Cryptography with Ordinary Differential Equations

Mehul Maheshwari
CS/MATH 514

1 Introduction

Cryptography, the science of secure communication, is crucial in safeguarding sensitive information in our increasingly connected world. As technology advances, so do the techniques and algorithms used to break down encryption and access sensitive information. Traditional encryption algorithms use pseudorandom number generators (PRNGs) to generate encryption keys. However, PRNGs have deterministic and predictable behavior, making them vulnerable to cryptographic attacks.

One emerging field within cryptography to protect data from unauthorized access is the utilization of Ordinary Differential Equations (ODEs) as a foundation for encryption algorithms. This approach takes advantage of the unpredictable behavior of chaotic ODEs, offering several advantages:

1) It provides a higher level of security by leveraging the inherent chaotic behavior of dynamical systems. This chaotic behavior introduces a high degree of randomness and complexity into the encryption process, making it more resistant to cryptographic attacks.
2) The sensitivity to initial conditions in chaotic systems ensures that small perturbations in the system's parameters or seed values lead to significantly different encryption keys. This property provides forward secrecy, as compromising one key does not compromise the security of other encrypted data.
3) Using chaotic ODEs expands the range of encryption algorithms available, offering novel and diverse approaches to secure communication.

Existing research in this field has taken steps toward providing insights into utilizing a chaotic ODE for encryption purposes, highlighting the potential and challenges of this emerging field. Notably, "Chaotic Systems for Encryption: A Survey" by Khan et al. (2013) provides an overview of various chaotic systems used for encryption. It helps break down the benefits of chaotic ODEs and discusses their potential for cryptographic applications. "Chaotic Differential Equations for Secure Communication" by Ibragimov et al. (2012) proposes a chaotic ODE-based encryption scheme and evaluates its performance and security. "A Secure Image Encryption Scheme Based on Chaotic Systems" by Mishra et al. (2018) then builds upon the findings, presenting an image encryption algorithm utilizing chaotic ODE systems for generating encryption keys and performing encryption and decryption operations on digital images.

Our objective in this paper is to combat the problem of deterministic PRNGs for encryption by creating a chaotic ODE encryptor. In this process, we will take an analytical lens into the different considerations of such an encryptor. Then, having completed an encryptor, we will conduct mathematical analyses of the results to determine the efficacy of the encryptor.

2 Method

When coding a chaotic ODE encryption and decryption system, several development considerations must be considered to ensure its effectiveness, security, and computational drain. Here is a breakdown of my considerations for creating the encryptor.

#1 Choice of Chaotic ODE Model

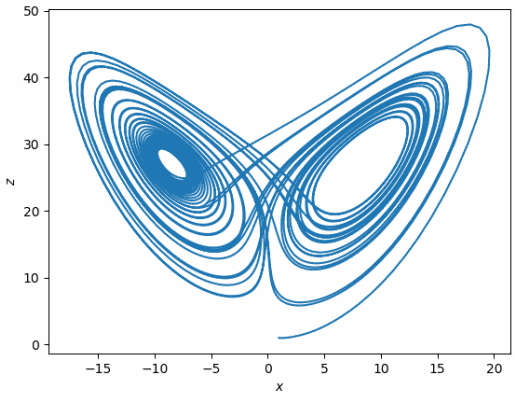
By researching already existing ODE encryptors, I created a set of metrics to determine which ODE model to choose. I found that two systems were used more commonly than the rest, the Lorenz system and the Chen system. Comparing these two systems against the following metrics helped me decide which model to choose.

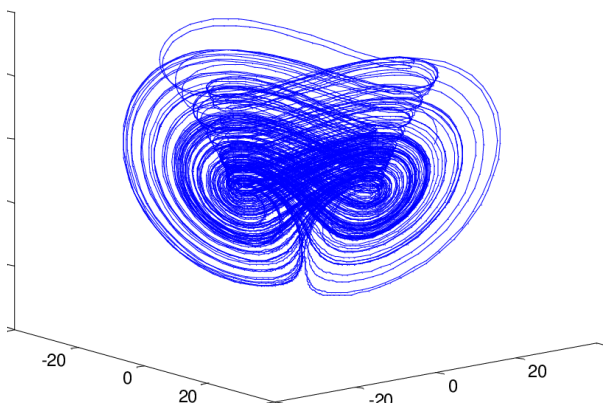
Lorenz System	**Chen System**
$dx/dt = \sigma(y - z)$	$dx/dt = -(abx)/(a + b) - yz + c$
$dy/dt = x(\rho - z) - y$	$dy/dt = ay + xz$
$dz/dt = xy - \beta z$	$dz/dt = by + xy$

Fig. 1 - Mathematical representation of the Lorenz and Chen ODE systems

Metric 1: Chaotic Behavior

The Lorenz and Chen models both possess essential characteristics that would be useful for encryption, such as being highly sensitive to initial conditions and complex dynamics, as visualized in Figure 2.





Lorenz Model -
https://www.johndcook.com/lorenz_xy.png

Chen Model -
https://www.researchgate.net/profile/Saad-Al-Azzawi-3/publication/287583778/figure/fig1/AS:651924361707522@1532442288518/The-attractor-of-Chen-system.png

Fig. 2 - Visual Representations of both ODE systems

Metric 2: Dimensionality

With the goal of encryption in mind, a system with greater dimensions introduces more randomness and less predictability. Although taking in different parameter values, the Lorenz and the Chen models were three-dimensional systems, making them both good choices.

Metric 3: Key Space

This metric measures the range created by both methods for possible encryption keys. I noticed that both methods offered a wide range of key space, which can also be visualized by Figure 2.

Metric 4: Computational Efficiency

Both methods' computation efficiency depends on the number of differential equations, which is tied for both methods at three. Furthermore, although the system dynamics may change the algorithm's complexity on a run-by-run basis since it is determined by the initial conditions we want to keep variable, this metric is also inconclusive.

Metric 5: Existing Security Analysis

The Lorenz model has been widely studied [1, 2], whereas the Chen system, on the other hand, has less available literature. Because of its widespread usage, numerous research studies examine its security properties. In contrast, the Chen model does not have that kind of extensive research, so its strengths and weaknesses in terms of security still need to be well established [4].

From these metrics it is difficult to distinguish the two chaotic ODE models as they both perform well. However due to the existing security analysis of the Lorenz system, I ultimately decided to move forward with it over the Chen system.

#2 Key Generation

When generating a key for encryption, several considerations must be made. Determining the seed values should be done to promote randomness and unpredictability. Key generation will also require a time span to determine the granularity of the ODE integration. Deciding upon a time span and step that does not overload the computation is necessary for efficiency.

#3 Encryption Algorithm

Once a key has been generated, a combination process is necessary with the unencrypted message. The encryption algorithm must utilize the Lorenz system and perform some operation with it. Considerations of what to choose must keep in mind efficiency to not overload the number of arithmetic operations needed to be computed. I found that common practices included bitwise operations, modular arithmetic, or XOR operations.

#4 Decryption Algorithm

Once encrypted, a decryption algorithm corresponding to the encryption cipher is needed. The decryption algorithm must utilize the key values to recover the original unencrypted text. The operation consideration from the encryption algorithm also plays into the decryption algorithm's complexity. Because of this, I decided upon the bitwise XOR operation for encryption/decryption to combine the message with the key.

#5: Security Analysis

Once we have the basis for our encryptor, we need different ways to measure the algorithm's success. To this end, I divided the security considerations into three categories.

Metric 1: Proper Encryption / Decryption

This metric assesses whether or not we can predict the encrypted message based on the original message without knowing the key. Although this metric is inherently more qualitative, I compare the original message to the encrypted message using the Levenshtein distance to get quantitative data. This distance mathematically represents the distance between two strings by checking the minimum number of single-character edits required to get from one string to another. This algorithm is represented in Figure 3.

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise} \end{cases}$$

Fig. 3 - Representation of how the Levenshtein distance is calculated

Because we are interested in checking the similarity between the two strings and not the number of edits, we perform the following calculations to get us a numerical value from zero to one where one represents a complete match, and zero represents a complete difference.

$$Similarity = 1 - \frac{lev(encryption, original)}{max\{length(original), length(encryption)\}}$$

Fig. 4 - Mathematical formula for the similarity value between the encryption and original

Metric 2: Sensitivity to the key

To tell whether we have improved from the traditional PRNGs, we need a metric to calculate whether small key changes create significant encryption changes. Inherently, PRNGs are bad at changing the outcome through perturbation because of their deterministic nature. Gauging small key changes' differences will help us understand whether the chaotic systems are improving cryptography. To evaluate this metric, we will compare the similarity, as shown in Figure 4, between the encrypted message and a message re-encrypted by small changes to the key.

With these, we have both quantitative and qualitative metrics for providing analysis about the effectiveness of the encryption method.

3 Code

Using all of the considerations as explained from the methods section, I created the following methods and functions in python.

ODE Model

This function defines a Lorenz system of equations with the appropriate parameters. It follows the same mathematical implementation as shown in Figure 1.

```python
# Consideration 1: Choice of Chaotic ODE Model
def lorenz_equations(t, u, sigma, rho, beta):
    x, y, z = u
    dxdt = sigma * (y - x)
    dydt = x * (rho - z) - y
    dzdt = x * y - beta * z
    return [dxdt, dydt, dzdt]
```

Fig. 5 - Code for generating Lorenz Equation

Key Generation

This function's role is to generate a key for the encryption process as explained in the methods section. Note that the x, y, and z coefficients are all randomly generated to promote variability. The key is generated by solving the chaotic ODE system, and evaluating the points as specified by the time span and step.

```python
# Consideration 2: Key Generation
def generate_key(seed, sigma, rho, beta, t_start, t_end, t_step):
    np.random.seed(seed)
    x0 = np.random.uniform(-20, 20)
    y0 = np.random.uniform(-20, 20)
    z0 = np.random.uniform(0, 40)
    u0 = [x0, y0, z0]

    t_span = (t_start, t_end)
    t_eval = np.arange(t_start, t_end, t_step)

    sol = solve_ivp(
        lorenz_equations,
        t_span,
        u0,
        args=(sigma, rho, beta),
        dense_output=True
    )

    key = sol.sol(t_eval)
    return key
```

Fig. 6 - Code for generating a key

Encryption Algorithm

This function is responsible for encrypting the algorithm with the same considerations as specified in the methods section such as making use of the bitwise XOR operation for cost effectiveness.

```python
# Consideration 3: Encryption Algorithm
def encrypt(message, key):
    tempKey = np.array(key)
    encrypted = ""
    key_len = len(key[0])

    for i, char in enumerate(message):
        key_idx = i % key_len
        key_value = tempKey[:, key_idx]

        # Use bitwise XOR operation for efficiency
        encrypted_char = chr(ord(char) ^ int(key_value.sum()) % 256)
        encrypted += encrypted_char

    return encrypted
```

Fig. 7 - Code for encrypting a message

Decryption Algorithm

This function decrypts the encrypted message in a very similar manner, taking advantage of the knowledge of the key.

```python
# Consideration 4: Decryption Algorithm
def decrypt(encrypted, key):
    decrypted = ""
    key_len = len(key[0])

    for i, char in enumerate(encrypted):
        key_idx = i % key_len
        key_value = key[:, key_idx]

        decrypted_char = chr(ord(char) ^ int(key_value.sum()) % 256)
        decrypted += decrypted_char

    return decrypted
```

Fig. 8 - Code for decrypting a message

Run Cryptography

This method is responsible for generating part of the output that can be used for testing and analysis and integrating the separate methods to create a cohesive cryptography program. It also takes advantage of the seed concept, in which repeating a seed will guarantee the same parameters as tested before, making separate runs of the program reproducible for testing.

```
# changing seed runs a new instance of cryptography. Keeping the same seed
will not change random numbers for reproducibility
def run_crypto(seed):
    # Parameters for the Lorenz system
    sigma = 10
    rho = 28
    beta = 8 / 3

    # Consideration 2: Key Generation
    t_start = 0
    t_end = 10
    t_step = 0.01
    key = generate_key(seed, sigma, rho, beta, t_start, t_end, t_step)

    # Consideration 3: Encryption
    message = "This is the message I want to encrypt"
    print("------------------------------")
    print("Original message:", message)
    encrypted_message = encrypt(message, key)
    print("Encrypted message:", encrypted_message)

    # Consideration 4: Decryption
    decrypted_message = decrypt(encrypted_message, key)
    print("Decrypted message:", decrypted_message)

    [similiarity, similiarity2] = analyze_encryption_effectiveness(message,
encrypted_message, key)

    # want to plot the differences between closeness of message and seed

    return [similiarity, similiarity2]
```

Fig. 9 - Code for running an iteration of the cryptography

Testing and Analysis

These functions are responsible for checking the security of our cryptography method and visualizing the quantitative data as specified in the methods section.

```
def analyze_encryption_effectiveness(message, encrypted_message, key):
    """
    Analyze the effectiveness of the encryption method
    """
    # Check minimum number of single character edits required to get from the
encrypted message to the original message

    divisor = float(max(len(message), len(encrypted_message)))
    numerator = float(Levenshtein.distance(message, encrypted_message))
    similarity = 1 - float(numerator / divisor)
```

```
    print(f"Similarity between the encrypted message and the original:
{similarity}")

    # Check sensitivity to the key by slightly modifying the key and
re-encrypting the message

    # The range will look like this: [-perturbation_range,
perturbation_range]
    perturbation_range = 2

    # Modify the key using the perturbations
    perturbations = np.random.uniform(-perturbation_range, perturbation_range,
size=(3, 1000))
    modified_key = key + perturbations

    modified_encrypted_message = encrypt(message, modified_key)

    divisor = float(max(len(encrypted_message),
len(modified_encrypted_message)))
    numerator = float(Levenshtein.distance(encrypted_message,
modified_encrypted_message))
    similarity2 = 1 - float(numerator / divisor)
    print(f"Modified encrypted message after slight perturbation is:
{modified_encrypted_message}")
    print("Similarity between messages with slight perturbation is:
{}".format(similarity2))
    return [similarity, similarity2]
```

Fig. 10 - Code for analyzing the cryptography

```
def graphAnalysis():
    """
    Graphing representations of numerical analysis
    """

    # variable definitions
    ITERATIONS = 200
    dataSet = {}

    i = 0
    while i < ITERATIONS:
        dataSet.update({i: run_crypto(i)})
        i = i + 1

    # Extract the keys and values from the dictionary
    keys = list(dataSet.keys())
    values = list(dataSet.values())

    # Extract value 1 and value 2 separately
    value1 = [v[0] for v in values]
    value2 = [v[1] for v in values]
```

```
# Generate the graph
plt.plot(keys, value1, label='Original vs Encryption')
plt.plot(keys, value2, label='Encryption vs Pertubation')
plt.xlabel('Iterations')
plt.ylabel('Difference')
plt.title('Encryption Differences per Iteration')
plt.legend()
plt.show()
print("_____")
print("Average similarity for original vs encryption", mean(value1))
print("Average similarity for perturbation vs encryption", mean(value2))
```

Fig. 11 - Code for graphing iterations of analysis

These methods working in conjunction provide us with the basis for creating our cryptography method utilizing chaotic ODEs, as well as testing the security and effectiveness of the cryptography.

4 Results

Recall that we are interested in two different qualitative metrics. The first measures the similarity between the original message and the encrypted message, and the second measures the similarity between the encrypted message and a new one modified by creating a slight perturbation to the key.

Running the full graphical analysis for two hundred iterations by encrypting the message “Encryption” which has a length of ten characters yielded the following results:

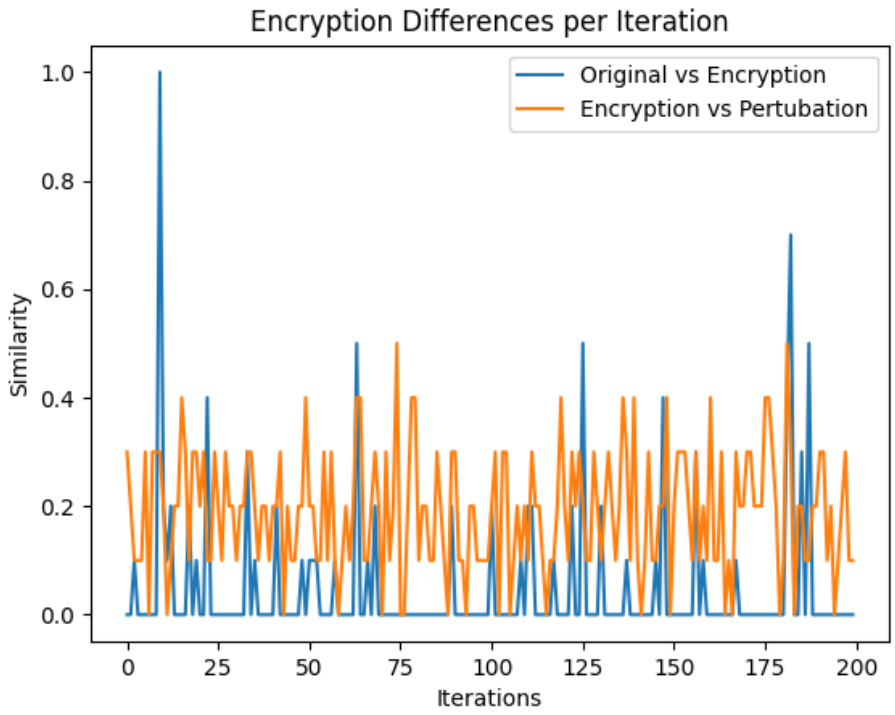


Fig. 12 - Output of analysis for message "Encryption"

```
Original message: Encryption
Encrypted message: lBL@LIHV-+
Original message: Here is my hidden message
Encrypted message: aI]WPO/<g !#"!/VRGCLMB
Decrypted message: Here is my hidden message
Similarity between the encrypted message and the original: 0.0
Modified encrypted message after slight perturbation is: bO_UQJ,?g/."#
,`V]@_OJB
Similarity between messages with slight perturbation is: 0.16000000000000003

Average similarity for original vs encryption 0.0356
Average similarity for perturbation vs encryption 0.19219999999999998
```

Fig. 13 - Last iteration of output and average values

Running the code with the message "Here is my hidden message" with the length of twenty five characters yielded the following output:

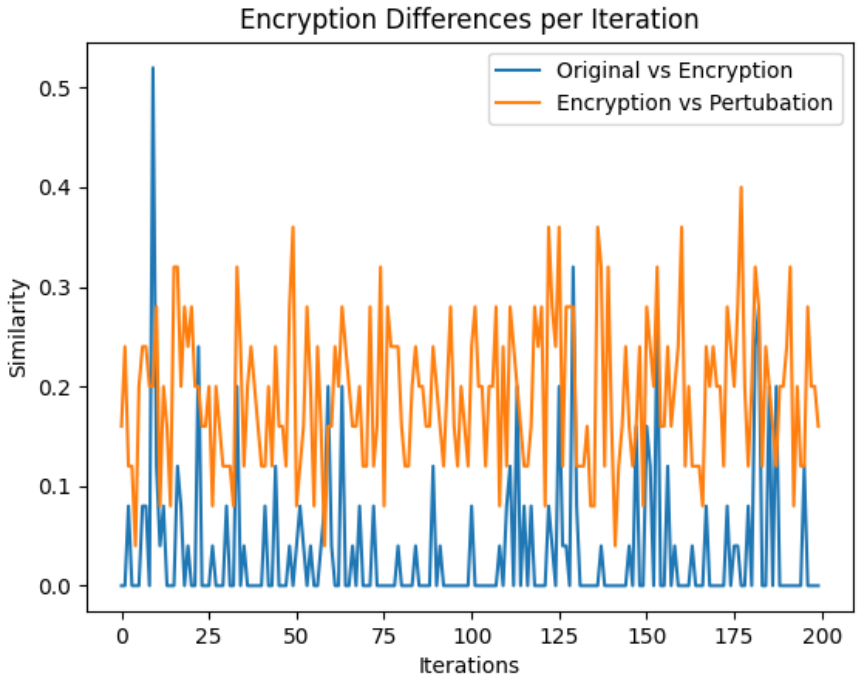


Fig. 14 - Output of analysis for message "Encryption"

```
Original message: Encryption
Encrypted message: lBL@LIHV-+
Decrypted message: Encryption
Similarity between the encrypted message and the original: 0.0
Modified encrypted message after slight perturbation is: oDNBMHMV.(
Similarity between messages with slight perturbation is: 0.09999999999999998
_____________________________________________________________
Average similarity for original vs encryption 0.0455
Average similarity for perturbation vs encryption 0.1885
```

Fig. 15 - Last iteration of output and average values

Continued outputs with different message sizes, as demonstrated by Figure 16, show that the size of the message does not significantly impact the security of the cryptography, which is good.

Characters Encrypted	10	25	35	45	75	405

Average Similarity Value of Original Message	0.0356	0.0455	0.0328571428571428 6	0.0314444444444444 4	0.0324666666666666 6	0.0380246913580246 9
Average Similarity Value of Perturbed Message	0.1921999999999999 8	0.1885	0.1951428571428571 2	0.1951111111111111 2	0.1925333333333333 3	0.1885308641975308 7

Fig 16 - Chart of analytical values compared against the characters encrypted

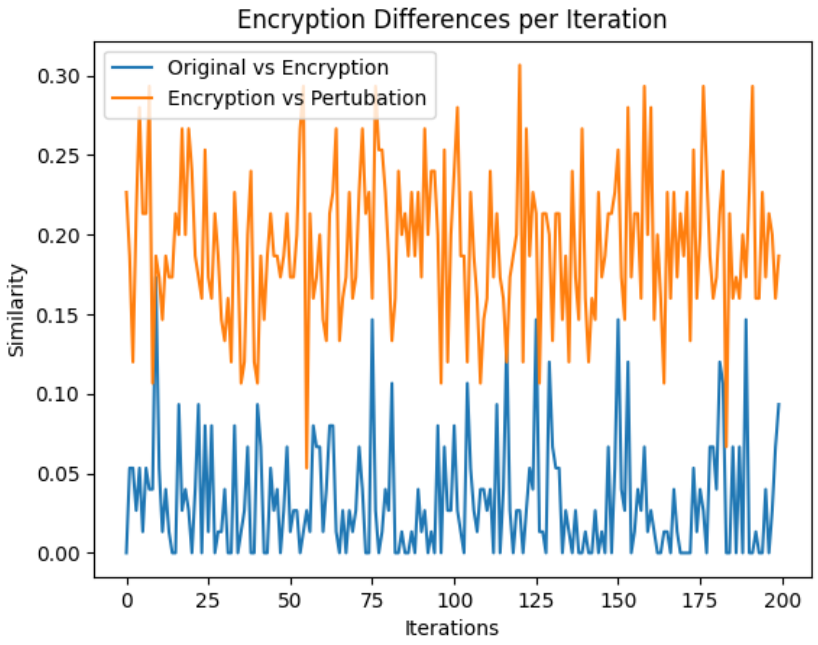


Fig. 17 - Graph from encryption scheme of 405 characters

5 Conclusions

These results show that our encryptor does a good job encrypting our message. Qualitatively, as demonstrated by Figures 13 and 15, we can see that the encrypted message does

not resemble the original message. Quantitatively, we can observe that slight perturbations to the system's key cause new encryptions that promote randomness. This distinction separates it from PRNGs, making the chaotic scheme a better model. Because of this, we have solved what we set out to do.

Interestingly, as the message size increases, there is no significant change in the cryptographic security of the system. Yet, the data between the similarity of the original vs the encrypted message and the data between the similarity of the encrypted message vs the perturbation message becomes more distinct and concentrated, as demonstrated through Figures 12, 14, and 17.

6 Potential Improvements

Future additions to the code could include adding picture encrypting processes by representing the information behind the photos as vectors and then subsequently encrypting those vectors. Cryptographically, there are various ways to improve the process demonstrated here.

One of these ways is to implement a secure key management system that includes key generation, storage, and distribution. Using industry-standard cryptographic key management practices to protect the encryption keys from unauthorized access would be a significant step forward in improving the quality of cryptography.

Another way to improve the code would be to implement comprehensive error handling and input validation mechanisms to gracefully handle exceptions, edge cases, and invalid inputs. Validate input data, such as the message and key, to ensure they meet the expected format and size requirements.

An issue that arises with cryptographic exchange is communication of the key in a secure manner. Incorporating secure key exchange protocols such as the Diffie-Hellman key exchange [5] to establish shared encryption keys securely between communicating parties would also help the process's security.

References

[1] "A New Chaotic Encryption Algorithm Based on Lorenz System and Sine Map" by Wei Wu and Ying Zhang (2018): This paper proposes a chaotic encryption algorithm based on the Lorenz system and sine map. The authors employ ODEs derived from the Lorenz system to generate pseudorandom sequences for encryption. (Wu and Zhang #)

[2] "A Secure Communication Scheme Based on Chaos and Cryptography" by Xu et al. (2013): This research paper proposes a chaotic ODE-based encryption scheme that combines the properties of the Lorenz system and a digital chaotic map. It demonstrates the feasibility of using chaotic ODEs for secure communication.

[3] "Chaotic Differential Equations for Secure Communication" by Ibragimov et al. (2012)

[4] "Encryption Algorithm Based on the Chaotic Dynamical System" by Li and Wang (2016): This paper presents an encryption algorithm based on the Chua's circuit, a chaotic ODE system. It explores the chaotic behavior of the Chua's circuit and demonstrates its potential for encryption applications.

[5] “A study on diffie-hellman key exchange protocols” by Mishra, Manoj & Kar, Jayaprakash. (2017). . International Journal of Pure and Applied Mathematics. 114. 10.12732/ijpam.v114i2.2.