix, which reconstructs the plaintext matrix. The θ matrix contains the row permutation sequences for the row-wise permutation operation on the γ_c matrix. In Algorithm 2(v), the private key

pair K_1 and K_2 are used to generate the β matrix. In Algorithm 2(vi), the XOR operation is performed between the plaintext matrix and the β matrix, and the resulting matrix is subtracted by the value 1000 to reconstruct the α matrix. In Algorithm 2(vii), the combined number CN, which is a row matrix, is reconstructed by concatenating the elements of the α matrix in a row-wise sequence. After that, for each group of combined numbers, the first four elements are concatenated and subtracted by the value 1000. The resulting values are then converted into ASCII character values and stored in variables $PT_1, PT_2, \dots, PT_{p-1}, PT_p$. Finally, a concatenation operation is performed to reconstruct the plaintext value. Furthermore, the ciphertext is generated by first subtracting the value 1000 from each element of the plaintext matrix γ_c. The resulting values are then converted into equivalent ASCII characters, which are assigned to ciphers $C_1, C_2, \dots, C_{p-1}, C_p$, where p represents the number of rows in the plaintext matrix and $q = K_1$ represents the number of columns. Finally, all the ciphers are concatenated to generate the ciphertext CT.

Algorithm 2: Decryption Process:

(i) ASCII value Decoding Representation::
Input: Cipther Text CT
Output: Combined Cipther Text Number CCN
$CTN_1 \leftarrow ASCIIVal(CT_1) + 1000$
$CTN_2 \leftarrow ASCIIVal(CT_2) + 1000$
...
$CTN_{l-1} \leftarrow ASCIIVal(CT_{l-1}) + 1000$
$CTN_l \leftarrow ASCIIVal(CT_l) + 1000$
$CCN \leftarrow Concat(CTN_1, CTN_2, ..., CTN_{l-1}, CTN_l)$

(ii) Decoding ASCII Matrix Generation::
Input: Cipher Text Combined Number CCN, Encryption Key pair (K_1, K_2)
Output: Encoded ASCII Matrix γ_r

$$\gamma_r := \begin{bmatrix} CCN_1 & CCN_2 & CCN_3 & --- & CCN_{k_1} \\ CCN_{k_1}+1 & CCN_{k_1}+2 & CCN_{k_1}+3 & --- & CCN_{2k_1} \\ - & - & - & --- & - \\ CCN_l - k_1 & CCN_l - k_1 - 1 & CCN_l - k_1 - 2 & --- & CCN_l \end{bmatrix}$$

(iii) Column-Wise Permutation::
Input: Matrix γ_r, Column Matrix $\theta^{'}$
Output: Permuted Matrix γ_c
$\gamma_c \leftarrow CP(\gamma_r, \theta^{'})$

(iv) Row-Wise Permutation::
Input: XOR Matrix γ_c, ROW Matrix θ
Output: Matrix γ
$\gamma \leftarrow RP(\gamma_c, \theta)$

(v) Keys Encoded Matrix Generation::
Input: Encryption Key pair (K_1, K_2)
Output: Keys Encoded Matrix β
$\beta(a_{ij}) \leftarrow (((i \times j + i + j)^{K_2}) \mod K_2) \mod 256$

(vi) XOR operation::
Input: Matrics γ and β
Output: Matrix α
$\alpha \leftarrow (\gamma \oplus \beta) - 1000$

(vii) Plain Text Reconsturction::
Input: Matrix α
Output: Cipher Text PT
$CN \leftarrow ConCat(\alpha(ij))$
$PT_1 \leftarrow ASCIIChar(ConCat(CN_1, CN_2, CN_3, CN_4) - 1000)$
$PT_2 \leftarrow ASCIIChar(ConCat(CN_5, CN_6, CN_7, CN_8) - 1000)$
...
$PT_p \leftarrow ASCIIChar(ConCat(CN_{l-3}, CN_{l-2}, CN_{l-1}, CN_l) - 1000)$
$PT \leftarrow Concat(PT_1, PT_2, ..., PT_{p-1}, PT_p)$

Table 2 Computational complexity based on sizes of the text file

Size of the text file (KB)	Execution time (s)
4.91	2.2
17.1	10.7
34.5	47.9
63.7	158.7
254	1672.7

Table 3 Computational complexity based on key (k_1) length

k_1	Execution time (s)
3	47.8
7	40.8
19	47.8
37	46.4
71	47.9

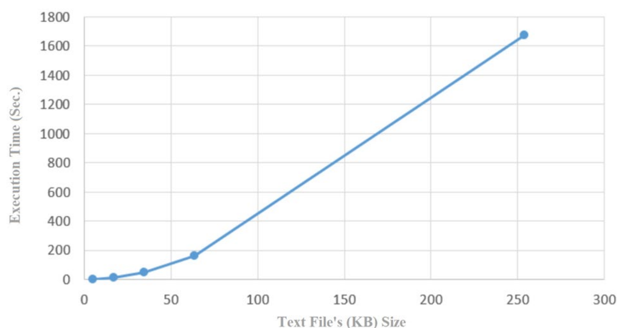

Fig. 3 Computational complexity based on sizes of the text file using line graph

Table 4 Computational complexity based on key (k_2) length

k_2	Execution time (s)
120	46.7
250	44.4
500	45.9
750	44.8
1000	45.7

4 Performance analysis

In performance analysis, we observed the computation complexity varying with different parameters and system specifications: The processor is an AMD A8-7410 APU (2.20 GHz), with 8.00 GB (6.91 GB usable) of DDR3 RAM, and a 64-bit Operating System (Windows 10 Pro), x64 based processor. The evaluation of the execution time includes the total time taken for encryption and decryption, as well as the time elapsed for file read and write operations in Java.

Table 2 shows the performance analysis based on sizes of the text file, by keeping the keys k_1 and k_2 fixed, total execution time (including both encryption and decryption) based on given parameter in below.

- A plain text file is encrypted and then decrypted.
- $k_1 = 120$ and $k_2 = 71$.

Figure 3 demonstrates that the line graph depicts the execution time of a proposed scheme based on the size of the text file (KB). Increasing the text file's (KB) size results in a increased execution time.

Table 3 shows the performance analysis based on key size, by Keeping the key k_1 fixed and applying the algorithm

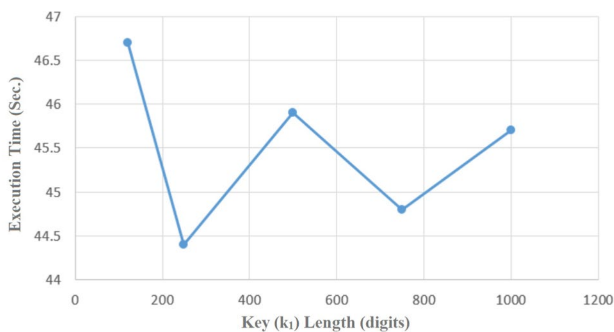

Fig. 4 Computational complexity based on key (k_1) length using line graph

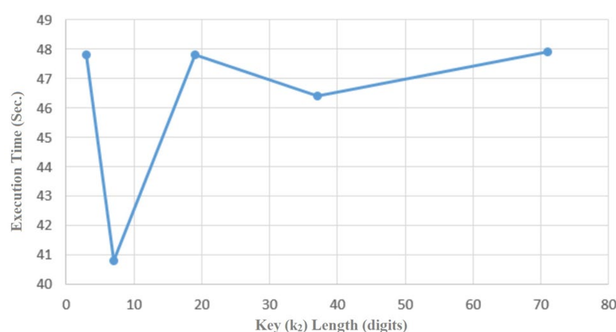

Fig. 5 Computational complexity based on key (k_1) length using line graph

Table 5 Comparison of symmetric encryption schemes

Scheme	Key size	Block size	Advantages	Disadvantages/limitations
DES [19]	56 bits	64 bits	1. Strong security	1. Key size too small
AES [20]	1. 128 bits 2. 192 bits 3. 256 bits	128 bits	1. Strong security 2. Large key sizes 3. Efficient implementation	1. Weak key attacks 2. Limited block sizes
Blowfish [21]	32–448 bits	64 bits	1. Strong security 2. Simple implementation 3. Large key sizes	1. Vulnerable to side channel attacks 2. Limited block sizes
Twofish [22]	1. 128 bits 2. 192 bits 3. 256 bits	128 bits	1. Strong security 2. High flexibility 3. No known attacks	1. Slow in software implementation 2. Limited block sizes 3. Not widely adopted
LBlock [24]	80 bits	64 bits	1. Performance in hardware implementation	1. Widely adopted in software implementation
LEA [25]	1. 128 bits 2. 192 bits 3. 256 bits	64 bits	1. High security with good performance	1. Limited block size
Piccolo [26]	1. 80 bits 2. 128 bits 3. 192 bits	64 bits	1. High performance 2. Low resource utilization	1. Limited key size
RC5 [27]	Up to 2040 bits	Up to 2040 bits	1. Variable block and key size 2. Efficient implementation 3. Strong security 4. Suitable for low-resource environments	1. No longer widely adopted 2. Limited support for hardware acceleration 3. Vulnerable to related key attacks
Proposed	Variable	Variable	1. Strong security 1. Large key sizes 1. Utilizes entire ASCII character set	1. Brute-force attacks for small key size

on the same file, variation of execution time with key k_1 based on given parameter in below.

- Size of the plain text file is 34.5 KB.
- $k_1 = 120$.

Figure 4 shows how a proposed scheme's execution time is represented by a line graph based on the length of the key (k_1), where increasing the length of the key has no effect whatsoever on the execution time.

Table 4 shows the performance analysis based on key size, by Keeping the key k_2 fixed and applying the algorithm on the same file, variation of execution time with key k_2 based on given parameter in below.

- Size of the plain text file is 34.5 KB.
- $k_2 = 23$.

Figure 5 shows how a proposed scheme's execution time is represented by a line graph based on the length of the key (k_2), where increasing the length of the key has no effect whatsoever on the execution time.

Table 5 displays execution times for different key lengths of a particular encryption scheme, and the figure shows that the proposed scheme's execution time is independent of key length. This is a significant advantage compared to other encryption schemes, which often experience longer execution times with longer key lengths. Moreover, the comparison table demonstrates limitations in block size, vulnerability to side-channel attacks, and limited adoption of some encryption schemes. As a result, the proposed scheme may offer better performance and security than these schemes.

5 Security analysis

The proposed algorithm provides high level of security as an attacker would need to know both keys (k_1 and k_2) to decrypt the ciphertext and test all possible combinations. The length of keys is not limited, making it computationally infeasible for an attacker to attempt 10^{170} times even for large key values like 10^{50} and 10^{120}. Using k_2 as an exponent in both encryption and decryption procedures may increase the computing cost, but even for smaller values of keys, larger files will take significantly more time to execute. For optimal security, it is recommended to use keys with large values.

6 Conclusion and future scope

The proposed approach is a lightweight symmetric key encryption technique that uses permutation matrices and

XOR operations for encryption and decryption with any key size, without increasing the computation complexity. The performance analysis demonstrated that larger key sizes provide higher security and that the size of the keys is completely independent of the execution time. However, the size of the file affects the execution time, meaning that as the text file size increases, so does the execution time.

The future work of the proposed algorithm includes modifying the functions used to generate the XOR matrix and permutation matrix to enhance the algorithm's security. Additionally, optimizations can be implemented to improve the speed of encryption and decryption of larger text files. Overall, the approach provides a secure and lightweight encryption technique for textual data and has potential for further enhancements to improve its performance and security.

Funding This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Availability of data and material (data transparency): Data available on request from the authors.

Code availability (software application or custom code) Code available on request from the authors.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. Bokhari MU, Shallal QM (2016) A review on symmetric key encryption techniques in cryptography. Int J Comput Appl 147(10):43–48
2. Abdullah AM (2017) Advanced encryption standard (AES) algorithm to encrypt and decrypt data. Cryptogr Netw Secur 16:1–11
3. Han S-J, Oh H-S, Park J (1996) The improved data encryption standard (DES) algorithm. In: Proceedings of ISSSTA'95 international symposium on spread spectrum techniques and applications, vol 3. IEEE, Mainz, Germany, pp 1310–1314
4. Knudsen LR et al (1998) On the design and security of RC2. In: International workshop on fast software encryption. Springer, Berlin
5. Noura M et al (2018) S-DES: an efficient and secure DES variant. In: IEEE middle east and north Africa communications conference (MENACOMM), IEEE, Jounieh, Lebanon, pp 1–6
6. Milanov E (2009) The RSA algorithm. RSA Laboratories, pp 1–11. https://sites.math.washington.edu/~morrow/336_09/papers/Yevgeny.pdf
7. Imam R, Anwer F, Nadeem M (2022) An effective and enhanced RSA based public key encryption scheme (XRSA). Int J Inf Technol 14(5):2645–2656
8. Merkle RC (1990) A certified digital signature. In: Conference on the theory and application of cryptology. Springer, New York
9. Yassein MB et al (2017) Comprehensive study of symmetric key and asymmetric key encryption algorithms. In: International conference on engineering and technology (ICET), IEEE, Antalya, Turkey, pp 1–7
10. Hellman ME (2002) An overview of public key cryptography. IEEE Commun Mag 40(5):42–49
11. Rawat AS, Deshmukh M (2019) Efficient extended Diffie–Hellman key exchange protocol. In: 2019 International conference on computing, power and communication technologies (GUCON). IEEE, pp 447–451
12. Rawat A, Deshmukh M (2020) Tree and elliptic curve based efficient and secure group key agreement protocol. J Inf Secur Appl 55:102599
13. Rawat AS, Deshmukh M (2020) Communication efficient Merkle-Tree based authentication scheme for smart grid. In: 2020 IEEE 5th international conference on computing communication and automation (ICCCA). IEEE, pp 693–698
14. Rawat AS, Deshmukh M (2021) Computation and communication efficient Chinese remainder theorem based multi-party key generation using modified RSA. In: Security and privacy, vol 25–32. Springer, Singapore
15. Rawat AS, Deshmukh M (2021) Computation and communication efficient secure group key exchange protocol for low configuration system. Int J Inf Technol 13(3):839–843
16. Sharma P, Purushothama BR (2023) Cryptanalysis of a secure and efficient Diffie–Hellman based key agreement scheme. Int J Inf Technol 15:1–9
17. Bhat R, Sunitha NR, Iyengar SS (2022) A probabilistic public key encryption switching scheme for secure cloud storage. Int J Inf Technol 15(2):1–16
18. Patel K (2019) Performance analysis of AES, DES and Blowfish cryptographic algorithms on small and large data files. Int J Inf Technol 11:813–819
19. Standard, Data Encryption (1977) Federal information processing standards publication 46, vol 23. National Bureau of Standards, US Department of Commerce, pp 1–18
20. Daemen J, Rijmen V (1999) AES proposal: Rijndael
21. Schneier B (2005) Description of a new variable-length key, 64-bit block cipher (Blowfish). In: Fast software encryption: Cambridge security workshop, Cambridge, UK, December 9–11, 1993 proceedings, pp 191–204
22. Schneier B et al (1998) Twofish: a 128-bit block cipher. NIST AES Propos 15(1):23–91
23. Anderson R, Biham E, Knudsen L (1998) Serpent: a proposal for the advanced encryption standard. NIST AES Propos 174:1–23
24. Wu W, Zhang L (2011) LBlock: a lightweight block cipher. In: Applied cryptography and network security: 9th international conference, pp 327–344
25. Hong D et al (2014) LEA: a 128-bit block cipher for fast encryption on common processors. In: Information security applications: 14th international workshop, pp 3–27
26. Vasiliadis G et al (2014) PixelVault: using GPUs for securing cryptographic operations. In: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, pp 1131–1142
27. Rivest RL (2005) The RC5 encryption algorithm. In: Fast software encryption: second international workshop Leuven, pp 86–96

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.