Securing UAV Flight Data Using Lightweight Cryptography and Image Steganography

Orkhan Valikhanli, Fargana Abdullayeva
Institute of Information Technology, Baku, Azerbaijan

***Abstract*—The popularity of Unmanned Aerial Vehicles (UAVs) in various fields has been rising recently. UAV technology is being invested in by numerous industries in order to cut expenses and increase efficiency. Therefore, UAVs are predicted to become much more important in the future. As UAVs become more popular, the risk of cyberattacks on them is also growing. One type of cyberattack involves the exposure of important flight data. This, in turn, can lead to serious problems. To address this problem, a new method based on lightweight cryptography and steganography is proposed in this work. The proposed method ensures multilayer protection of important UAV flight data. This is achieved by two layers of encryption using a polyalphabetic substitution cipher and ChaCha20-Poly1305 authenticated encryption, as well as randomized least significant bit (LSB) steganography. Most importantly, through this work, a balance is kept between security and performance. Additionally, all experiments are carried out on real devices, making the proposed method more practical. The proposed method is evaluated using MSE, PSNR, and SSIM metrics. Even with a capacity of 8000 bytes, it achieves an MSE of 0.04, a PSNR of 62, and an SSIM of 0.9998. It is then compared to existing methods. The results show better practical use, stronger security, and higher overall performance.**

***Keywords*—UAV; GCS; cyberattack; cryptography; steganography; flight data**

I. Introduction

UAVs have become popular in many different fields including scientific research, agriculture, military, surveillance, aerial photography, delivery services, infrastructure inspections, and more. UAVs can do many tasks quickly and are cheaper compared to other traditional methods. Moreover, they can also perform complex tasks efficiently and reduce operational risks. This is the reason UAVs have become common across various industries.

However, UAVs also face significant challenges related to their cybersecurity. There are many types of cyberattacks targeting UAVs. A cyberattack on a UAV could result in loss of control, data leakage, mission failure, and even injuries or death [1, 2]. In December 2011, the American UAV RQ-170 Sentinel was captured by Iranian forces. Both, GPS spoofing and GPS jamming attacks were performed to capture the UAV [3]. In December 2009, UAV video feed recordings were discovered after capturing militants. Militants used SkyGrabber software to capture satellite videos using the satellite antenna [4]. Since the videos were not encrypted, the militants were able to take advantage of this vulnerability. Ground Control Stations (GCSs) of UAVs are also vulnerable to various cyberattacks. In September 2011, a keylogger virus was detected in the GCS of Predator and Reaper UAVs. According to reports, technicians attempted to delete the virus however, it kept reappearing [3, 5]. Hooper et al. [6] demonstrated that commercial UAVs are vulnerable to common security attacks. To prove this, authors performed buffer overflow, Denial of Service (DoS), and ARP cache poisoning attacks. All experiments showed that some commercial UAVs are vulnerable to those attacks.

In this study, a novel method for the protection of important UAV flight data is proposed. The proposed method is multilayered which consists of three main phases. First, data from the flight controller of the UAV is encrypted using a polyalphabetic substitution cipher. Second, lightweight ChaCha20-Poly1305 cryptography is implemented to encrypt data to increase security. Finally, randomized LSB steganography is used to hide data in an image. Only after completing all those steps, the stego image is sent to GCS. Subsequently, the GCS extracts encrypted data from the stego image and then decrypts it to reveal the actual important flight data. Overall, the proposed method uses three different keys shared between the UAV and the GCS. One key is used for the polyalphabetic cipher to randomize blocks. Second key is used for lightweight ChaCha20-Poly1305 cryptography. The third key is used for randomized LSB steganography. Moreover, a shared secret constant is used to add more security to the system. The main contributions of this study are as follows:

- This study proposes a novel seed derivation approach. The proposed approach offers a high level of security.
- The proposed work uses the polyalphabetic substitution cipher to add initial security to the system. In the proposed scheme, not only does the same character within a single word differ, but it also differs for each operation (for each time an image is sent). This approach indeed increases security.
- The proposed work uses ChaCha20-Poly1305, a lightweight authenticated encryption scheme. While ChaCha20 provides the stream cipher function, Poly1305 handles message authentication. This approach ensures the confidentiality and integrity of the data.
- The proposed work uses randomized LSB with a key instead of simply LSB. Moreover, similar to the first contribution, the position of information in the image is different for each operation. This makes the detection of hidden information inside the image even more difficult.

- The results of various experiments demonstrate that the proposed method outperforms others in both security and imperceptibility. Furthermore, the experiments were conducted in a real environment (a real flight computer) rather than a simulated one. This is important from a practical perspective.

The remainder of this study is organized as follows: Section II presents related works. In Section III, the problem statement and methods are given. Section IV presents the proposed multilayer protection method, including algorithms and the operational workflow of the system. In Section V, the results of experiments are demonstrated. Section VI concludes this work and discusses future work.

II. Related Works

There are numerous cryptography and steganography methods available for securing data. However, only a limited number of studies focus on UAVs. Due to this limitation, similar systems such as IoT are also analyzed in this work. Alkodre et al. [7] presented a shuffling-steganography algorithm to protect UAV data. The proposed algorithm is hybrid which uses both text-based and image-based steganography. The main idea of the method is to divide data based on a pattern, then hide a part in a text cover, and another part in an image. For encrypting the data authors used Data Encryption Standard (DES) and also implemented Advanced Encryption Standard (AES). For image steganography, however, LSB is used. Lin et al. [8] proposed an XOR-based encoding strategy to transform secret digits into smaller ones. Moreover, frequency-based encoding is used to hide data in two images. One of the images is stored in the UAV to avoid interception attacks. The second image, however, is sent to command station. The proposed method determines whether the second image has been altered by extracting the secret data from the two images after the UAV mission is over. Rodríguez Marco et al. [9] proposed new techniques for transmitting information to the GCS, making it possible to accurately know the aircraft performance in icing conditions. The idea is to use onboard cameras to capture icing conditions on wings and stabilizers. Moreover, information is hidden inside captured images using LSB steganography before sending it to GCS. Syed et al. [10] used steganography to hide UAV images within audio file. To hide the data, the authors used LSB coding with XOR operation. After hiding the data, the audio file was transmitted to GCS, where the image was extracted. A secret key was used during both the embedding and extraction processes. Alarood et al. [11] proposed a stenographic technique to ensure privacy and authenticity in Internet of Things (IoT) networks. The proposed stenographic technique is based on the pixel characteristics of the cover image in the spatial domain. The main idea is to classify pixels into highly smooth and less smooth domains to select the extra eligible pixels. Hassaballah et al. [12] proposed an image steganography method to secure data in the Industrial Internet of Things (IIoT). The proposed method embeds secret data in the cover images using a metaheuristic optimization algorithm called Harris Hawks optimization to effectively choose image pixels that can be used to hide bits of secret data within integer wavelet transforms. AlEisa [13] used steganography to embed the patient's personal information in their medical images to enhance confidentiality in case of a distant diagnosis. IoT is used to enhance medical data security in order to preserve confidentiality and integrity. As a steganography method, the LSB of the approximate coefficient of integer wavelet transform is used. Rostam et al. [14] proposed a combination of chaos functions and steganography method based on image blocking to preserve IoT privacy. Block centers are used to generate the initial key of the chaos function. Subsequently, randomly selected secret data bits are hidden in the pixels of randomly selected blocks.

The analyzed works have some limitations. Most of them focus only on security at the steganography level and use weak encryption methods or don't use cryptography at all. Many works also ignore the fact that devices have limited resources, which can affect how well their methods work. In this work, however, all mentioned issues are considered.

III. Problem Statement and Methods

Flight data of UAV is important as it contains all the necessary information about the status, operation, and performance of the UAV. On the other hand, unprotected flight data poses a serious risk. This is because attackers may intercept it and take advantage of it. Moreover, some flight data is even more essential and should be protected at all costs. For instance, let's consider a situation where a military UAV flies over enemy territory. In this situation, important flight information, such as position, altitude, etc. of the UAV should be secure. If not, the UAV may be located and captured. This could indeed create more problems. Videos, images, or other secret information recorded by the UAV may be revealed to the enemy. To solve this problem, it's necessary to implement various techniques to secure necessary flight information. One solution to this problem is to use cryptography to encrypt data. However, encryption alone might not be sufficient. In this situation, steganography is essential. By embedding the encrypted data within files, steganography adds an additional layer of security. This indeed makes sensitive information less detectable. The combination of cryptography and steganography provides a robust approach to secure important flight data. Considering all mentioned above, general information about cryptography, steganography, cryptographic hash functions, pseudorandom number generators (PRNGs) etc. will be presented in this section.

A. Cryptography

The term "cryptography" derives from two Greek words: κρυπτός (kryptos) – "secret" and γραφω (grapho) – "write" [15]. Cryptography is the science of secret writing with the goal of hiding the meaning of a message [16]. Mainly two types of cryptography are used for the encryption of sensitive data. These are symmetric cryptography and asymmetric cryptography. In symmetric cryptography, the same key is used for both encryption and decryption. In asymmetric cryptography, however, a pair of keys are used: a public key for encryption and a private key for decryption. Moreover, the public key is shared openly and the private key is kept secret. For each type of cryptography, different algorithms were introduced. Symmetric cryptography uses algorithms like AES, DES, Triple Data Encryption Standard (3DES), ChaCha20, and others. Asymmetric cryptography on the other hand uses

algorithms like Rivest-Shamir-Adleman (RSA), Elliptic Curve Cryptography (ECC), and others. In our context, some of the symmetric cryptography algorithms will be considered.

1) Data Encryption Standard and 3DES: Data Encryption Standard (DES) was developed in the 1970s and later adopted as a standard by the U.S. National Institute of Standards and Technology (NIST) in 1977. Moreover, DES itself is based on the Lucifer cipher, developed by Horst Feistel [16]. The block size of DES is 64 bits and the key size is 56 bits. Because of its relatively short key size, DES is now considered unsafe.

To overcome the limitations of DES, 3DES was introduced. 3DES increases security by applying the DES algorithm three times to each data block. This is possible by using two or three unique 56 bit keys. Thus, 3DES supports key sizes of 112 and 168 bits. 3DES uses an encrypt-decrypt-encrypt (EDE) scheme. If the two-key version is implemented then key 1 is used to encrypt data. Afterwards, key 2 is used to decrypt the same data. Finally, key 1 is used again for encryption. However, if the three-key version is implemented, then key 1 is used to encrypt data. Afterwards, key 2 is used to decrypt the same data. Finally, key 3 is used for encryption. While 3DES has stronger security compared to DES, it is computationally intensive and slower.

2) Advanced encryption standard: In 1997 the NIST called for proposals for a new Advanced Encryption Standard (AES). In 2001, NIST declared the block cipher Rijndael as the new AES and published it as a final standard [16]. Rijndael is named after cryptographers, Joan Daemen and Vincent Rijmen. AES was developed to address the weaknesses and limitations of DES. The block size of AES is 128 bits. AES supports 128, 192, and 256 bits key sizes. It's important to say that the performance of the AES can vary depending on whether a processor has built-in hardware acceleration for AES operations. Processors without AES hardware support depend on software-based implementations. This approach can be slower because AES operations are computationally intensive.

3) Chacha20 and Chacha20-Poly1305: Chacha20 is a modern and efficient stream cipher designed by Daniel J. Bernstein [17]. It is a modified version of the Salsa20 cipher. Chacha20 uses 512 bit blocks and the key size is 256 bits. Moreover, it also uses 96 bit nonce for encryption. In most cases, Chacha20 is faster and more efficient than traditional ciphers like AES. This makes Chacha20 suitable for systems like IoT, UAV, and others.

In the Chacha20-Poly1305 combination, Chacha20 is a stream cipher and Poly1305 is a message authenticator. Poly1305 is a message authentication code (MAC) algorithm that ensures authenticated encryption by generating a tag to verify the integrity and authenticity of the encrypted message [18]. There are various protocols that use Chacha20-Poly1305 including Secure Shell Protocol (SSH), Transport Layer Security (TLS), etc.

4) Monoalphabetic and polyalphabetic ciphers: Monoalphabetic and polyalphabetic ciphers are substitution ciphers. Substitution ciphers are block ciphers that replace symbols (or groups of symbols) with other symbols or groups of symbols [19]. In monoalphabetic ciphers, a single substitution rule is applied throughout the entire message. This means that every letter in the ciphertext always matches the same letter in the plaintext. Polyalphabetic ciphers, however, use multiple substitution rules to encrypt the message. This means that the same letter in the plaintext matches different letters in the ciphertext. Polyalphabetic ciphers are harder to break than monoalphabetic ciphers, particularly if the key is unknown.

B. Cryptographic Hash Functions

1) Hash functions take a message as input and generate a fixed-size output referred to as hash value or simply hash. To be more specific, a hash function h maps bitstrings of arbitrary finite length to strings of fixed length, say n bits [19]. Cryptographic hash functions should have two main properties to be secure. These are preimage resistance (one-wayness) and collision resistance. Preimage resistance means that reversing the hash value to get the original input should be infeasible. Collision resistance however means that it should be infeasible to find two different inputs that produce the same hash value. There are also two main types of hash functions such as keyless and keyed. Keyless hash functions don't require a key to operate. MD4, MD5, and SHA-256 are some examples of keyless hash functions. Keyed hash functions require a key to operate. The key provides an additional level of protection. Hash-based Message Authentication Code (HMAC) is one of the best examples of keyed hash functions. While keyless hash functions provide data integrity, the keyed hash function provides both data integrity and authentication. Therefore, each type of hash function has its own usage.

2) Cryptographic hash functions have a wide range of applications. For instance, hash is used to verify data integrity. They ensure that data has not been modified during transmission. Hash is also used to store passwords in a database securely. This is possible by only storing the hash value of the password instead of storing it as plain text in the database. Later, during the authentication process, the hash of the entered password is compared to the stored hash. Moreover, digital signatures use hashes to generate a unique fingerprint of the data.

C. Pseudorandom Number Generators

Pseudorandom number generators (PRNGs) are algorithms designed to generate sequences that are computed from an initial seed value [16]. The seed value is the starting point for PRNG. Using seed value, PRNG generates a sequence of numbers using a deterministic algorithm. Since the algorithm is deterministic, the same seed will always produce the same sequence of numbers. There are many proposed PRNG algorithms including Mersenne Twister, linear congruential generator (LCG), middle-square, blum blum shub (BBS), permuted congruential generator (PCG), linear feedback shift register (LFSR), etc. PRNGs are used in a wide range of applications such as cryptography, modeling, statistical analysis, gaming, and others. Cryptographically secure pseudorandom number generators (CSPRNGs) are a special type of PRNGs designed to be unpredictable and resistant to cyberattacks.

D. Steganography

The word steganography is a composite of the Greek words steganos, which means "covered", and graphia, which means "writing" [20]. Steganography is a technique of hiding secret information within common data or objects to evade detection. Thus, it is possible to hide secret information in various media formats such as text, image, audio, and video. Additionally, the secret information itself may be in any of these formats. For example, in the case of text steganography, punctuation and spacing can be modified to hide information. Audio steganography uses techniques such as modifying frequencies or embedding secret data into the LSB of the audio signal. Video steganography may modify frames or pixels to hide information. Image steganography uses techniques of Discrete Cosine Transform (DCT) and Discrete Wavelet Transform (DWT). However, LSB embedding is the most common technique due to its simplicity and low computational requirements. As the name suggests, LSB technique modifies the least significant bits of pixel values of an image. Since these bits barely affect the pixel color, the changes are unnoticeable to the human eye. There are also some terms used in steganography such as cover object, stego object, and stego key. A cover object refers to the original object used as a carrier for secret information. A stego object is the result of embedding secret data into the cover object [20]. A stego key is a secret key used during the embedding process to control, where and how the secret data is embedded.

IV. The Proposed Work

In this work, we present a novel approach that combines lightweight cryptography and steganography techniques to securely embed important flight data into images captured by UAV. These images are then sent to the GCS. In GCS, hidden flight data is extracted from images. The resource limitations that come with UAVs were carefully taken into account when building the suggested system. It is well known that unlike computers, servers, or other systems, UAVs may not always have sufficient computational resources (CPU, RAM, etc.). This especially applies to small-sized UAVs. To address these limitations, the proposed approach keeps an optimal balance between security and performance. The structure of the UAV and GCS with their main components is described in Fig. 1.

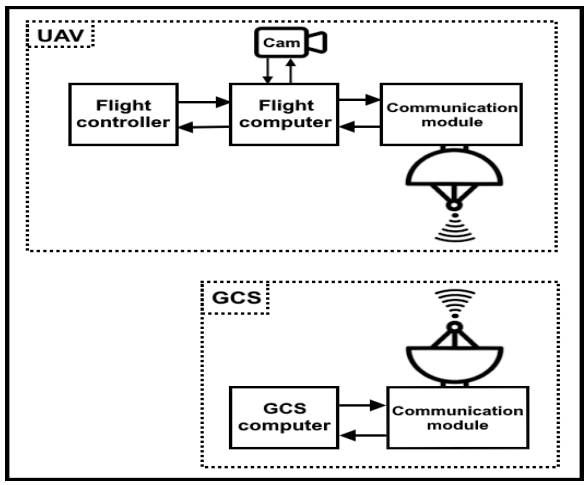

Fig. 1. The structure of UAV-GCS for proposed work.

A. The Proposed Algorithms

This subsection discusses the proposed algorithms separately. Later, it will be explained how these algorithms work together to demonstrate the functioning of the entire system. Some of the algorithms run on both the UAV and the GCS, while others run only on one of them. The distinction between these algorithms is based on their specific roles, such as initialization, seed derivation, encryption, decryption, embedding, and extraction. Each algorithm includes specific parameters, and some are executed several times on both the UAV and the GCS.

1) Initialization for polyalphabetic substitution: Initialization for polyalphabetic substitution is performed both on UAV and GCS. This step is used to create blocks for polyalphabetic cipher encryption and reversed blocks for polyalphabetic cipher decryption. As seen in Algorithm 1, during block generation, multiple substitution blocks are generated using the defined characters. Each block maps one character to another character by shifting its position in the list of characters. For instance, the letter "A" may map to the letter "B" in one block, to the letter "C" in the next block, and so on. For the decryption process, the code also creates a reversed version of each block. In the reversed blocks, the mapping is flipped. This allows retrieving the original character from the substituted one. The final step returns blocks (for encryption) and reversed blocks (for decryption).

Algorithm 1 Initialization for polyalphabetic substitution

Input: characters
Output: blocks and reverse_blocks

1. Generating substitution blocks:
1.1. Initialize an empty list to store blocks: blocks ← []
1.2. **for** shift **in** 0 **to** length(characters)-1:
    block←CreateMap(characters, ShiftChrs(characters, shift))
    blocks.add(block)
  end for
2. Generate reverse blocks:
2.1. Initialize an empty list to store rev. blocks: reverse_blocks ← []
2.2. **for** block **in** blocks:
    reverse_block ← ReverseMapping(block)
    reverse_blocks.add(reverse_block)
  end for
3. Return blocks and reverse_blocks

2) Seed derivation: Deriving seeds is one of the most important steps, because in our proposed approach, seeds are both used by polyalphabetic cipher and randomized LSB. The process of seed derivation in this work uses a cryptographic hash function. One of the properties of hash functions is their avalanche effect. The avalanche effect ensures that a small change in the input produces a significantly different output. In this work, date and time are used as the source of change. Since date and time values vary constantly, the hash function reacts to these small differences. As a result, the seed value becomes highly sensitive to even small changes. Therefore, this property of the hash function is taken advantage of in this work. However, using a keyless hash for our approach has potential risks. This is because attackers may analyze how the

seed information is generated. Initially, they could determine whether date and time information is used for seed. Then, they could try all hash functions and try to generate seeds themselves. To solve this problem, HMAC-SHA256 is implemented. Since HMAC-SHA256 requires a key to generate a hash, it will be difficult for attackers to achieve their goals. Furthermore, additional secret constant parameter is also used in input along with date and time. The main idea behind using both date and time information along with a secret constant is to increase security. Generally, using only date and time information to derive a seed is secure in our system because the key size is sufficient to provide security. However, including a secret constant in the system makes it even more secure. Even with the secret key, an attacker will be unable to determine exactly which information the hash is derived from. Moreover, the attacker must know both the key and the secret constant to get the correct hash value. Thus, while date and time information make the system dynamic, the secret constant increases its security even further. Algorithm 2 demonstrates the proposed seed derivation algorithm.

Algorithm 2 Seed derivation

Input: key, date_time and secret_constant
Output: derived_seed

1. Combining input data:
combined_data ← CombineInputData(date_time, secret_constant)
2. Convert combined data into a byte encoded representation:
encoded_data ← encode (combined_data)
3. Compute keyed-hash HMAC-SHA256:
hash ← HMAC-SHA256 (key, encoded_data)
4. Extract the first 8 bytes of the hash and represent them as a big-endian integer seed:
derived_seed ← IntegerFromBytes(hash[0:8],"big-endian")
5. Return derived_seed

As seen in Algorithm 2, the first step is combining input data including date and time information and secret constant. In the next step, combined data is encoded. The encoded data then is passed to HMAC-SHA256 as a parameter, along with the secret key. The length of the secret key is 256 bits. After generating the hash, the first 8 bytes (64 bits) of hash value are selected. This is because for our case the input of the PRNG function requires 8 bytes of data.

3) Encryption using polyalphabetic cipher (UAV): As mentioned earlier, polyalphabetic cipher uses multiple substitution rules to encrypt the data which makes them more secure compared to monoalphabetic ciphers. Therefore, the polyalphabetic cipher is selected as initial encryption method in this work. Algorithm 3 demonstrates the process of encryption using the polyalphabetic cipher. During initialization, an empty list is assigned to ciphertext and then PRNG is initialized using seed. As a PRNG function, Small Fast Chaotic 64 (SFC64) is selected due to its high performance. The PRNG generates random values, which are used to select substitution blocks. The seed ensures that the same sequence of random values is produced each time encryption is performed. This in turn makes the process deterministic and reproducible for decryption. After initialization, the algorithm processes every character in the input plaintext. A random substitution block is chosen if the character is part of the specified character set. The character is then replaced with its counterpart from the chosen block. If a character is not found in the character set then it remains unchanged. Finally, the ciphertext is returned.

Algorithm 3 Encryption using polyalphabetic cipher

Input: plaintext, seed, characters, and blocks
Output: ciphertext

1. Initialize an empty list to ciphertext:
ciphertext ← []
2. Initialize random generator rng with seed:
rng ← InitializeRNG(seed)
3. Process every character and replace it with its counterpart from blocks:
3.1 **for** each char **in** plaintext:
3.2 **if** char **in** characters:
block ← GetRandomBlock(rng, blocks)
ciphertext.add(block[char])
else:
ciphertext.add(char)
end if
end for
4. Return ciphertext

4) Encryption using ChaCha20-Poly1305: Using polyalphabetic cipher only may not secure the important data. To increase the security even further, ChaCha20-Poly1305 is implemented as the main encryption method. The encryption process for ChaCha20-Poly1305 is demonstrated in Algorithm 4. First, ChaCha20-Poly1305 is initialized with the provided encryption key. Then, the data is encrypted using the cipher and a nonce. Once the data is encrypted, an authentication tag is also generated. An authentication tag is important since it ensures the integrity of the encrypted data. This tag is then added to the encrypted data to form the final encrypted data with the tag. Finally, the function returns the encrypted data with a tag. Thus, using ChaCha20-Poly1305 instead of ChaCha20 alone ensures not only confidentiality but also both confidentiality and integrity of the data.

Algorithm 4 Encryption using ChaCha20-Poly1305

Input: key, data, and nonce
Output: encrypted_data_with_tag

1. Initialize ChaCha20-Poly1305:
cipher ← InitializeChaCha20Poly1305(key)
2. Encrypt the plaintext and generate the authentication tag:
encrypted_data, auth_tag ← EncryptAndAuthenticate(cipher, nonce, data)
3. Add the authentication tag to the encrypted data:
encrypted_data_with_tag ← Concatenate(encrypted_data, auth_tag)
4. Return encrypted_data_with_tag

5) Using randomized LSB for embedding: Implementation of only LSB instead of randomized LSB is not secure. This is because an attacker can use steganography analysis techniques to detect hidden data. Since the LSB is modified in a predictable manner, patterns may easily be detected. This, in

turn, makes it easier for an attacker to extract the hidden information. Therefore, randomized LSB with seed is implemented in this work. The algorithm of implementation is demonstrated in Algorithm 5. First, the input image is loaded and processed in RGB format. This means that each pixel consists of three color channels: red, green, and blue. After the image is converted into a pixel array. Next, the input data is converted into binary form. This ensures that data can be embedded as individual bits within the image pixels. This step is important since the LSB technique works at the bit level. To add randomization to the system, SFC64 as a PRNG function is initialized using the given seed. The PRNG function generates a randomized sequence of pixel positions. This ensures that data is embedded in an unpredictable order. The use of a seed means that the same random sequence can be reproduced later during extraction. The algorithm then goes through the randomly chosen pixel positions one by one, embedding a single bit of binary data into the LSB of the corresponding pixel channel. This process repeats until every bit of data has been embedded. The modified pixel array is then converted back into an image. Finally, modified image is saved.

Algorithm 5 Using randomized LSB for embedding

Input: image_path, output_image_path, data, seed
Output: stego_image

1. Load the image and convert to array format:
pixels ← LoadImageAsArray(image_path)
2. Convert data into binary representation:
binary_data ← ConvertToBinary(data)
3. Initialize pseudorandom generator with seed and shuffle pixel positions:
indices ← GenerateRandomizedPixelIndices(pixels, seed)
4. Embed binary data into least significant bits of pixel values:
4.1 data_index ← 0
4.2 **for each** (i, j) **in** indices **while** data_index < Len(binary_data) **do**
ModifyLSB(pixels[i, j], binary_data[data_index])
data_index ← data_index + 1
end for
5. Convert to image form:
stego_image ← ConvertToImageForm(pixels)
6. Save the modified image:
SaveImage(stego_image, output_image_path)

6) Using randomized LSB for extraction: During the extraction of data, the first step is loading the stego image and converting it into a pixel array as demonsrated in Algorithm 6. Then SFC64 as PRNG is initialized. Using the same seed allows the generation of a randomized sequence of pixel positions identical to the one used during embedding. This ensures that data is extracted in the same order as it was previously embedded. Next, the length prefix is extracted. The length prefix determines the exact length of the hidden message to ensure the correct amount of data is read. After determining the message length, the function continues to extract the encrypted message and nonce from the randomized pixel sequence. The process continues until the expected number of bits has been collected. Finally, the binary data is converted into bytes and returned as the extracted data, preserving the original encoding of the hidden information. This completes the LSB extarction process.

Algorithm 6 Using randomized LSB for extraction

Input: stego_image_path, seed
Output: extracted_data

1. Load the image and convert to array format:
pixels ← LoadImageAsArray(stego_image_path)
2. Initialize pseudorandom generator with seed and shuffle pixel positions:
indices ← GenerateRandomizedPixelIndices(pixels, seed)
3. Extract the length prefix:
message_length ← ExtractLength(pixels, indices)
4 Extract the actual encrypted message:
4.1 Initialize_Empty(binary_data)
4.2 data_index ← 0
4.3 total_bits ← ComputeTotalBits(message_length)
4.4 **for each** (i, j) **in** indices **while** data_index < total_bits **do**
binary_data ← binary_data + ExtractLSB(pixels[i, j])
data_index ← data_index + 1
end for
5. Convert binary data:
extracted_data ← ConvertBinaryToBytes(binary_data)
6. Return extracted_data

7) Decryption using ChaCha20-Poly1305: During the decryption process, the first step is to initialize the ChaCha20-Poly1305 cipher using a key (Algorithm 7). In the next step, the authentication tag and the encrypted data are parsed from the encrypted data with the tag. To ensure that the data is not modified, the authentication tag is verified. If the verification fails, an invalid tag error is raised, indicating possible corruption or modification. As a result, the decryption process is terminated and does not continue. If authenticated, however, the data is decrypted using the ChaCha20 cipher and the nonce. Finally, the original data is returned.

Algorithm 7 Decryption using ChaCha20-Poly1305

Input: key, encrypted_data_with_tag, nonce
Output: decrypted_data

1. Initialize ChaCha20-Poly1305:
cipher ← InitializeChaCha20Poly1305(key)
2. Parse encrypted_data and auth_tag:
encrypted_data, auth_tag ← Parse(encrypted_data_with_tag)
3. Verify the authenticity of the encrypted information:
if VerifyAuthTag(cipher, nonce, encrypted_data, auth_tag) **is false**:
throw InvalidTag
4. Decrypt the data using the cipher:
decrypted_data ← decrypt(cipher, nonce, encrypted_data)
5. Return the decrypted_data

8) Decryption using polyalphabetic cipher: Polyalphabetic decryption follows the same structure as encryption. However, it uses the reverse transformation to recover the original data. As seen in Algorithm 8, the first step is to initialize the random generator (similar to polyalphabetic encryption). Since the PRNG produces the same sequence of random values using the same key, the decryption process selects the

same sequence of blocks that were used during encryption. Next, the algorithm processes each character in the encrypted text. If the character is part of the defined character set, then the corresponding reverse mapping block is retrieved based on the sequence generated by the PRNG. This ensures that each character is restored to its original form. If the character is not in the defined set then it remains unchanged.

Algorithm 8 Decryption using polyalphabetic cipher

Input: ciphertext, seed, characters, reverse_blocks
Output: plaintext

1. Initialize an empty list to plaintext:
plaintext ← []
2. Initialize random generator rng with seed:
rng ← InitializeRNG(seed)
3. Process every character and replace it with its counterpart from reverse blocks:
3.1 **for each** char **in** ciphertext:
3.2 **if** char **in** characters:
block ← GetRandomBlock(rng, reverse_blocks)
plaintext.Add(block[char])
else:
plaintext.Add(char)
end if
end for
4. Return plaintext

B. Operational Workflow of the Proposed System

In this section, we describe the operational workflow of the proposed system. Thus, this section presents how the proposed algorithms can be implemented and executed in practice. While the previous section describes the algorithms individually, this section however focuses on their integration, data flow, and execution within the system.

1) UAV: The all secure embedding process takes action in UAV as described in Fig. 2. Initially, operators of UAV initialize 3 shared keys and one secret constant in UAV as well as in GCS. The substitution key (key 1) is used to derive a seed for polyalphabetic cipher to randomize blocks. Stream cipher key (key 2) is used for Chacha20-Poly1305 encryption and decryption. Stego key (key 3) is used to derive seed for randomized LSB image steganography. Finally, the secret constant is used by the proposed seed derivation function. After initializing keys and secret constant, a set of characters is created that includes uppercase letters (A to Z), lowercase letters (a to z), digits (0 to 9), and some common special symbols (,.-_/!@#$%^&*()). Choosing characters depends on flight controller. In most cases, flight controllers use those characters to record flight data. After initialization of those characters, they are passed to Algorithm 1. Algorithm 1 then initializes blocks for polyalphabetic cipher encryption. Once the initialization step is done, the UAV starts its mission and waits for command from the GCS. When a command is received, the UAV captures an aerial image using the onboard camera. The UAV's flight computer also requests the flight controller to get necessary flight data. The type of flight data required depends on the application. It may include sensor readings, power statues, telemetry data and others. If the flight data includes location information and the channel in which the image is sent, is not secure, then the image itself can reveal the position of the UAV. To solve this problem, the image can be sent using a secure channel, or instead of the actual aerial image, a previously stored decoy image can be sent. Next, current date and time information is obtained, which includes the following sequence: year, month, day, hour, minute, second, and millisecond. It is important to note that obtained date and time information are used multiple times in each secure embedding process (each cycle). Therefore, they are kept until the next secure embedding process. After obtaining date and time information, Algorithm 2 is implemented to generate a seed for polyalphabetic cipher. To achieve this, the substitution key, secret constant, and the obtained date and time information are used. The generated seed is then passed to Algorithm 3 along with flight data (in text form), defined characters, and initialized blocks. As a result, the initial encryption process is finalized. However, as mentioned earlier, only substitution encryption may not be safe. That's why, the outcome of Algorithm 3 is then passed to Algorithm 4 for furthermore encryption using ChaCha20-Poly1305. Algorithm 4 also requires both a key and a nonce to operate. Therefore, a stream cipher key is provided as the key, while a 12-byte nonce is generated using a CSPRNG and passed to Algorithm 4. Algorithm 4 then encrypts the data, ensuring both its confidentiality and integrity. The next step is to hide the encrypted data in an image. To achieve this, the structure of the message must first be established. This structure consists of a length prefix, a nonce, and the encrypted data. The length prefix is necessary because the length must be known during extraction. Additionally, the nonce is included in the hidden message for later ChaCha20-Poly1305 decryption. It is important to note that, in most cases, a nonce should be unique rather than secure. However, in this work, it is still included in the hidden message to provide an additional layer of security for nonce. After creating a hidden message structure, Algorithm 2 is implemented again to generate a seed for image steganography, however, this time stego key is used. Once the seed is generated, it is then used by Algorithm 5 to embed a hidden message in the image. The image format in this work is selected as PNG. This is because PNG supports lossless compression as well as metadata. After using Algorithm 5, the stego image is created with the hidden message inside. In the final step, time and date information are embedded in the metadata of stego image. This step is important since the GCS will use this metadata to derive seed using keys and secret constant. Once the secure embedding process completes, the image is sent to GCS.

2) GCS: Similar to UAV, GCS also begins its operation by initialization. First, 3 same shared keys and secret constant are initialized. Then, the same set of characters is created. After initialization of those characters, they are passed to Algorithm 1. However, this time reverse blocks are returned instead of blocks for polyalphabetic cipher decryption. Those steps complete the GCS initialization process.

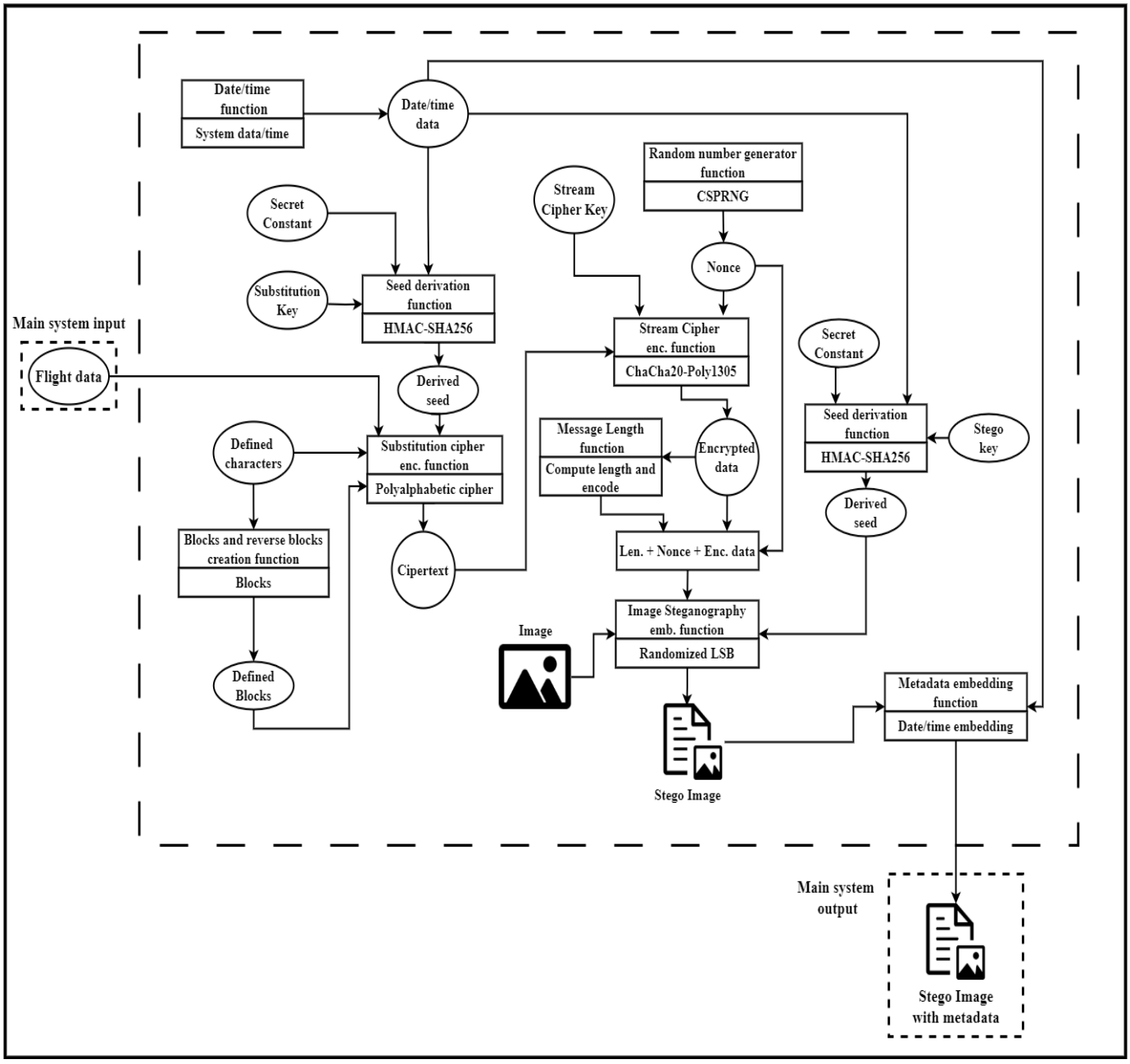

Fig. 2. Operational workflow for UAV.

When the GCS sends a command to the UAV and receives a stego image, it begins the reversing process to retrieve the hidden message (Fig. 3). To accomplish this, the date and time information from the stego image's metadata is first extracted. Next, the date and time information, along with the stego key and secret constant, is used to derive the seed by implementing Algorithm 2. This seed is then used by Algorithm 6 to extract the hidden message from the image. Since the message length and structure are known, nonce along with encrypted data are separated. Then for ChaCha20-Poly1305 decryption, Algorithm 7 is provided with encrypted data, stream cipher key, and nonce. Once ChaCha20-Poly1305 decryption is done, the ciphertext is produced. To decrypt the ciphertext, Algorithm 2 is implemented using the substitution key, along with the date and time information and the secret constant, to derive the seed. Finally, Algorithm 8 is implemented to decrypt ciphertext using seed, defined characters, and previously initialized reverse blocks. Decrypted ciphertext reveals the plaintext which includes important flight data.

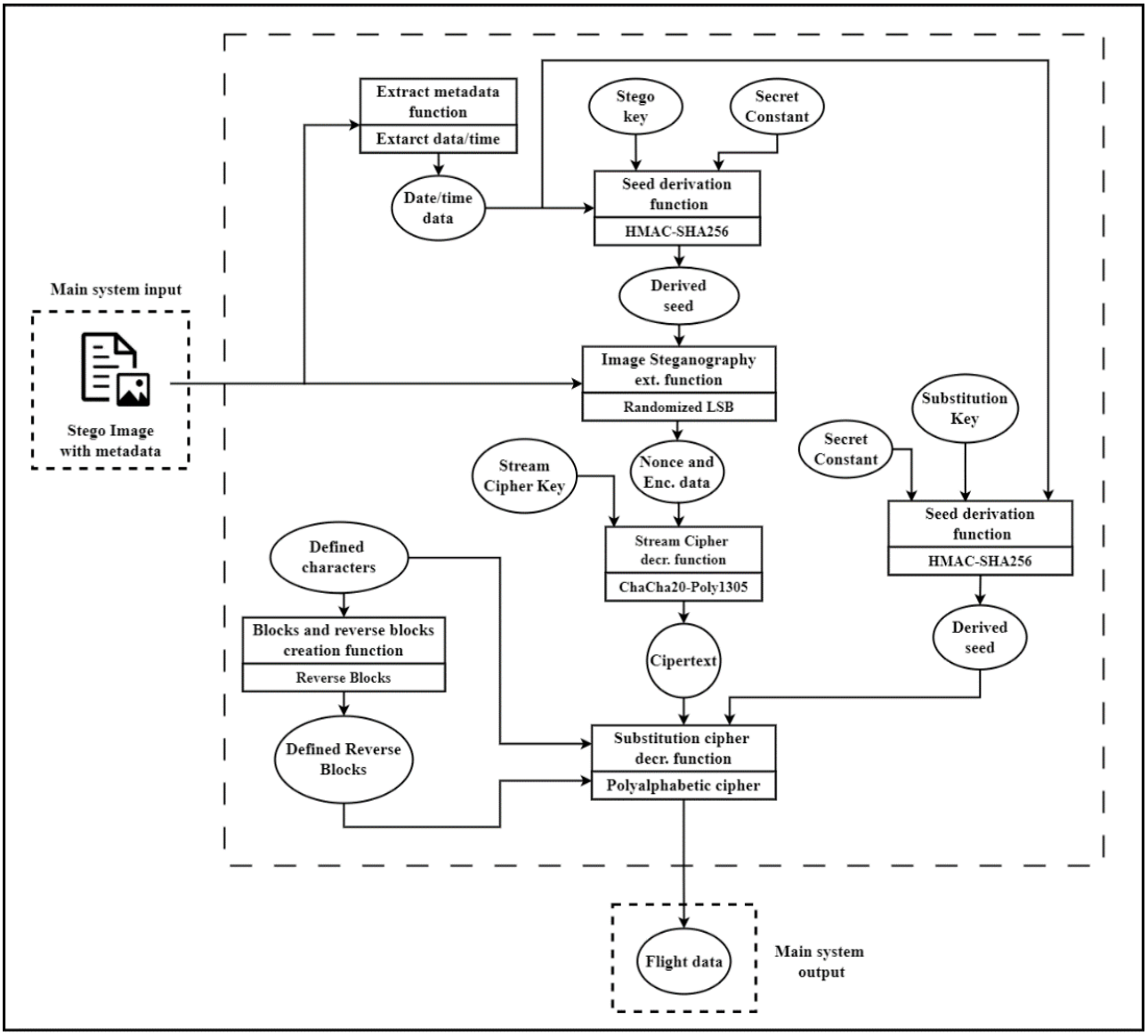

Fig. 3. Operational workflow for GCS.

V. The Experiments

In this section, the experimental setup, evaluation metrics, results of various experiments, and a comparison of the proposed work with other studies are presented.

A. Experimental Setup

The proposed approach was implemented in a real-world environment using a UAV-GCS system rather than a simulated setup. This ensures that the results accurately reflect real-world conditions. The main components used in this work are given below:

1) Flight controller. The flight controller used in this work is Pixhawk 2.4.8. Pixhawk 2.4.8 has 32-bit STM32F427 Cortex-M4 processor. It is equipped with 256KB RAM, 2MB flash memory, I/O ports, and various sensors. The controller runs ArduPilot firmware.

2) Flight computer. The secure embedding process is carried out in Raspberry Pi 3 Model B+ acting as a flight computer. Raspberry Pi 3 Model B+ has a Broadcom BCM2837B0 processor, which is a quad-core Cortex-A53 (ARMv8) 64-bit SoC running at 1.4GHz. It comes with 1GB of LPDDR2 SDRAM. The device's operating system is Raspberry Pi OS (previously referred to as Raspbian). As a programming language, Python 3.8 is used.

3) Protocol. The communication between the flight controller and the flight computer is established using a special protocol, MAVLink.

4) GCS computer. The entire extraction process is carried out on a Lenovo Ideapad 330, which is used as the GCS computer in the experiments. The computer is equipped with an Intel Core i7 8550U processor running at 1.80 GHz, and 16GB of DDR4 DRAM. The operating system is Windows 10. Python 3.8 is used as the programming language.

Fig. 4 demonstrates the connection of the UAV's flight controller, flight computer, and camera module.

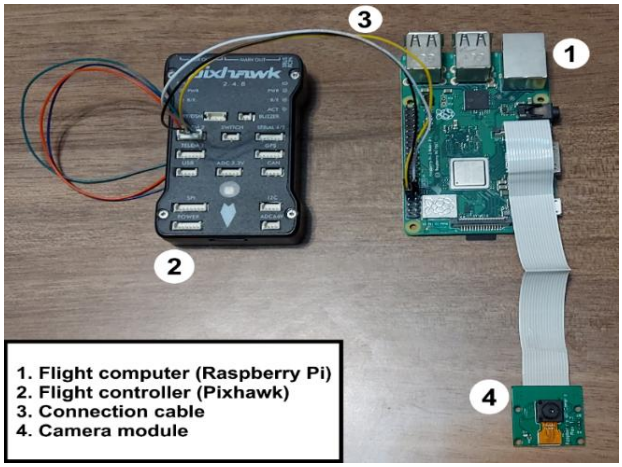

Fig. 4. Connection of UAV's components.

B. Evaluation Metrics

To assess the performance of the proposed method, various evaluation metrics are used in this work including Mean Squared Error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index Measure (SSIM). MSE measures the average squared difference between the original and modified images. A lower value of MSE indicates better image quality. MSE can be calculated using Eq. (1).

$$MSE = \frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}[R(i,j) - C(i,j)]^2 \quad (1)$$

where, M, N are the dimensions of the image, R(i,j) is the original image pixel, C(i,j) is the modified image pixel. PSNR measures the quality of the modified image compared to the original one. A higher value of PSNR indicates better image quality. PSNR can be calculated using Eq. (2).

$$PSNR = 10\log_{10}\left(\frac{MAX_I^2}{MSE}\right) \quad (2)$$

where, MAX_I is the maximum possible pixel value of the image. SSIM is used to measure the structural similarity between two images. A higher value of SSIM indicates better image quality. SSIM can be calculated using Eq. (3).

$$SIM\ (i,j) = \frac{(2\mu_i\mu_j + C_1)(2\sigma_{ij} + C_2)}{(\mu_i^2 + \mu_j^2 + C_1)(\sigma_i^2 + \sigma_j^2 + C_2)} \quad (3)$$

where, μ_i,μ_j are the mean values of images i and j, C_1,C_2 are constants, σ_i σ_j are variances, and σ_{ij} is covariance.

C. Results

In systems such as UAVs, processing time is a critical factor that must be considered. Table I demonstrates the average encryption (UAV) and decryption (GCS) duration of the main cryptographic algorithms depending on the data size. As seen from the results, ChaCha20-Poly1305 outperforms the others during the encryption process. Moreover, while other cryptographic algorithms only perform encryption, ChaCha20-Poly1305 handles both encryption and MAC calculation for data integrity. Even while performing this additional task, it still outperforms other algorithms. However, during the decryption phase, AES has a slightly faster decryption time when the data size is larger. This is understandable since, as mentioned earlier in this study, most processors support hardware acceleration for AES. Because the GCS (laptop) used in this work supports hardware acceleration, AES may occasionally achieve better performance than the other algorithms.

TABLE I. PERFORMANCE OF MAIN CRYPTOGRAPHIC ALGORITHMS

Main cryptographic algorithms	Data size (bytes)	Encryption duration (milliseconds)	Decryption duration (milliseconds)
ChaCha20-Poly1305	100	1.201	0.118
	8000	1.279	0.271
AES	100	1.502	0.181
	8000	1.692	0.232
3DES	100	1.515	0.197
	8000	2.229	0.490

Table II demonstrates the steganographic performance metrics of the proposed method. For a comprehensive evaluation, results are presented for a variety of capacities, ranging from 100 bytes to 8000 bytes. This ensures an evaluation of the performance of the suggested method at both low and high embedding rates. Even at 8000 bytes, the method maintains high performance, demonstrating its effectiveness in preserving image quality while embedding larger amounts of data.

TABLE II. STEGANOGRAPHIC PERFORMANCE METRICS FOR DIFFERENT CAPACITIES

Capacity (bytes)	MSE	PSNR	SSIM
100	0.0006637	79.9107	0.9999
500	0.0026791	73.8507	0.9999
1000	0.0052515	70.9279	0.9999
2000	0.0103632	67.9758	0.9999
3000	0.0154202	66.2498	0.9999
4000	0.0203997	65.0345	0.9999
5000	0.0255533	64.0563	0.9998
8000	0.0409431	62.0089	0.9998

To evaluate the proposed system, it is also necessary to present the total time for the crypto-steganography process. Table III demonstrates the total average secure embedding duration and the total average secure extraction duration for different capacities.

TABLE III. TOTAL EMBEDDING AND EXTRACTION DURATIONS

Capacity (bytes)	Total secure embedding duration (seconds)	Total secure extraction duration (seconds)
100	1.758	0.182
500	1.799	0.185
1000	1.926	0.207
2000	2.093	0.238
3000	2.294	0.262
4000	2.624	0.298
5000	2.767	0.321
8000	3.138	0.429

Fig. 5 demonstrates a side-by-side comparison of the original and stego image captured and processed by the UAV's flight computer. As can be seen from the images, there are no visible differences.

Fig. 5. Side-by-side comparison of original (left) and stego image (right).

D. A Comparison of the Proposed Work with other Studies

A direct comparison between our proposed method and the works discussed in Section II of this study is challenging due to inconsistencies in the evaluation metrics. Some of these studies do not accurately mention necessary image quality metrics such as MSE, PSNR, or SSIM, which are important for comparison. Moreover, these metrics are affected by various factors, including image dimensions and the amount of information embedded in the image. However, these factors vary across studies. Furthermore, most of the works analyzed either use weak cryptographic techniques or do not use encryption at all before using steganography. As a result, they rely only on the steganographic level of security. This is insufficient from a security perspective. Some of the works analyzed are as follows. Syed et al. [10] achieved an MSE of 0.001 and SSIM of 0.97 using the LSB-XOR technique. According to the study, the security only relies on the steganography level and no encryption was used beforehand. As a result, this approach may lack security. Rostam et al. [14] achieved a PSNR value above 45, an SSIM value above 0.98 and MSE value less than 1.02 using chaotic LSB steganography with block-based embedding. The system's security relies entirely on chaotic functions. While chaotic systems can provide some level of security, they do not offer the same cryptographic guarantees as encryption algorithms. The system mainly hides data rather than encrypting it. Thus, the hidden data may be extracted if the technique is discovered. Alarood et al. [11] achieved a PSNR value of 66.61 and an SSIM value of 0.9998 using a pixel classification-based spatial domain. Similar to other works, no additional encryption method is used in their work. Also, the study states that it cannot work as a real-time system.

In contrast to other works, our approach ensures security at every level. It uses two different encryption techniques to increase resilience before implementing steganography. Most importantly, even though the keys are static, the seed derivation algorithm makes the system dynamic. Furthermore, even with a high data capacity, the proposed approach maintains strong steganographic performance. As a result, the system becomes difficult to analyze and exploit.

VI. Conclusion and Future Work

In this work, a multilayer protection method is proposed to secure important UAV flight data. The main idea behind this approach is to make it as difficult and resource-intensive as possible for an attacker to succeed. By adding multiple layers of security, attackers are forced to waste their time and resources. Additionally, even if one layer is bypassed, the others remain active to protect the data. Moreover, various experiments are conducted using real hardware. The results of these experiments demonstrate that the proposed work is better both from the point of security and performance. It's important to mention that older and slower version of the flight computer was intentionally chosen for our experiments to test how our system would perform on it. Even with earlier hardware versions, the entire process took only a few seconds to complete. It will be significantly faster on newer versions of the flight computer.

In the future, this study will be extended by employing other cryptographic and stenographic methods. Additionally, while this study focuses on hiding text-type information within images, future research could explore other steganographic methods. For instance, it could involve hiding sensitive images within images or within transmitted video, etc. Furthermore, future studies might examine more adaptable techniques that change according to the type of data being sent.

Acknowledgment

This work was supported by the Azerbaijan Science Foundation-Grant № AEF-MCG-2023- 1(43)-13/04/1-M-04.

References

[1] F.J. Abdullayeva, "Cybersecurity issues of some class unmanned aerial vehicle systems: A survey", in NATO Science for Peace and Security Series – D: Information and Communication Security. IOS press, vol. 62, pp. 31-39, 2022.

[2] F. Abdullayeva and O. Valikhanli, "A survey on UAVs security issues: attack modeling, security aspects, countermeasures, open issues", Control Cybern., vol. 52, no. 4, pp. 405–439, 2023.

[3] C. G. L. Krishna and R. R. Murphy, "A review on cybersecurity vulnerabilities for unmanned aerial vehicles", in 2017 IEEE Int. Symp. Saf., Secur. Rescue Robot. (SSRR), Shanghai, China, pp. 194-199, Oct. 11–13, 2017.

[4] A. Y. Javaid, W. Sun, V. K. Devabhaktuni, and M. Alam, "Cyber security threat analysis and modeling of an unmanned aerial vehicle system", in 2012 IEEE Int. Conf. Technol. Homeland Secur. (HST), Waltham, MA, USA, pp. 585-590, Nov. 13–15, 2012.

[5] N. Shachtman, "Exclusive: Computer virus hits U.S. drone fleet", Wired, 2011. Available at: https://www.wired.com/2011/10/virus-hits-drone-fleet/ [Accessed: 09 January 2025].

[6] M. Hooper et al., "Securing commercial WiFi-based UAVs from common security attacks", in MILCOM 2016 - 2016 IEEE Mil. Commun. Conf. (MILCOM), Baltimore, MD, USA, pp. 1-6, Nov. 1–3, 2016.

[7] A. B. Alkodre et al., "A shuffling-steganography algorithm to protect data of drone applications", Comput., Mater. and Continua, vol. 81, no. 2, pp. 2727–2751, 2024.

[8] Y.-I. Lin, Y.-H. Huang, and C.-C. Chen, "An Effective Dual-Image Reversible Hiding for UAV's Image Communication", Symmetry, vol. 10, no. 7, p. 271, 2018.

[9] J. E. Rodríguez Marco, M. Sánchez Rubio, J. J. Martínez Herráiz, R. González Armengod, and J. C. P. Del Pino, "Contributions to Image Transmission in Icing Conditions on Unmanned Aerial Vehicles", Drones, vol. 7, no. 9, p. 571, 2023.

[10] F. Syed, S. H. Alsamhi, S. K. Gupta, and A. Saif, "LSB - XOR technique for securing captured images from disaster by UAVs in B5G networks", Concurrency Computation: Pract. Experience, vol. 36, no. 12, pp. 1-13, 2024.

[11] A. Alarood, N. Ababneh, M. Al-Khasawneh, M. Rawashdeh, and M. Al-Omari, "IoTSteg: ensuring privacy and authenticity in internet of things networks using weighted pixels classification based image steganography", Cluster Comput., vol. 25. no. 3. pp. 1607–1618, 2021.

[12] M. Hassaballah, M. A. Hameed, A. I. Awad, and K. Muhammad, "A Novel Image Steganography Method for Industrial Internet of Things Security", IEEE Trans. Ind. Inform., vol. 17, no. 11, pp. 7743–7751, 2021.

[13] H. N. AlEisa, "Data Confidentiality in Healthcare Monitoring Systems Based on Image Steganography to Improve the Exchange of Patient Information Using the Internet of Things", J. Healthcare Eng., vol. 2022, pp. 1–11, 2022.

[14] H. E. Rostam, H. Motameni, and R. Enayatifar, "Privacy-preserving in the Internet of Things based on steganography and chaotic functions", Optik, vol. 258, pp. 1-15, 2022.

[15] R. Alguliyev and Y. Imamverdiyev, Kriptoqrafiyanın əsasları [Fundamentals of cryptography], Baku, Azerbaijan: İnfor. Texno., 2006. (in Azerbaijani)

[16] C. Paar and J. Pelzl, Understanding Cryptography a Textbook for Students and Practitioners. Berlin, Heidelberg: Sprin. Berlin Heidel., 2010.

[17] D. J. Bernstein, "ChaCha, a variant of Salsa20," in Workshop record of SASC, pp. 3–5, 2008.

[18] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC Editor, p. 46, 2018.

[19] A. J. Menezes, S. A. Vanstone, and P. C. Van Oorschot, Handbook of App. Crypt. CRC Press, 1997.

[20] J. Fridrich, Steganography in Digital Media: Principles, algorithms, and applications. Cambridge Univ. Press, 2010.