International Journal of Engineering Research and Science & Technology

www.ijerst.org

ISSN : 2319-5991

Vol. 21 No. 3 (1) 2025

ijerst.editor@gmail.com

editor@ijerst.com

International Journal of Engineering Research and Science & Technology

ISSN 2319-5991 www.ijerst.com
Vol. 21, No. 3(1), 2025

Research Paper

AN IMPROVED CHAOTIC IMAGE ENCRYPTION FOR ENHANCED SECURITY AND KEY SPACE OPTIMIZATION

B. Rajesh Reddy[1], S. Charandeep[2], Maila Nithin[2], Patemidi Praveen[2],
[1]Assistant Professor, [2]UG Student, [1,2] Department of Computer Science and Engineering,
[1,2]Kommuri Pratap Reddy Institute of Technology, Ghanpur, Ghatkesar, Telangana, 500088.
Email - rajeshreddy.reddy54@gmail.com

Received: 05-6-2025 Accepted: 06-7-2025 Published: 14-7-2025

ABSTRACT

Embedding secret data within digital images (steganography) is a widely used technique for covert communication. However, traditional LSB methods lack confidentiality and integrity checks, making them vulnerable to extraction and tampering. To address these shortcomings, this work proposes an integrated solution that combines encryption (ECC or ChaCha20), LSB steganography, and compression to securely hide short payloads inside cover images. Public-key cryptography offers strong security but can be computationally intensive, whereas symmetric stream ciphers like ChaCha20 deliver high performance but require key distribution. Conventional LSB steganography directly embeds plaintext bits, exposing messages to steganalysis and eavesdropping. A robust system is therefore needed to ensure that extracted data cannot be read without the key, and that any alteration is detectable. In this project, the secret message is first encrypted using either ECIES over secp256k1 (asymmetric) or ChaCha20 (symmetric), and its SHA-256 hash is computed to ensure integrity. The resulting ciphertext bytes are Base64-encoded and converted into an ASCII bitstream, which is then embedded into the least significant bits of an RGB image. The stego image is subsequently compressed using zlib to reduce file size and obscure potential embedding artifacts. On the receiver side, the compressed file is decompressed, the LSB bits are extracted to reconstruct the Base64 ciphertext, the SHA-256 hash is recalculated to verify integrity, and the payload is decrypted. A Tkinter GUI guides users through uploading images, entering secrets, selecting encryption modes, and viewing performance graphs that compare ECC and ChaCha20. Experimental results show that ChaCha20 encryption is significantly faster than ECC for short messages, while ECC provides asymmetric security without the need for key exchange. By integrating strong encryption, steganography, and compression, this system mitigates the limitations of traditional LSB methods—ensuring confidentiality, integrity, and minimal perceptual distortion—making it well-suited for applications requiring covert, tamper-evident communication.

Keywords: Steganography, ChaCha20, Elliptic Curve Cryptography (ECC), Confidentiality and Integrity, LSB Embedding.

1. INTRODUCTION

Images, as an essential form of multimedia data, encompass a wide range of sensitive information, including personal privacy, business secrets, and medical images. With the continuous development of communication technology and the widespread use of information transmission, ensuring the security and confidentiality of image information has become particularly urgent. Image encryption, as a crucial technology in the field of information security, is essential to protect the safe transmission of such vital information. Due to the characteristics of images, such as massive data volume, strong correlation, high redundancy, and distinctive recognition features, traditional text encryption methods like AES and DES prove to be slow and ineffective in encrypting and decrypting

images. These methods can no longer meet the encryption needs of large-capacity image data. With the progress of network science and the multimedia industries, data transmission has become the main choice for many people where Digital images are stored or transmitted via public channels [1]. Thus, data security is of great importance [2]. Due to vast quantities of data in digital images, higher redundancy, and stronger relationship between neighboring pixels and is not effective, and it has been found that the classical data encryption technique has many technical defects [3]. When compared to other classical encryption algorithms, the experimental outcome shows that image encryption depends on the concept of chaos and has better features [4].

In many aspects, the study of Chaos has made tremendous progress [5]. Chaos meets the requirement of image encryption due to its features of chaos, like inherent randomness, and maximum sensitivity to primary conditions and control parameters in recent times, chaos research has become increasingly popular [6]. Pseudo-randomness, unpredictability, and high sensitivity to primary value are the features of a chaotic system making it well-suited for image encryption methods [7]. Also, the image encryption approach depends on chaos is extensively studied [8]. Generally, this encryption technique involves two different steps: scrambling and diffusion [9]. Shuffling is used to reduce the relationship between pixels, and Diffusion is used to make the pixel value equally distributed [10].

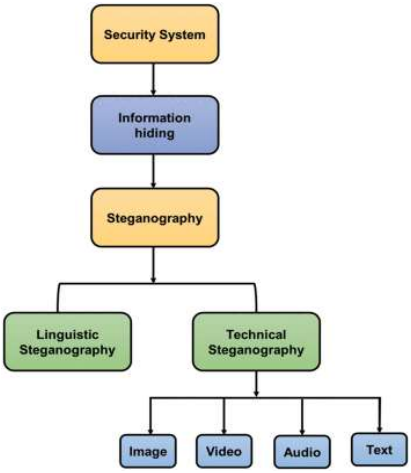

Fig.1: Steganography Classification.

This work enhances traditional LSB steganography by integrating encryption (ECC or ChaCha20), integrity verification (SHA-256), and compression (zlib) to securely embed secret messages in images. The system encrypts the message, hashes it for integrity, embeds it into an image's LSBs after Base64 encoding, and compresses the stego image. A Tkinter GUI allows user interaction, and results show ChaCha20 is faster for short messages, while ECC is better for secure key exchange. This approach ensures confidentiality, integrity, and reduced detectability for covert communication.

2. LITERATURE SURVEY

Alohali et al. [11] introduce a Blockchain Driven Image Encryption using an arithmetic optimization algorithm (AOA) with the Fractional-Order Lorenz System (BDIE-AOFOLS) algorithm. This proposed model employed the Fractional-Order Lorenz System (FOLS) model that incorporates the Fractional Lorenz, Arnold Map, and Tent Map schemes. Additionally, an AOA is performed for the optimum key generation procedure for achieving the optimum value of peak signal-to-noise ratio (PSNR).

Zhu et al. [12] presented a Chaotic Digital Image encrypting system that depends on a maximized Artificial Fish Swarm (AFS) approach and coding of DNA. Firstly, the key is related to the normal image pixels via the MD5 hash function, and the value of the hash produced by the normal images is utilized as

the primary value of the hyperchaotic scheme to enhance the key sensitivity. Then, the AFS approach is implemented to disorganize the pixel position in the block.

The authors in Ref. [13], introduced an image encryption technique that depends on the upgrading procedures of PSO and Hyperchaotic Complex Lü (HCL) systems. Particularly, a mechanism of key generation incorporated with the Secured Hash approach of 256 hash is initially presented for generating the HCL scheme's primary values. Later, the plain images are disorganized by the velocity and position upgrading procedure of the PSO method.

Khaitan et al. [14] suggested a public key crypto scheme that integrates the Chaos Tent Map (CTM) function along with an Improved Salp Swarm Optimization (ISSO) approach for the image's decrypting and encrypting process. This ISSO approach is upgraded by enforcing mutation and crossover for generating a decrypting key. This encrypting system encompasses a circular shit operation and a permutation, which are controlled by control parameters and chaos-based keys. At the time of the encrypting procedure, an eight-bit shift catalogue is utilized with XOR operation to improve the cypher image's randomness.

Maniyath and Thanikaiselvan [15] present an analytical research model for proposing a state-of-the-art framework, in which the Deep Neural Network (DNN) is utilized for the optimization of the plain encryption method's achievement. The vigorousness of the optimizing principles is additionally supplemented with a chaotic map notion for improved safety accomplishments.

Ferdush et al. [16] proposed a technique that utilizes the Genetic Algorithm (GA) for an improved pixel value encryption and PSO to enhance the process of optimization. This technique is segmented into two sectors. Firstly, the plain RGB images are implemented for the primary populace and later the GA is employed for image encryption. Secondly, the PSO model is employed for determining the best-encryption images. Lastly, the best images are put under a re-encryption process still an optimal value is achieved.

Zeng and Wang [17] present a scheme for Hyperchaotic Image Encryption (HIE) which is based on Cellular Automata (CA) and PSO algorithms. At first, to enhance the capacity for resisting the outbreaks of the plaintext, the hyperchaotic scheme's primary conditions are produced by the values of hash functions that are closely associated with the plaintext images yet to undergo encryption. Addi tionally, PSO's fitness is the correlative coefficient among the image's neighboring pixels.

In Ref. [18], a fusion technique is proposed by implementing the Tent Chaotic Mapping (TCM) and DNA Sequence (DNA-S) models. At first, the initial and TCM images are put under the encryption process distinctly by implementing the DNA-S model. Later, the logical XOR operator is enforced on the models.

Ma [19] employed the three-dimensional Chen chaotic system and the Fisher-Yates permutation scheme to encrypt color images. In this approach, the sequence generated by the three-dimensional chaotic system serves as the cipher stream for the Fisher–Yates permutation scheme, disrupting the pixel positions. Subsequently, a new set of initial values is used to generate a new sequence through the chaotic system, ultimately operating with the plaintext pixel to alter its value. Three-dimensional chaotic systems provide higher security compared to their one-dimensional and two-dimensional systems, exhibiting more complex dynamical behavior, that enhances the system's unpredictability against potential attackers. Three-dimensional chaotic systems offer relatively robust protection with somewhat controllable computational complexity, making them suitable for moderately complex image encryption applications. However, although relatively secure, three-dimensional chaotic systems have a reasonably limited keyspace, which may need improvement in highly security-demanding scenarios.

To overcome this deficiency, Zhao [20] proposed a new image encryption method using a four-dimensional chaotic system combined with DNA coding. Experimental analyses demonstrate that the method not only achieves a substantial keyspace but also enhances security performance. Four-dimensional chaotic systems provide a larger keyspace than three-dimensional systems, heightening the difficulty of attacks and improving encryption security. As dimensionality increases, the dynamical behavior becomes more intricate, further strengthening the encryption. Four-dimensional chaotic systems are better suited for scenarios with higher security requirements than three-dimensional systems.

3. PROPOSED METHODOLOGY

This application is a standalone Tkinter-based tool designed to securely hide and retrieve text messages within images. It combines elliptic curve cryptography (ECC) or the ChaCha20 stream cipher with least significant bit (LSB) steganography and zlib compression. The primary goal is to embed encrypted messages within image files while maintaining data confidentiality, ensuring message integrity, and minimizing perceptual distortion in the host image. The system offers two encryption modes: ECC, which uses asymmetric key pairs for secure communication without key exchange, and ChaCha20, a symmetric cipher that provides fast encryption using a 256-bit key and a unique nonce per image. Once a message is encrypted using either encryption method, the resulting ciphertext is Base64-encoded and converted into an ASCII bitstream. This bitstream is then embedded into the least significant bits of the red, green, and blue channels of an image, processed in raster order to ensure minimal visual impact. After the embedding process, the modified image is saved and compressed using the zlib library. This compression step not only reduces the file size for storage or transmission but also obscures potential embedding artifacts.

The ECC encryption process, based on ECIES, involves generating a shared secret using the recipient's public key and an ephemeral key. This shared secret is used to derive symmetric keys, which encrypt the plaintext. The resulting package includes the ephemeral public key, the ciphertext, and a message authentication code (MAC). Decryption involves deriving the same shared secret with the private key, verifying the MAC, and decrypting the ciphertext.

ChaCha20, a high-speed stream cipher, generates a pseudorandom keystream from a 256-bit key and a 96-bit nonce. Encryption and decryption are symmetric, using XOR operations between the keystream and plaintext or ciphertext. The project generates a new nonce for each encryption session and stores it alongside the encrypted image for future decryption.

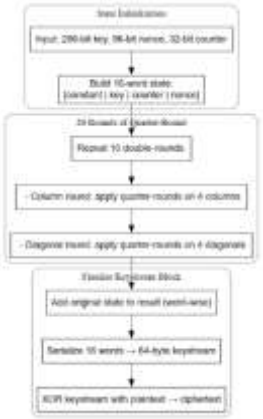

Fig.2: Internal operational flow of ChaCha20.

LSB steganography works by embedding each bit of the encrypted, Base64-encoded message into the LSBs of the image's RGB channels. The embedding proceeds sequentially in raster scan order. During extraction, these bits are reassembled into characters until a special sentinel marker ("$t3g0") is found, indicating the end of the hidden message.

International Journal of Engineering Research and Science & Technology

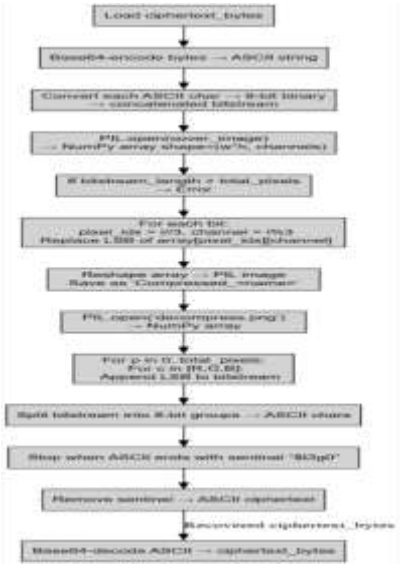

Fig.3: LSB steganography internal operational flow.

To reduce file size and further obscure patterns, zlib compression is applied after embedding. The raw bytes of the stego image are compressed, and the output overwrites the original image file. On the receiver's end, the compressed file is decompressed to restore the modified image before LSB extraction.

To ensure data integrity, the application uses SHA-256 to hash the ciphertext immediately after encryption. This hash is displayed in the interface. During message recovery, SHA-256 is computed again on the extracted ciphertext, and both hashes are compared. A match confirms that the data has not been altered or corrupted during transmission or storage.

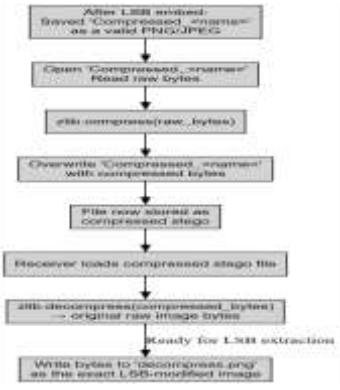

Fig.4: Compression and decompression using zlib.

In summary, this project integrates encryption, steganography, compression, and integrity verification into a cohesive application. It allows users to securely embed and extract encrypted messages within images while comparing the performance of ECC and ChaCha20. The system offers a practical solution for tamper-evident, covert communication through images.

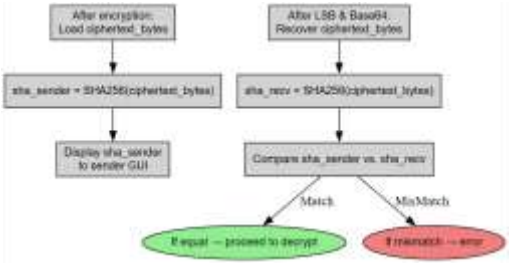

Fig.5: SHA-256 integrity verification operational flow.

4. RESULTS AND DISCUSSION

The ECC encryption process began by either loading or generating the ECC public key (pri.key), which was then used to encrypt the message "Hi how are you. We will meet at 7pm" using the ECIES algorithm. Following encryption, the system computed the SHA-256 hash of the ciphertext and inserted the resulting digest into the "Generated SHA" field for integrity verification. The ciphertext, after being Base64-encoded into an ASCII bitstream, was embedded into the least significant bits (LSBs) of the selected cover image. This modified image was saved as Compressed_cover.png inside the ReceivedCompressImages/ folder and subsequently compressed using zlib to reduce its size and obscure embedding patterns.

The logging area of the application's GUI displayed relevant information such as the Base64-encoded ECC ciphertext string, confirmation that the compressed image had been saved, and the time taken for ECC encryption (e.g., "ECC Encryption Time: 0.123456"). The appearance of the SHA-256 digest in the "Generated SHA" field confirmed the encryption's integrity, signaling that the stego file was ready for transmission to a receiving party.

Upon reception, the user initiated the extraction process by clicking the "Receiver Upload & Decode Message" button, which opened a file dialog defaulting to the ReceivedCompressImages/ directory. After

selecting the previously saved Compressed_cover.png file, the application proceeded through a series of automated steps. First, the zlib-compressed data was decompressed, producing the decompress.png file—an exact replica of the original LSB-modified image. Next, the system performed LSB extraction by scanning the least significant bits of every pixel's red, green, and blue channels in raster order. These bits were reassembled into ASCII characters until the sentinel string "$t3g0" was detected, indicating the end of the embedded message.

The extracted ASCII string was then Base64-decoded to reconstruct the original ECC ciphertext bytes. A new SHA-256 hash was computed and displayed in the text area as "Receiver Generated SHA code = <hex digest>", which the user compared against the original "Generated SHA" to confirm that no tampering had occurred. Finally, the system decrypted the ciphertext using the ECC private key (pvt.key), successfully recovering and displaying the original plaintext: "Hi how are you. We will meet at 7pm." The application also displayed decompress.png in a new window, visually confirming that the stego image was indistinguishable from the original cover image. Each step of the decoding workflow was logged in the text area, providing clear feedback and confirmation that the extraction and decryption process had succeeded without error.

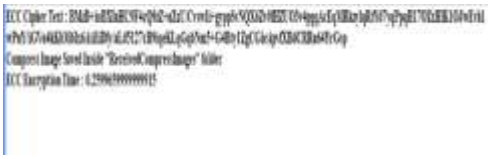

Fig.6: uploading sender side image, and secrete message embedding using compress and send operation (i.e., ECC computation).

The ChaCha20 encryption process began by loading an existing 32-byte symmetric key from the key file or generating a new one if none was found. Using this key, the system created a new ChaCha20.new(key=K) cipher object, which automatically generated a secure, random nonce to ensure encryption uniqueness. The plaintext message "Hi how are you. We will meet at 7pm" was then encrypted using this cipher, producing the chacha_ciphertext.

To ensure the integrity of the ciphertext, the system computed its SHA-256 hash and placed the resulting hexadecimal digest into the "Generated SHA" field within the GUI. This digest serves as a fingerprint of the ciphertext, allowing the receiver to verify that the message has not been altered during transmission or storage. To support future decryption, the application also saved the randomly generated nonce by pickling it and storing it under the nonce/cover.png file path. This ensures that the receiver can retrieve the correct nonce associated with the encrypted image for accurate decryption.

Next, the encrypted data was Base64-encoded to convert it into an ASCII string suitable for steganographic embedding. This encoded string was then converted into a binary bitstream and embedded into the least significant bits (LSBs) of the RGB channels of the selected cover image. The resulting stego image was saved as Compressed_cover.png within the ExtensionCompressImages/ folder and then compressed using zlib to minimize file size and reduce the visibility of embedding artifacts.

The application logged each of these actions in the text area of the GUI. Log entries included the Base64 representation of the ChaCha20 ciphertext, confirmation of the saved compressed image, and the encryption duration (e.g., "CHACHA20 Encryption Time: 0.045678"). The presence of the computed SHA digest in the "Generated SHA" field confirmed the integrity of the ciphertext, indicating that the encrypted, compressed stego image was ready for secure transfer to the intended ChaCha20-based receiver.

Fig.7: GUI application after performing uploading sender side image, and secrete

message embedding using compress and send operation (i.e., CHACHA20 computation).

Fig.8: after performing receiver upload and decode operation using CHACHA20.

Fig. 8 demonstrates the ChaCha20-based receiver workflow. Upon clicking "Receiver CHACHA Decode Message", the GUI opened a dialog pointing to ExtensionCompressImages/ where the user selected Compressed_cover.png. The system then:

The ChaCha20 decryption process began when the receiver selected the compressed stego file. The system first applied zlib. Decompress (...) to this file, producing decompress.png, which is an exact replica of the LSB-modified stego image. This image was then loaded as a NumPy array, allowing the application to sequentially read the least significant bit of each red, green, and blue channel across all pixels. These bits were reassembled into an ASCII bitstream until the sentinel string "$t3g0" was encountered, signaling the end of the embedded message.

The extracted ASCII string was then Base64-decoded to reconstruct the original chacha_ciphertext. Immediately after, the system computed the SHA-256 hash of this ciphertext and displayed the result in the text area as: Extension Receiver Generated SHA code = <hex digest>

The user was then prompted to verify that this hash matched the original "Generated SHA" shown earlier, confirming the integrity of the message and that no tampering had occurred.

With integrity verified, the system proceeded to decrypt the ciphertext. It loaded the 32-byte symmetric key from the key file and retrieved the associated nonce by unpickling it from nonce/cover.png. Using these, it instantiated a ChaCha20 cipher object via Cha Cha 20. New (key=K, nonce=nonce) and called the. Decrypt (...) method on the ciphertext. The result was the original plaintext message: "Hi how are you. We will meet at 7pm". This was appended to the GUI's text area with the message:

ChaCha20 Extracted Hidden Secured Message = Hi how are you. We will meet at 7pm

Finally, the Tkinter interface automatically displayed the decompress.png image, allowing the user to visually confirm that the recovered stego image was perceptually identical to the original. All key steps—decompression, LSB extraction, SHA verification, decryption, and display—were clearly logged in the text area, confirming that the ChaCha20 decoding and message recovery were successfully completed.

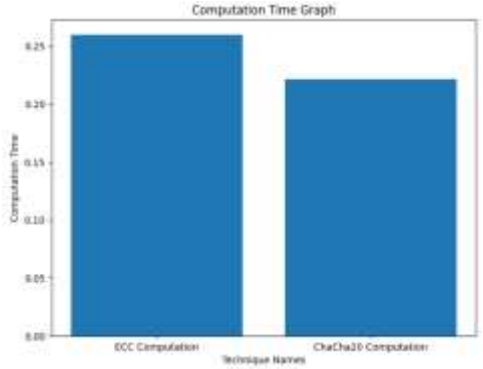

Fig. 9: Computational time performance using ECC, and CHACHA20 algorithms.

Fig. 9 presents a bar chart generated by Matplotlib comparing encryption times for ECC vs. ChaCha20 on identical secret messages and cover images. The x-axis lists two labels— "ECC Computation" and "ChaCha20 Computation"—while the y-axis represents elapsed time in seconds. In the sample plot, the ECC bar is taller (e.g., around 0.26 s) than the ChaCha20 bar (e.g., around 0.22 s), illustrating that ChaCha20 symmetric encryption is measurably faster than ECC's asymmetric operation for short messages. Labels on top of each bar (not shown in the code but visible in the window) confirm the exact timings. This visualization helps users decide between stronger-but-slower ECC or faster-but-still-secure ChaCha20 when performance is a priority.

5. CONCLUSION

This project successfully demonstrates a unified framework that combines cryptographic encryption (ECC and

ChaCha20), LSB steganography, and zlib compression to securely hide secret messages within images. By encrypting the plaintext before embedding, the system ensures that even if an adversary discovers the LSB-encoded data, the payload remains unintelligible without the correct key. The ECC path leverages ECIES over the secp256k1 curve to provide strong, asymmetric confidentiality, while the ChaCha20 path offers faster, symmetric encryption. In both cases, computing a SHA-256 digest of the ciphertext and displaying it to the user enables reliable integrity verification on the receiver side. LSB embedding inserts the Base64-encoded ciphertext bitstream into the least significant bits of an RGB image, resulting in minimal perceptual distortion. Subsequent zlib compression reduces the final file size and helps obscure potential statistical artifacts. On the receiver side, decompression, LSB extraction, and SHA-256 integrity verification enable robust payload recovery and decryption. Performance benchmarks confirm that ChaCha20 encryption is significantly faster than ECC for short messages, offering users the flexibility to choose between speed and the added security of asymmetric encryption. The Tkinter GUI integrates all components, guiding users through uploading images, entering secrets, selecting encryption modes, and viewing logs and performance graphs. Overall, this integrated approach addresses the core limitations of traditional LSB steganography—namely, the lack of confidentiality, absence of integrity checks, and susceptibility to statistical analysis—while maintaining ease of use. By validating both ECC and ChaCha20 paths, the system strikes a practical balance between computational efficiency and security strength, making it well-suited for applications requiring covert, tamper-evident communication.

REFERENCES

1. M.A. Alohali, M. Aljebreen, F. Al-Mutiri, M. Othman, A. Motwakel, M.I. Alsaid, A.A. Alneil, A.E. Osman, Blockchain-driven image encryption process with arithmetic optimization algorithm for security in emerging virtual environments, Sustainability 15 (6) (2023) 5133.
2. Y. Zhu, C. Wang, J. Sun, F. Yu, A chaotic image encryption method based on the artificial fish swarms algorithm and the DNA coding, Mathematics 11 (3) (2023) 767.
3. Y. Luo, X. Ouyang, J. Liu, L. Cao, Y. Zou, An image encryption scheme based on particle swarm optimization algorithm and hyperchaotic system, Soft Comput. (2022) 1–27.
4. S. Khaitan, S. Sagar, R. Agarwal, Chaos Cryptosystem with Optimal Key Selection for Image Encryption, Multimedia Tools and Applications, 2022, pp. 1–16.
5. S.R. Maniyath, V. Thanikaiselvan, An efficient image encryption using deep neural network and chaotic map, Microprocess. Microsyst. 77 (2020) 103134.
6. J. Ferdush, G. Mondol, A.P. Prapti, M. Begum, M.N.A. Sheikh, S.M. Galib, An enhanced image encryption technique combining genetic algorithm and particle swarm optimization with chaotic function, Int. J. Comput. Appl. 43 (9) (2021) 960–967.
7. J. Zeng, C. Wang, A Novel Hyperchaotic Image Encryption System Based on Particle Swarm Optimization Algorithm and Cellular Automata, vol. 2021, Security and Communication Networks, 2021, pp. 1–15.
8. S.Y.D. Nezhad, N. Safdarian, S.A.H. Zadeh, New method for fingerprint images encryption using DNA sequence and chaotic tent map, Optik 224 (2020) 165661.
9. Ma, K., Teng, L., Wang, X. & Meng, J. Color image encryption scheme based on the combination of the fisher-yates scrambling algorithm and chaos theory. *Multimed. Tools Appl.* **80**, 24737–24757 (2021).

10. Zhao, J., Wang, S. & Zhang, L. Block image encryption algorithm based on novel chaos and DNA encoding. *Information* **14**(3), 150 (2023).