

## SECURE IMAGE STEGANOGRAPHY USING ECC AND CHACHA20 WITH COMPRESSION

J. Prashanthi<sup>1</sup>, P. Preethi<sup>2</sup>, M. Vignesh<sup>2</sup>, J. Tharun<sup>2</sup>, P. Ravinder<sup>2</sup>

<sup>1</sup>Assistant Professor, <sup>2</sup>UG Student, <sup>1,2</sup> Department of Computer Science and Engineering (Data Science),

<sup>1,2</sup>Sree Dattha Group of Institutions, Sheriguda, Ibrahimpatnam, 501510, Telangana.

### To Cite this Article

J. Prashanthi, P. Preethi, M. Vignesh, J. Tharun, P. Ravinder, "Secure Image Steganography Using Ecc And Chacha20 With Compression", *Journal of Science Engineering Technology and Management Science*, Vol. 02, Issue 08, August 2025, pp: 658-668, DOI: <http://doi.org/10.64771/jsetms.2025.v02.i08.pp658-668>

Submitted: 15-07-2025

Accepted: 21-08-2025

Published: 28-08-2025

### 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.

This is an open access article under the creative commons license

<https://creativecommons.org/licenses/by-nc-nd/4.0/>



## 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].

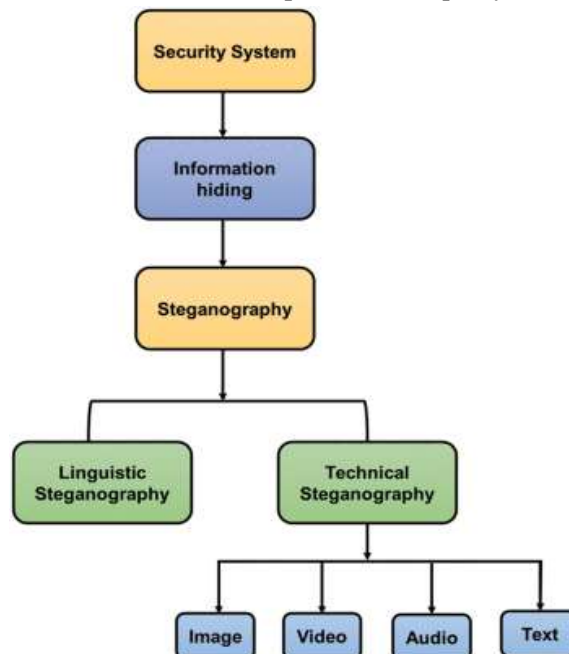


Fig.1: Steganography Classification.

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**

The symmetric encryption technique is one of the oldest and most famous methods of maintaining data security; the secret key can be a text or several random characters. The secret key is implemented by a text message to change its content [1]. The method of encryption used in this technique is to convert each character to several alphabetic characters when the sender and recipient know the keys of all parties and then use the secret key to encrypt and decrypt the message. Blowfish is an encryption technique, more specifically a block cipher. It is frequently referred to as a cipher. Blowfish uses keys ranging from 3 to 2448 bits and has a 64-bit block size. Its creator, Bruce Schneier, claims it is free to use, open source, and royalty free. Although Blowfish is utilized in several cipher suites and encryption methods, AES is often employed. Blowfish is secure since no cryptanalysis attempt has succeeded [2,3].

The tool (AES) was chosen because it meets our objectives. We need a powerful encryption mechanism without the complexity of sharing multiple keys as asymmetric encryption, especially when demanding to send the parameter required for extracting the data on a secure tunnel. AES is a symmetric block encryption that uses a 128-bit encryption key, which is very hard to break. Also, the block encryption mechanism makes it possible to predict the encryption data using their language characteristic since it replaces the same data block with different coding types and sizes each time. The AES overcomes the short key use by 3DES and other symmetric encryption, which is technically the best choice for our proposed algorithm [4]. Text steganography principles can be applied to a standard text file by hiding the message in the header information of the file. Using it this way helps maintain the original text file and keep it the same without any change in the content itself; on the other hand, this mechanism increases the size of the file. Another mechanism replaces bits of the text file content and changes the original text, but it is not widely used since it makes the text unreadable. Text steganography can also be applied to PDF documents or other text standards [5].

The Least Significant Bit (LSB) is implemented by replacing the least significant bit in the cover image with the bit from a secret message; the LSB method hides the binary values '101100101' in a 24-bit image. The algorithm starts by uploading the cover image and the secret message, and then an end marker represented by an array of characters is added to the secret message [6]. This is performed to allow the awareness of the recovery phase of when to stop recovering in case the secret message bit's numbers do not need the full cover image pixels to be hidden since, in this case, the recovery tool continues to extract wrong data if no ending marker is found. After that, the algorithm receives each character in the secret message using a loop. It converts it to a binary value and selects three sequence pixels from the cover image based on the index of the count parameter, which starts from zero and is incremented each time a pixel is selected. Later, the algorithm extracts the three-colour components, R, G, and B, for each pixel chosen and converts each colour component to a binary value. It masks the least bit of each colour component to zero by applying (AND with 11111110). This allows the replacement of the secret character binary bits by the least bit in each colour component, which is performed later by applying XOR with one bit from the selected character bits. It is worth mentioning that the least significant bit (LSB) algorithm has been improved using the selected least considerable bit (SLSB); these techniques proposed to enhance the performance of the LSB method by hiding information only in one of the three colours at each pixel of the cover image to reduce the chance of the confidential data being detected [7,8].

Peak signal-to-noise ratio (PSNR) measures the quality of the stego image compared with the cover image in decibels. The higher the PSNR, the better the quality. Where MAX is the maximum possible pixel value of the image, this metric describes the proportion of a signal's maximal strength to the

power of corrupting noise that compromises the accuracy of its representation. This metric determines signal reconstruction quality, like video coding and data encoding [9].

Visual attack is the only stego attack that allows attackers visual analysis of the stego image. The most common attack of this type is the display of the least significant bit of an object as a binary image and analysis of the resulting image for differences between pixel shows. This attack is too slow and expensive [10]. A statistical attack assumes that the least significant bit of the cover file is random. This attack aims to compare the frequency distribution of the potential cover file with the expected distribution of the cover file. If the new data are not the same as the statistical data expected to contain the standard data, then it probably contains confidential data. It is worth mentioning that the distribution analysis is performed via mathematical methods like the standard deviation of the image histogram, which makes the analysis much faster and widely used [11]. Compared to some state-of-the-art techniques, the proposed method has demonstrated improved perceptual transparency measures, such as peak signal-to-noise ratio and the structural similarity index. Additionally, the proposed method has exhibited high resistance to stego attacks, such as pixel difference histograms and regular and singular analysis. The out-of-boundary pixel issue in many current data-hiding techniques has also been effectively addressed in work [12]. The study of information hiding holds great significance in data security. With the advancements in computational technology, covert communication methods have become widely recognized among researchers and common data communication participants.

Image steganography is one of the top choices among experts in data hiding. The biggest challenge in designing a steganographic system is to balance measures such as the quality of the cover image, capacity, and robustness to various attacks [13]. This study aims to comprehensively review various existing image steganography techniques concerning their performance evaluation standards. The challenges faced and the future directions of this field are also discussed [14]. A comprehensive performance evaluation of reversible image steganography techniques was conducted. Techniques from the past two decades were compared using standard test images, with PSNR and embedding capacity as the evaluation parameters. The results of each technique were tabulated, analyzed, and compared to provide a clear understanding of their performance. Descriptive statistics and an in-depth analysis were also performed to assess these past techniques of the standard test [15]. A study was conducted on the scaling parameter, alpha, in the context of image steganography. Various cover and payload images, including live images from a webcam, were preprocessed and normalized. The cover and payload images were then subjected to Haar Discrete Wavelet Transformation (DWT).

### **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.

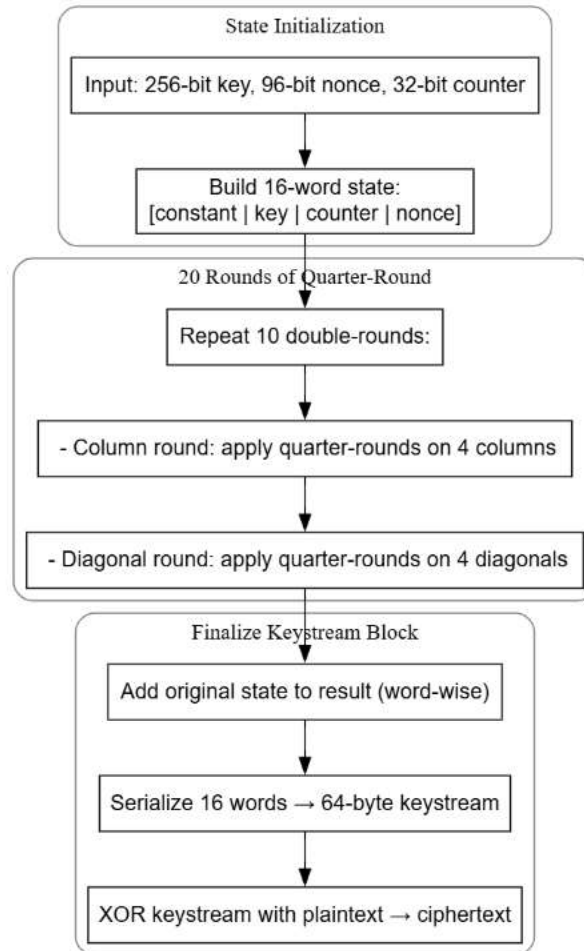


Fig.2: Internal operational flow of ChaCha20.

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.

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.

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.

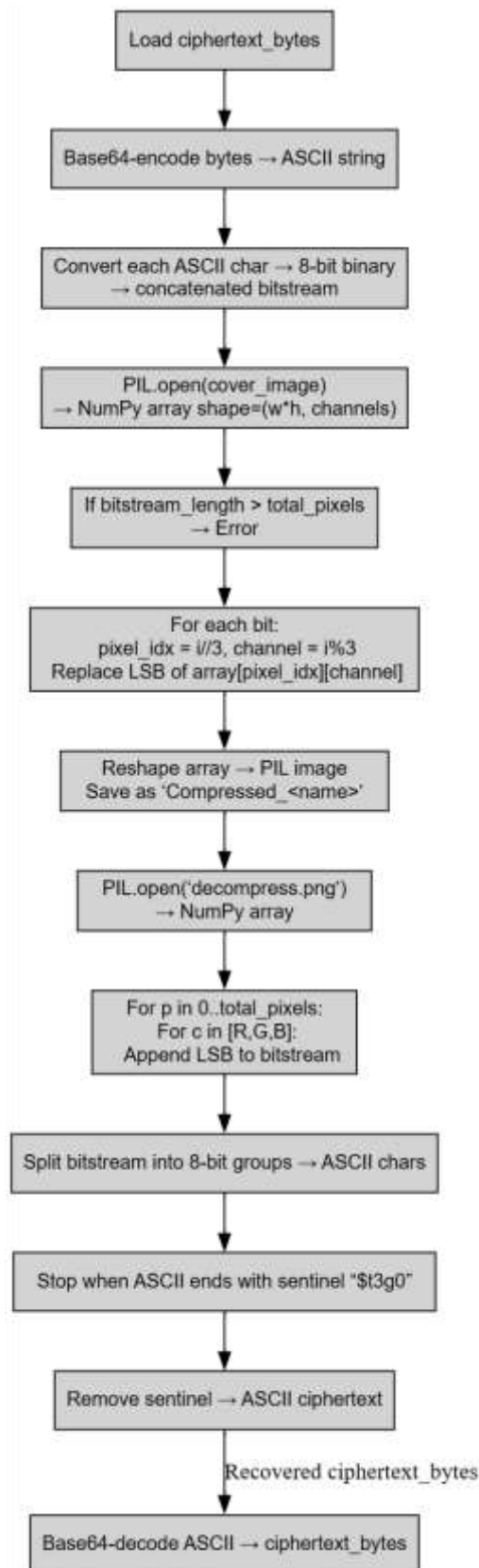


Fig.3: LSB steganography internal operational flow.

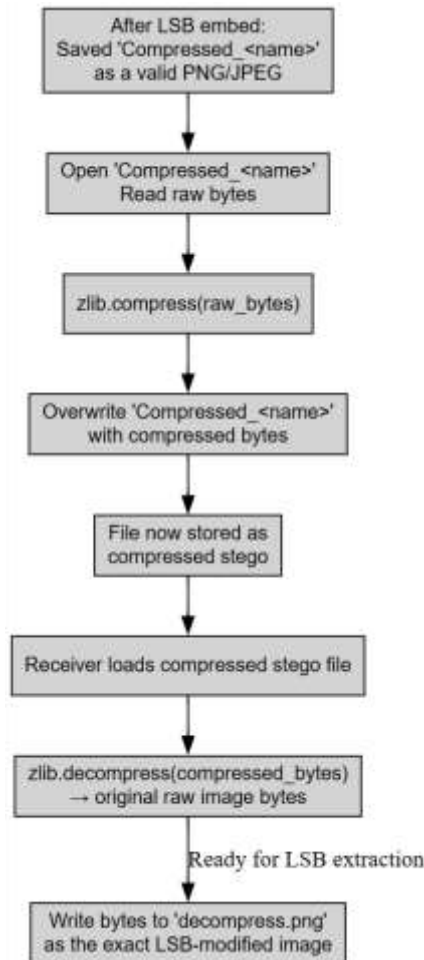


Fig.4: Compression and decompression using zlib.

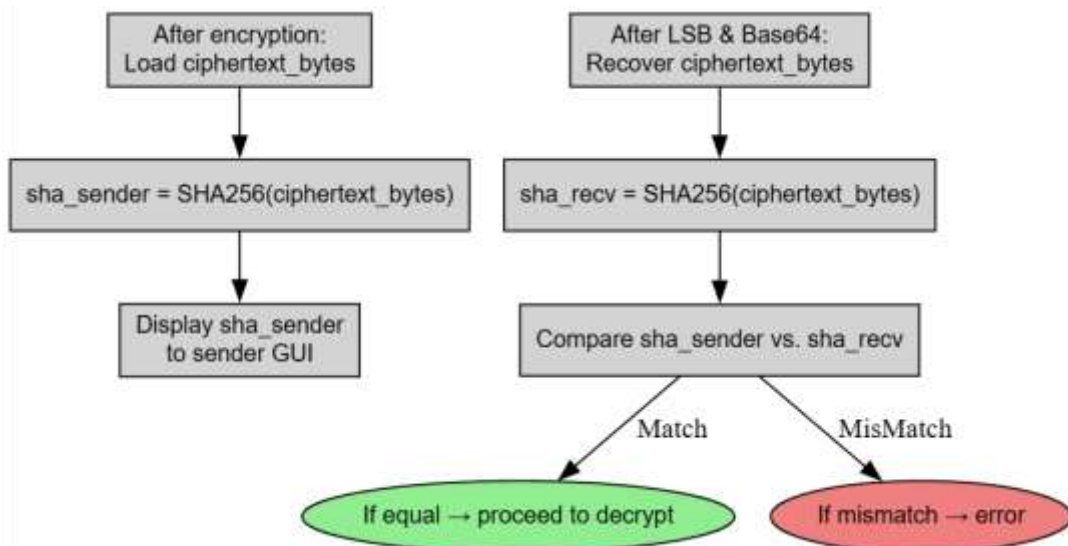


Fig.5: SHA-256 integrity verification operational flow.

#### 4. RESULTS AND DISCUSSION

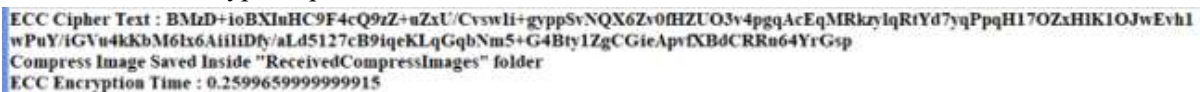
The ECC encryption process began by either loading or generating the ECC public key (pri.key), which was then used to encrypt the message “Hi how are you. We will meet at 7pm” using the ECIES algorithm. Following encryption, the system computed the SHA-256 hash of the ciphertext and inserted the resulting 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.



```

ECC Cipher Text : BMzD+ioBXInHC9F4cQ9zZ+nZaU/Cvswli+gyppSvNQX6Zs0fHZUO3v4pgqAcEqMRkzylqRtYd7yqPpqH17OZaHIK1OJwEvh1
wPuY/IGVu4kKbM6lx6AiiIdfy/aLd5127cB9iqeKLqGqbNm5+G4Bty1ZgCGieApvEXBdCRRu64YrGsp
Compress Image Saved Inside "ReceivedCompressImages" folder
ECC Encryption Time : 0.2599659999999915
  
```

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.

```
Receiver Generated SHA code = fa6ecd7c82c7de119eed412a6c6993486ebe2231a5c643014685adc742857983
CHACHA Cipher Text : 45MCvq9rY8c66HgevpMHc8rGi71AbIHdjow/pXjC3rTTUGQ=
Compress Image Saved Inside "ExtensionCompressImages" folder
CHACHA20 Encryption Time : 0.2214236000000085
```

Fig.7: GUI application after performing uploading sender side image, and secrete message embedding using compress and send operation (i.e., CHACHA20 computation).

```
Extension Receiver Generated SHA code = e98a9a6f9f7e1f5053dd408c99cbddb1c72a0317be475a8d8283df0cd80b402
ChaCha20 Extracted Hidden Secured Message = Hi how are you. We will meet at 7pm
```

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.

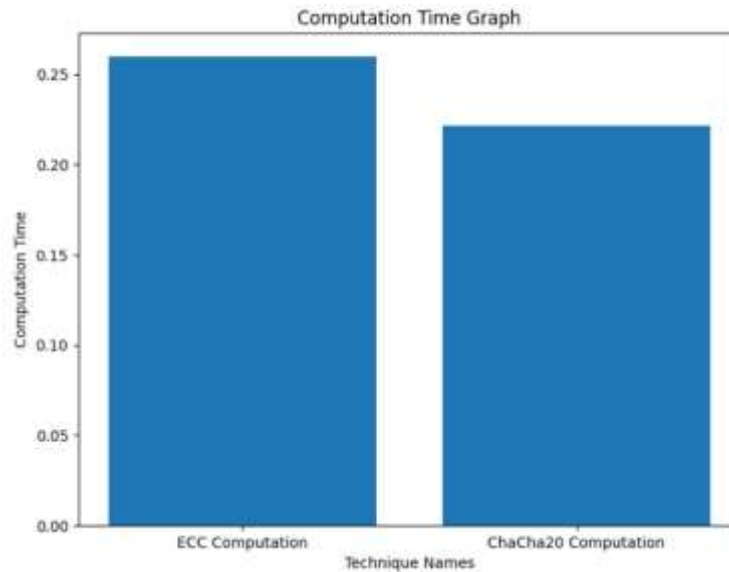


Fig. 9: Computational time performance using ECC, and CHACHA20 algorithms.

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. Rajasekar, V.; Premalatha, J.; Dhanaraj, R.K.; Geman, O. Introduction to Classical Cryptography. In *Quantum Blockchain: An Emerging Cryptographic Paradigm*; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2022; pp. 1–29.
2. Aumasson, J.P. *Crypto Dictionary: 500 Tasty Tidbits for the Curious Cryptographer*; No Starch Press: San Francisco, CA, USA, 2021.
3. Saad Almutairi, S. Manimurugan, M.A. A new secure transmission scheme between senders and receivers using HVCHC without any loss. *EURASIP J. Wirel. Commun. Netw.* 2019, 2019, 1–15.
4. Altigani, A.; Hasan, S.; Barry, B.; Naserelden, S.; Elsadig, M.A.; Elshoush, H.T. A polymorphic advanced encryption standard—a novel approach. *IEEE Access* 2021, 9, 20191–20207.
5. Semenchenko, O.; Iakovenko, O.; Lekakh, A.; Parkhomenko, M.; Podlesny, S.; Karaban, O. The Analysis of the Codes in the Textual Steganography Technologies. In Proceedings of the 2021 IEEE 3rd International Conference on Advanced Trends in Information Theory (ATIT), Kyiv, Ukraine, 15–17 December 2021; pp. 31–35.
6. Pandey, D.; Wairya, S.; Al Mahdawi, R.S.; Najim, S.A.D.M.; Khalaf, H.A.; Al Barzinji, S.M.; Obaid, A.J. Secret data transmission using advanced steganography and image compression. *Int. J. Nonlinear Anal. Appl.* 2021, 12, 1243–1257.
7. Xian, Y.J.; Wang, X.Y.; Zhang, Y.Q.; Wang, X.Y.; Du, X.H. Fractal sorting vector-based least significant bit chaotic permutation for image encryption. *Chin. Phys. B* 2021, 30, 060508.
8. Yang, D. Correlogram, predictability error growth, and bounds of mean square error of solar irradiance forecasts. *Renew. Sustain. Energy Rev.* 2022, 167, 112736.
9. Anjum, U.; Hussain, A.; Ali, C.B.; Afzal, U.; Hussain, I.; Noorwali, A.; Shah, S.A. JPEG Image Compression Using Multiple Core Strategy in FPGA achieving High Peak Signal to Noise Ratios. In Proceedings of the 2021 International Congress of Advanced Technology and Engineering (ICOTEN), Taiz, Yemen, 4–5 July 2021; pp. 1–6.
10. Vennam, P.; TC, P.; BM, T.; Kim, Y.G.; BN, P.K. Attacks and preventive measures on video surveillance systems: A review. *Appl. Sci.* 2021, 11, 5571.
11. Mustafa, M.S. An Effect Image Steganography System Based on Pixels Disparity Value and Secret Message Compression. Master's Thesis, Altınbaş Üniversitesi/Lisansüstü Eğitim Enstitüsü, Istanbul, Turkey, 2022.
12. Sahu, M.; Padhy, N.; Gantayat, S.S.; Sahu, A.K. Local binary pattern-based reversible data hiding. *CAAI Trans. Intell. Technol.* 2022, 7, 695–709.
13. Sahu, M.; Padhy, N.; Gantayat, S.S.; Sahu, A.K. Performance analysis of various image steganography techniques. In Proceedings of the 2022 Second International Conference on Computer Science, Engineering and Applications (ICCSEA), Gunupur, India, 8 September 2022; pp. 1–6.
14. Malarvizhi, N.; Priya, R.; Bhavani, R. Reversible Image Steganography Techniques: A Performance Study. In Proceedings of the 2022 7th International Conference on Communication and Electronics Systems (ICCES), Coimbatore, India, 22–24 June 2022; pp. 780–787.
15. Tevaramani, S.S.; Ravi, J. Image steganography performance analysis using discrete wavelet transform and alpha blending for secure communication. *Glob. Trans. Proc.* 2022, 3, 208–214.