DAYANANDA SAGAR UNIVERSITY

Devarakaggalahalli, Harohalli Kanakapura Road, Dt, Ramanagara, Karnataka 562112

Bachelor of Technology
in
COMPUTER SCIENCE AND ENGINEERING
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

22AM3603- Compiler Design and System Software

Report on

# Chakravyuh: Hybrid Key Exchange and Secure Communication

By

Agasthya R Kumar - ENG22AM0001
Akhila Rao D - ENG22AM0002
Gaana Shree S - ENG22AM0014
Nitya P Shetty - ENG22AM0037

Under the supervision of

Prof. PAVITHRA A

Assistant Professor, Computer Science & Engineering (AI & ML)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)
SCHOOL OF ENGINEERING
DAYANANDA SAGAR UNIVERSITY
(2024-2025)

# DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING

Department of Computer Science & Engineering
(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)
Kudlu Gate, Bangalore – 560068
Karnataka, India

## CERTIFICATE

This is to certify that the Compiler Design and System Software project work titled **"Chakravyuh: Hybrid Key Exchange and Secure Communication"** is carried out by **Agasthya R Kumar (ENG22AM0001) , Akhila Rao D (ENG22AM0002), Gaana Shree S (ENG22AM0014) and Nitya P Shetty (ENG22AM0037),** a bonafide students of Bachelor of Technology in Computer Science and Engineering (AI&ML) at the School of Engineering, Dayananda Sagar University, Bangalore in partial fulfillment for the award of degree in Bachelor of Technology in Computer Science and Engineering(AI&ML), during the year **2024-2025**.

**Guide & Chairperson:**

**Prof Pavithra A**

Assistant Professor
Dept. of CSE (AIML)
School of Engineering
Dayananda Sagar University

Date:

| **Name of the Examiner** | **Signature of Examiner** |
|---|---|
| 1. | |
| 2. | |

# DECLARATION

We, **Agasthya R Kumar (ENG22AM0001) , Akhila Rao D (ENG22AM0002), Gaana Shree S (ENG22AM0014) and Nitya P Shetty (ENG22AM0037),** are students of seventh semester B.Tech in **Computer Science and Engineering (AI & ML)**, at School of Engineering, **Dayananda Sagar University**, hereby declare that the Compiler Design and System Software project work titled **"Chakravyuh: Hybrid Key Exchange and Secure Communication"** has been carried out by us and submitted in partial fulfilment for the award of degree in **Bachelor of Technology in Computer Science and Engineering (AI&ML)** during the academic year **2024 - 2025**.

| **Student** | | **Signature** |
|---|---|---|
| **Name1:** | **Agasthya R Kumar** | |
| **USN:** | **ENG22AM0001** | |
| **Name2:** | **Akhila Rao D** | |
| **USN:** | **ENG22AM0002** | |
| **Name3:** | **Gaana Shree S** | |
| **USN:** | **ENG22AM0014** | |
| **Name4:** | **Nitya P Shetty** | |
| **USN:** | **ENG22AM0037** | |

**Place: Harohalli**

**Date:**

# ACKNOWLEDGEMENT

It is a great pleasure for us to acknowledge the assistance and support of many individuals who have been responsible for the successful completion of this project work.

First, we take this opportunity to express our sincere gratitude to the School of Engineering & Technology, Dayananda Sagar University for providing us with a great opportunity to pursue our Bachelor's degree in this institution.

*We would like to thank* ***Dr. Udaya Kumar Reddy K R, Dean, School of Engineering , Dayananda Sagar University*** *for his constant encouragement and expert advice.*

*It is a matter of immense pleasure to express our sincere thanks to* ***Dr. Jayavrinda Vrindavanam, Guide, Professor and Chaiperson****,* ***Department of Computer Science and Engineering (AI&ML)****,* ***Dayananda Sagar University,*** *for providing right academic guidance that made our task possible and for sparing her valuable time to extend help in every step of our project work, which paved the way for smooth progress and fruitful culmination of the project.*

*We would like to thank our* ***Project Coordinator Prof. Pavitra A, Assistant Professor*** *and all the staff members of Computer Science and Engineering for their support.*

*We are also grateful to our family and friends who provided us with every requirement throughout the course. We would like to thank one and all who directly or indirectly helped us in the Project work.*

# TABLE OF CONTENTS

# ABSTRACT

The advent of quantum computing poses a serious threat to existing cryptographic systems, particularly those based on classical algorithms such as RSA and ECC. To address these concerns and ensure secure data transmission during the transition to quantum-safe cryptography, this project proposes a hybrid file transfer system that integrates Elliptic Curve Diffie-Hellman (ECDH) with NTRU, a post-quantum key exchange protocol, and AES encryption. The hybrid approach combines the efficiency and proven security of ECDH with the quantum-resistance of NTRU, ensuring that file transfers remain secure against both classical and quantum computing threats. AES, a widely trusted symmetric encryption algorithm, is used for securing the actual data transmission. This hybrid system enables secure communication today while preparing for future quantum threats, making it a transitional solution for quantum-safe security. By leveraging the strengths of both classical and post-quantum cryptography, the proposed system ensures confidentiality, integrity, and authenticity of files shared over digital platforms, offering a robust defense against evolving cyber threats.

.

# CHAPTER 1 INTRODUCTION

As the digital world becomes increasingly interconnected, the need to securely transmit sensitive data has never been more critical. The widespread use of encryption protocols such as RSA and Elliptic Curve Cryptography (ECC) ensures the confidentiality and integrity of data during file transfers, but these algorithms are vulnerable to future threats posed by quantum computing. Quantum computers, once they reach sufficient scale and capability, will be able to efficiently solve mathematical problems that underpin many current cryptographic systems, such as integer factorization and the elliptic curve discrete logarithm problem, rendering traditional encryption methods obsolete.In response to these growing concerns, the field of post-quantum cryptography (PQC) has emerged, focusing on developing cryptographic algorithms that are resistant to quantum-based attacks. NTRU is one such post-quantum key exchange algorithm, designed to provide security against quantum adversaries by relying on the hardness of lattice-based problems, which are believed to be resistant to attacks by quantum computers. However, post-quantum cryptography is still in its developmental stages, and many of its algorithms are not yet widely adopted.To bridge the gap between classical cryptography and post-quantum solutions, hybrid cryptographic systems have been proposed. A hybrid approach combines both classical and post-quantum algorithms, offering the best of both worlds: the efficiency and maturity of traditional cryptographic methods with the quantum resilience of post-quantum algorithms. This report presents a hybrid file transfer system that uses Elliptic Curve Diffie-Hellman (ECDH) for classical key exchange, integrated with NTRU for post-quantum key exchange, and AES for symmetric encryption of the data being transferred. By combining these three cryptographic techniques, the system ensures secure file transfer during the transition period before quantum-safe cryptography becomes universally implemented.The hybrid file transfer system provides a robust and secure solution for data protection today while preparing for future quantum threats. This approach offers both forward-looking security and backward compatibility, ensuring that files can be safely transmitted over digital channels, regardless of the cryptographic vulnerabilities introduced by quantum computing in the near future.

In today's digital era, organizations, governments, and individuals rely heavily on the internet to share, store, and manage data. With the increasing volume of digital communication and data exchange, the need for secure file transfer has become more critical than ever. Unfortunately, this dependence on digital infrastructure has also led to a significant rise in cyberattacks, where malicious actors exploit vulnerabilities to steal, manipulate, or destroy data.Cyberattacks such as phishing, ransomware, data breaches, and man-in-the-middle attacks are now commonplace. Attackers often intercept data during transmission or exploit weak encryption methods to gain unauthorized access. High-profile incidents—such as breaches of healthcare records, corporate intellectual property theft, and financial fraud—highlight the urgent need for advanced, reliable cryptographic systems to ensure data security and privacy.To combat these threats, modern cryptographic techniques such as RSA, AES, and Elliptic Curve Cryptography (ECC) have been widely adopted. These methods rely on the computational difficulty of mathematical problems like integer factorization and discrete logarithms to secure data. However, a new threat is emerging—quantum computing—that challenges the very foundation of classical cryptography.

Quantum computers, leveraging principles of quantum mechanics, have the potential to perform complex calculations far more efficiently than classical computers. Algorithms such as Shor’s algorithm can solve problems like prime factorization and discrete logarithms in polynomial time, which would allow a sufficiently powerful quantum computer to break widely-used encryption schemes like RSA and ECC. This means that encrypted data currently considered secure could be decrypted in the future once quantum computers become practically viable.

# CHAPTER 2 PROBLEM DEFINITION

The increasing threat posed by quantum computing presents a serious challenge to the security of data transmitted over the internet. While classical encryption techniques like RSA and ECC have long been trusted for secure communication, they are vulnerable to quantum attacks, particularly those based on Shor's algorithm, which can efficiently factor large numbers and compute discrete logarithms in polynomial time. As quantum computers continue to advance, they will render these classical encryption methods insecure, exposing sensitive data to the risk of being decrypted and compromised.

In addition to the potential quantum threat, the current cryptographic landscape faces several other issues:

- Vulnerability to Quantum Attacks: Current public-key encryption schemes, including RSA and ECC, will be broken by quantum computers capable of executing Shor's algorithm. The advent of quantum computers will thus make many encryption techniques obsolete, leading to potential breaches of sensitive data stored or transferred digitally. This creates a critical need for a secure file transfer system that can withstand quantum attacks.
- Transition to Post-Quantum Cryptography: While post-quantum algorithms like NTRU (a lattice-based key exchange algorithm) have shown promise in securing data against quantum attacks, they are not yet widely adopted or standardized. The transition to quantum-safe cryptography is expected to take several years, during which time data security must still be ensured. The challenge lies in ensuring that current file transfer systems remain secure against quantum threats while maintaining backward compatibility with classical encryption techniques.
- Hybrid Solutions for Interim Security: The problem is further complicated by the need to ensure that file transfer systems can securely communicate both today and in the future, as the cryptographic landscape evolves. A hybrid approach combining both classical and post-quantum cryptographic methods provides an interim solution to this challenge.

- This hybrid method can leverage the strengths of both ECDH, a proven and efficient classical key exchange mechanism, and NTRU, a quantum-safe alternative, to offer robust encryption during this transition period.
- Efficient Data Encryption: The security of the file transfer system depends not only on the encryption of the key exchange process but also on the actual data being transmitted. AES (Advanced Encryption Standard) has long been trusted as a secure and efficient symmetric encryption algorithm for encrypting large amounts of data. The challenge is to integrate AES encryption into a system that uses both classical and post-quantum key exchange protocols without introducing excessive computational overhead.
- Long-term Security and Future-proofing: Even though large-scale quantum computers are not yet available, they are expected to arrive within the next few decades. Given the long lifespan of encryption standards and protocols, it is critical to design systems that will continue to secure data beyond the era of quantum computers. Ensuring that file transfers are quantum-resistant, while not compromising the performance and user experience of current systems, is essential for long-term data security.

The hybrid file transfer system proposed in this report aims to solve these challenges by combining Elliptic Curve Diffie-Hellman (ECDH) with NTRU to perform the key exchange and using AES encryption to securely transfer the data. The system will maintain backward compatibility with existing systems using classical cryptography while providing quantum resistance through the integration of lattice-based cryptography. By addressing both classical and quantum vulnerabilities, this hybrid approach ensures that file transfer systems can securely handle sensitive data both today and in the future, providing a solution to the growing threat of quantum computing.

# CHAPTER 3 LITERATURE REVIEW

The advent of quantum computing poses an existential threat to classical cryptography. Quantum algorithms such as Shor's can factor integers and compute discrete logarithms in polynomial time, effectively breaking RSA and elliptic-curve schemes [4], [12]. Similarly, Grover's algorithm could roughly halve the effective key length of symmetric ciphers, so that an AES-128 key offers only about 64 bits of security against a quantum adversary [12]. In effect, data encrypted today must often remain secret for decades, so "store-now, decrypt-later" attacks using a future quantum computer are a serious concern [4], [12]. These facts have spurred an urgent transition to post-quantum cryptography (PQC), where algorithms are chosen to resist quantum attacks. NIST has already selected lattice-based CRYSTALS-Kyber for key encapsulation and CRYSTALS-Dilithium for signatures [12]. However, newly proposed PQC schemes must be used cautiously: some finalists (e.g., SIKE, Rainbow) were broken after standardization rounds. Thus, a hybrid migration strategy is recommended: combining classical and PQC methods so that even if one component is compromised, the other maintains security [5], [8].

Secure communication in the post-quantum era therefore requires careful design. Hybrid systems can mix classical symmetric and asymmetric encryption with PQC and even quantum key distribution (QKD). A hybrid approach typically generates a shared secret via both a conventional algorithm and a post-quantum one, then derives a single session key [5], [8]. This way, breaking either component alone is insufficient for an attacker. Conversely, if new PQC schemes turn out to have flaws, the classical component still offers protection. Many standards bodies and vendors now emphasize hybrid solutions. For example, guidelines note that schemes like ECDH combined with Kyber or the integration of QKD into TLS can ensure backward compatibility while adding quantum resistance [28]. Similarly, wolfSSL recommends hybrid key-exchanges (e.g., ECDHE P-256‖Kyber768) and hybrid signatures (e.g., RSA-3072‖Dilithium2) for TLS [13]. In summary, the quantum threat to confidentiality and integrity motivates hybrid post-quantum cryptography: using both symmetric and asymmetric encryption (classical and quantum-safe) and even steganography to build a more resilient secure channel.

## 3.1. Symmetric Encryption (AES, Blowfish)

Symmetric ciphers remain the workhorses for bulk encryption. AES (Advanced Encryption Standard) is widely deployed; even against quantum attacks it can be made safe by doubling key lengths (Grover's search reduces a 128-bit key to 64-bit security) [12]. In fact, Kyber-1024 is intended to match roughly 256-bit classical security (comparable to AES-256) [64], while Kyber-768 matches AES-192. In practice, using AES-256 or comparable secure symmetric ciphers (like ChaCha20) is recommended for confidentiality [11]. For example, many hybrid frameworks use AES-128 or AES-256 to encrypt the actual data once the key is agreed.

By contrast, older ciphers like Blowfish have fallen out of favor. Blowfish uses a 64-bit block size (and up to 448-bit key), which today is considered too small for large data volumes. Indeed, the Sweet32 attacks demonstrated that any 64-bit-block cipher (e.g., Blowfish, 3DES) in CBC or similar modes can succumb to practical collision attacks if enough ciphertext is gathered [40]. As noted by Bhargavan and Leurent, "it is well-known … that a short block size makes a block cipher vulnerable to birthday attacks" [10]. AES instead uses 128-bit blocks, avoiding these issues [10]. Accordingly, post-quantum migration guidelines list AES-CTR/GCM as recommended primitives and explicitly deprecate Blowfish and DES-family ciphers [11].

Several works demonstrate hybrid use of symmetric ciphers. Shafique et al. propose a telemedicine image-encryption scheme where a QKD-derived key secures a one-time-pad (OTP) used to encrypt symmetric AES-like keys, and then multiple AES-based confusion/diffusion steps scramble image pixels [7]. While their focus is on IoT imaging, the principle is general: a high-rate symmetric cipher (or OTP) is applied to data, with keys protected by stronger mechanisms. In any case, the symmetric cipher's role is to encrypt arbitrary-length messages with minimal overhead, making it an essential component of hybrid systems.

## 3.2. Asymmetric Encryption (RSA, Lattice-Based PQC)

Classical public-key schemes like RSA and ECC (Elliptic Curve Cryptography) are fundamentally broken by quantum computers [4]. Shor's algorithm will render RSA-2048 and typical elliptic curves insecure, so they must eventually be phased out. In the hybrid era, RSA or ECC may still be used in parallel with PQC algorithms, but pure RSA/ECC should be avoided for future-proof security. Performance studies show that lattice-based PQC can match or outperform classical schemes. Demir et al. benchmarked CRYSTALS-Kyber and Dilithium and found they "achieve efficient execution times, outperforming classical schemes such as RSA and ECDSA at equivalent security levels" [12]. This suggests that Kyber/Dilithium not only provide quantum resistance but can be practical replacements.

Accordingly, NIST has standardized Kyber (a key-encapsulation mechanism) and Dilithium (a digital signature scheme) for public-key encryption and signing [12]. Many hybrid proposals use these new primitives. For example, industry products often pair Kyber768 with ECDH in a TLS handshake, or use RSA-3072/Dilithium2 dual certificates for signatures [13]. The MDPI review by Dekkaki et al. explicitly notes hybrid schemes that combine classical and PQC methods, e.g., "keys from both schemes combined… for instance, combining ECDH with the Kyber post-quantum KEM scheme" [8]. Practical implementations follow this idea: wolfSSL supports hybrid key exchange (e.g., ECDHE-P-256‖Kyber768) and hybrid signatures (e.g., ECDSA-P256‖Dilithium2 or RSA-3072‖Dilithium2) in TLS 1.3 [13].

Some research directly compares hybrid combinations. For example, a recent survey considered using RSA+AES versus Kyber+AES for encrypting files. While preliminary, such studies show that the "hybrid method works but it also requires extra overhead and affects performance" [9]. In practice, the added cost of running two key exchanges is relatively small compared to the total data transfer, and is widely considered worthwhile for future security [6], [5]. The main caution is to avoid "hybrid OR" setups where a client can downgrade to the classical algorithm; instead one should enforce "hybrid AND" (both algorithms used) to prevent downgrade attacks [6], [8].

## 3.3. Hybrid Cryptographic Approaches

Hybrid cryptosystems intentionally combine classical and post-quantum primitives in parallel. A common pattern is to perform a dual key-encapsulation: one with a conventional KEM (e.g., RSA-KEM or ECDH) and one with a PQC KEM (e.g., Kyber), then feed both outputs into a key-derivation function to produce the session key [5], [8]. For signatures, one may sign a message with both a classical signature (e.g., ECDSA) and a PQC signature (e.g., Dilithium) and require both to be valid. In either case, the security is "AND"-ed: an attacker must break both components to forge or decrypt. Xiphera's blog describes this well: a hybrid system "remains secure due to the PQC component" if classical is broken, and vice versa [5].
Several standards and studies emphasize hybridization. The IETF is defining hybrid KEX for TLS 1.3, and both NIST and international agencies recommend deploying PQC in parallel with elliptic curves [28]. Dekkaki et al. note that ENISA and others "emphasize the necessity of hybridization for future cryptographic security" [8]. Research has begun to quantify the impact: for instance, Prototyping PQC guidelines from NIST (CSRC) advocate testing algorithms in a hybrid fashion. The wolfSSL hybrid solution listing shows practical combinations (e.g., ECDHE-P256‖Kyber768, RSA-3072‖Dilithium2) that are already implemented in TLS libraries [13].

Of course, hybrid schemes introduce their own considerations. Two well-known issues are downgrade attacks and performance trade-offs. If a server offers both classical and PQC options ("hybrid OR"), an attacker might force the server to use only the classical, vulnerable cipher [6], [8]. To avoid this, pure "hybrid AND" must be enforced internally. Performance-wise, adding a PQC step increases CPU time and handshake size. However, studies find that Kyber and Dilithium can be hardware-accelerated (e.g., AVX2), narrowing the gap with RSA/ECDSA [12]. In any case, hybrid adoption is seen as an intermediate step: once PQC algorithms mature, they may stand alone. For now, hybrid deployments in protocols like TLS and IPsec are widely regarded as the safest migration path [5], [8].

# CHAPTER 4 PROJECT DESCRIPTION

The project implements a Hybrid Cryptography system for secure file encryption and decryption, combining the strengths of multiple cryptographic algorithms across several layers to ensure enhanced data protection. The core idea is to use symmetric encryption algorithms such as AES and Blowfish for high-speed, bulk data encryption, while leveraging asymmetric encryption (RSA) to securely handle key exchange. This dual-layer approach ensures that data is encrypted efficiently, and the keys used in that encryption are protected during transfer or storage.

The project is divided into several modular notebooks. The main project notebook serves as the base for implementing the encryption and decryption logic using the hybrid approach. It handles the generation of RSA key pairs, uses AES or Blowfish to encrypt the data, and then encrypts the session keys using RSA. The hybrid-crypto-tests supports both manual and automated testing, enabling verification of cryptographic functions over a variety of input types. It also logs testing outcomes, making the system easier to debug and maintain.

Another key component is the compression mechanisms (like ZIP or zlib) which focuses on file-specific cryptographic operations. It includes to reduce file size before encryption, enhancing storage and transmission efficiency. This file also manages metadata, ensuring that filenames and extensions are preserved correctly after encryption and decryption. Finally, a framework to automatically test the entire system using predefined inputs, ensuring consistency and reliability across executions is constructed.

Together, these components demonstrate an effective and secure hybrid cryptographic model that can be used for protecting sensitive data, particularly in scenarios involving file storage, transfer, or secure communications. The layered approach, combining encryption algorithms and testing infrastructure, showcases practical application of hybrid cryptography in real-world systems.

# CHAPTER 5 REQUIREMENTS

## 5.1. Software:

**1. Operating System**

Compatible with Windows, Linux, or macOS.

**2. Programming Language**

Python 3.6+

**3. Python Libraries**

Pycryptodome: For AES, RSA, Blowfish, and other cryptographic operations.

notebook or jupyterlab: To run the .ipynb files.

**4. Jupyter Notebook Environment**

or any IDE that supports Jupyter, like VS Code, JupyterLab, or Google Colab.

## 5.2. Hardware:

**1. Processor**

Minimum: Dual-core CPU (Intel i3 or equivalent)

Quad-core CPU (Intel i5/i7 or equivalent) for faster file operations and testing.

**2. RAM**

Minimum: 4 GB

8 GB or more (especially if handling large files or running automated tests)

**3. Storage**

Minimum: 500 MB free disk space

1–2 GB, depending on the size of test files and encrypted data.

# CHAPTER 6 WORK FLOW

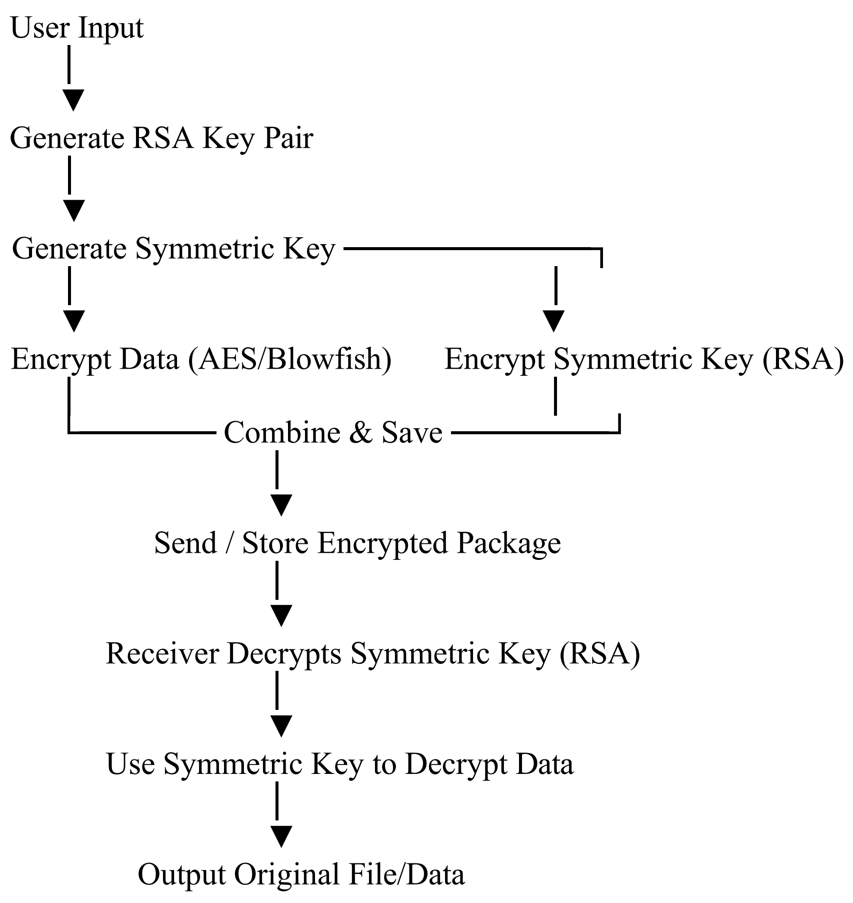

User Input
Generate RSA Key Pair
Generate Symmetric Key
Encrypt Data (AES/Blowfish)
Encrypt Symmetric Key (RSA)
Combine & Save
Send / Store Encrypted Package
Receiver Decrypts Symmetric Key (RSA)
Use Symmetric Key to Decrypt Data
Output Original File/Data

Figure 1: Workflow

# CHAPTER 7 METHODOLOGY

## Data Encryption

The System consists of three Encryption Layers, a Key Generator, and a List of Keys. The Key Generator generates the random n-bits Key depending on the Encryption Al gorithm, while the List of Keys stores the Key Generated in each layer.

- Step 1: The plaintext P is first Encrypted using the Blowfish Algorithm with a 32 Bit / 64 Bit / 128 Bit Key, KBlowf ish. The Key KBlowf ish is generated by the Key Generator and is used for Blowfish Encryption. It is then appended to the List of Keys, L. The Plaintext, P is encrypted to generate Ciphertext C1.

  C1 = Blowf ish(Plaintext = P, Key = KBlowf ish) (5.1)

  L = [ ] ⊕ KBlowf ish (5.2)

- Step 2: The Ciphertext, C1 is then encrypted using RSA Encryption with the 1024/2048 Bit Public Key, KRSA−P ublic generated by the Key Generator. A Private Key, KRSA−P rivate , is also generated for Decryption. While the Public Key is used in Encryption, it is not stored in the List of Keys, L. The Private Key generated is appended to the List of Keys. C1 is encrypted to generate Ciphertext C2.

  C2 = RSA(Plaintext = C1, Key = KRSA−P ublic) (5.3)

  L = [ KBlowf ish ] ⊕ KRSA−P rivate (5.4)

- Step 3: The Ciphertext, C2 is then encrypted using AES-128 Encryption with the 128 Bit, KAES generated by the Key Generator. The Key, KAES generated is appended to the List of Keys, L. This Step gives the final encrypted ciphertext C.

  Ciphertext, C = AES(Plaintext = C2, Key = KAES) (5.5)

  L = [ KBlowf ish , KRSA−P rivate ]KAES (5.6)s.

## Key Encryption:

Using the proposed system, the Keys used for encryption at the various layers can be securely stored. The List of keys, L stores all the keys generated throughout the Data Encryption Process. Whenever the key for a particular Encryption Layer is generated, it is appended to the List of Keys, L. In the system, the encryption layers are Blowfish, RSA, and AES, respectively, so the Keys used, are stored in the same order as:

List of Keys, L = [ KBlowf ish , KRSA−P rivate , KAES ] (5.8)

- Step 1: This List, L is then passed into a function that converts the list into a single string of keys separated by separators ( x , * , / )

  LS = Stringify( L, separator = 0 ×0 ) (5.9)

- Step 2: The String, LS is then encrypted using the AES Encryption Algorithm with a Key generated from user-input password. The user inputs a password, PW which is hashed using SHA1, & the first 16 Bits of the Hash is used as the key KP assword . The Key, KP assword is used for the Encryption, generating the encrypted string LS−Encrypted.

  HashedP assword, HP = SHA(PW ) (5.10)

  Key, KP assword = HP [0 : 16] (5.11)

  LS−Encrypted = AES(LS, KP assword) (5.12)

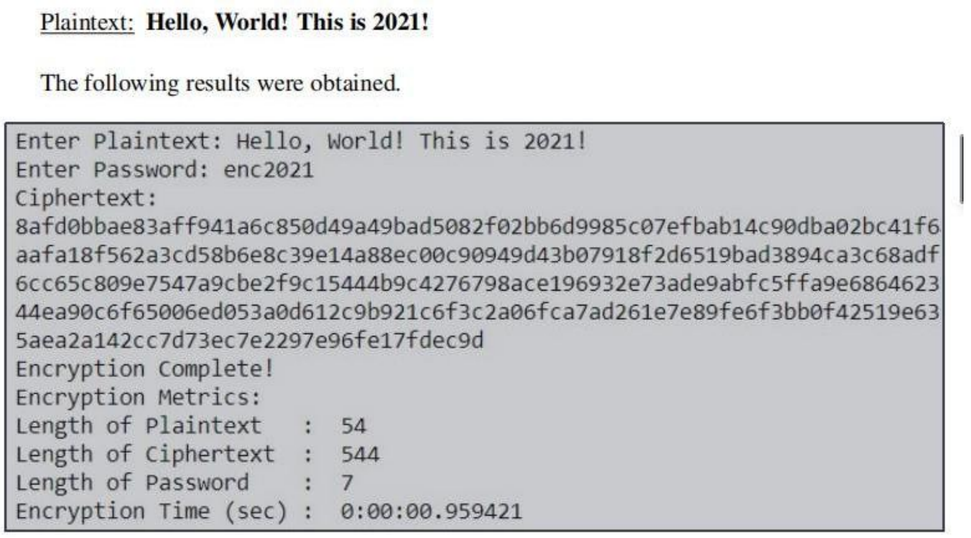
- Step 3: This Encrypted string is then Embedded into a Cover Image using Least Significant Bit Steganography, giving the embedded Stego-Image.

  Stego Image = LSB − Steganography(LS−Encrypted , Cover − Image) (5.13)

The Stego-Image is transferred to the Receiver along with the Encrypted Data.

# CHAPTER 8 Results

The Proposed Hybrid Crypto-system was implemented using Python and tested on a Windows PC with an Intel i3 processor and 4 GB of RAM. For demonstrating the En cryption of data, the following plaintext was encrypted. To encrypt the keys, ‘enc2021’was used as the password.



Plaintext: **Hello, World! This is 2021!**

The following results were obtained.

```
Enter Plaintext: Hello, World! This is 2021!
Enter Password: enc2021
Ciphertext:
8afd0bbae83aff941a6c850d49a49bad5082f02bb6d9985c07efbab14c90dba02bc41f6
aafa18f562a3cd58b6e8c39e14a88ec00c90949d43b07918f2d6519bad3894ca3c68adf
6cc65c809e7547a9cbe2f9c15444b9c4276798ace196932e73ade9abfc5ffa9e6864623
44ea90c6f65006ed053a0d612c9b921c6f3c2a06fca7ad261e7e89fe6f3bb0f42519e63
5aea2a142cc7d73ec7e2297e96fe17fdec9d
Encryption Complete!
Encryption Metrics:
Length of Plaintext   :  54
Length of Ciphertext  :  544
Length of Password    :  7
Encryption Time (sec) :  0:00:00.959421
```

Figure 2: Result:

# REFERENCES

[1] Abdulbast A Abushgra and Khaled M Elleithy. A shared secret key initiated by epr authentication and qubit transmission channels. *IEEE Access*, 5:17753–17763, 2017.

[2] Shweta Agrawal and Ishaan Preet Singh. Reusable garbled deterministic finite automata from learning with errors. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[3] Bechir Alaya, Lamri Laouamer, and Nihel Msilini. Homomorphic encryption systems statement: Trends and challenges. *Computer Science Review*, 36:100235, 2020.

[4] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[5] Majid Alotaibi. Improved blowfish algorithm-based secure routing technique in iot-based wsn. *IEEE Access*, 9:159187–159197, 2021.

[6] Abdelrahman Altigani, Shafaatunnur Hasan, Bazara Barry, Shiraz Naserelden, Muawia A Elsadig, and Huwaida T Elshoush. A polymorphic advanced encryption standard–a novel approach. *IEEE Access*, 9:20191–20207, 2021.

[7] Sangwoo An and Seog Chung Seo. Designing a new xts-aes parallel optimization implementation technique for fast file encryption. *IEEE Access*, 10:25349–25357, 2022.

[8] S Anandakumar. Image cryptography using rsa algorithm in network security. *International Journal of Computer Science & Engineering Technology*, 5(9):326–330, 2015.

[9] Prabhanjan Ananth, Alexander Poremba, and Vinod Vaikuntanathan.

Revocable cryptography from learning with errors. *arXiv preprint arXiv:2302.14860*, 2023.

[10] Chinnaraji Annamalai. Application of factorial and binomial identities in information, cybersecurity and machine learning. *International Journal of Advanced Networking and Applications*, 14(1):5258– 5260, 2022.

[11] Chinnaraji Annamalai. Computation of combinatorial geometric series and its combinatorial identities for cryptographic algorithm and machine learning. 2022.

[12] Chinnaraji Annamalai. Computation of multinomial and factorial theorems for cryptography and machine learning. 2022.

[13] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela J¨aschke, Christian A Reuter, and Martin Strand. A guide to fully homomorphic encryption. *Cryptology ePrint Archive*, 2015.

[14] Kwame Assa-Agyei and Funminiyi Olajide. A comparative study of twofish, blowfish, and advanced en- cryption standard for secured data transmission. *International Journal of Advanced Computer Science and Applications*, 14(3), 2023.

[15] Ayush Bhatia, Vimal Bibhu, Bhanu P Lohani, and Pradeep K Kushwaha. An application framework for quantum computing using artificial intelligence techniques. In *2020 Research, Innovation, Knowledge Management and Technology Application for Business Sustainability (INBUSH)*, pages 264–269. IEEE, 2020.

## CODE:

```
from Crypto.Cipher import Blowfish, PKCS1_OAEP, AES
from Crypto.PublicKey import RSA
from Crypto.Util.Padding import pad, unpad
from binascii import hexlify , unhexlify
import hashlib , json, string, random
from stegano import lsb
from datetime import datetime

# Key Generator
def key_generator(size, case="default", punctuations="required"):
if case=="default" and punctuations=="required":
return ''.join(random.choices(string.ascii_uppercase +
string.ascii_lowercase + string.digits + string.punctuation, k = size))
elif case=="upper-case-only" and punctuations=="required":
return ''.join(random.choices(string.ascii_uppercase + string.digits +
string.punctuation, k = size))
elif case=="lower-case-only" and punctuations=="required":
return ''.join(random.choices(string.ascii_lowercase + string.digits +
string.punctuation, k = size))
elif case=="default" and punctuations=="none":
return ''.join(random.choices(string.ascii_uppercase + string.digits +
string.ascii_lowercase, k = size))
elif case=="lower-case-only" and punctuations=="none":
return ''.join(random.choices(string.ascii_lowercase + string.digits , k =
size))
elif case=="upper-case-only" and punctuations=="none":
return ''.join(random.choices(string.ascii_uppercase + string.digits, k =
size))
# Plaintext Input
plaintext = input('Enter Plaintext: ').encode()

log_plaintext_length = len(hexlify(plaintext))

# Password for Keys
password = input('Enter Password: ')
log_password_length = len(password)

log_start_time = datetime.now()

hash = hashlib.sha1()
hash.update(password.encode())
password_encryption_cipher = AES.new( hash.hexdigest()[:16].encode() ,
AES.MODE_CBC, iv= '16bitAESInitVect'.encode())

# Dictionary of Keys
keys_iv = {}
```

```
# Blowfish Layer 1

blowfish_key = key_generator(size=16).encode()
blowfish_cipher = Blowfish.new(blowfish_key, Blowfish.MODE_CBC)

blowfish_ciphertext = blowfish_cipher.encrypt(pad(plaintext,
Blowfish.block_size ))

keys_iv['blowfish_iv'] = hexlify(blowfish_cipher.iv).decode()
keys_iv['blowfish_key'] = hexlify(blowfish_key).decode()

# RSA Layer 2

rsa_key = RSA.generate(2048)
rsa_private_key = rsa_key
rsa_public_key = rsa_key.publickey()

cipher_rsa = PKCS1_OAEP.new(rsa_public_key)
rsa_plaintext = blowfish_ciphertext

rsa_ciphertext = bytearray()
for i in range(0, len(rsa_plaintext), 190):
rsa_ciphertext.extend(cipher_rsa.encrypt(rsa_plaintext[i:i+190]))

keys_iv['rsa_n'] = rsa_private_key.n
keys_iv['rsa_e'] = rsa_private_key.e
keys_iv['rsa_d'] = rsa_private_key.d

# AES Layer 3
aes_key = key_generator(size=16).encode()
aes_cipher = AES.new(aes_key, AES.MODE_CBC)
aes_plaintext = rsa_ciphertext

aes_ciphertext = aes_cipher.encrypt(pad(aes_plaintext, AES.block_size))

ciphertext = aes_ciphertext

log_ciphertext_length = len(hexlify(ciphertext))

keys_iv['aes_iv'] = hexlify(aes_cipher.iv).decode()
keys_iv['aes_key'] = hexlify(aes_key).decode()


# Encryption of Key and IV String
encrypted_keys_and_iv =
hexlify(password_encryption_cipher.encrypt(pad(json.dumps(keys_iv).encode()
, AES.block_size)))

#LSB Steg
lsb_stegano_image = lsb.hide("/content/drive/MyDrive/SEM-
6/CDSS/test/cover_image.png", encrypted_keys_and_iv.decode())
```

```
lsb_stegano_image.save("/content/drive/MyDrive/SEM-
6/CDSS/test/stego_image.png")
```

```
log_end_time = datetime.now()
```

```
log_duration = str(log_end_time - log_start_time)
```

```
with open('/content/drive/MyDrive/SEM-6/CDSS/logs/encryption-log.txt',
'a+') as log_file:
log_file.write( "\n| " +str(log_plaintext_length)
+" | "+str(log_ciphertext_length)
+" | "+str(log_password_length)
+" | "+log_start_time.strftime("%H:%M:%S")
+" | "+log_end_time.strftime("%H:%M:%S")
+" | "+str(log_duration)
+" |"
)
```

```
print('Plaintext in Hexadecimal: ')
print(hexlify(plaintext).decode())
print('Ciphertext: ')
print(hexlify(ciphertext).decode())
print('Encryption Complete!')
```

```
print('Encryption Metrics:')
```

```
print( "Length of Plaintext : "+str(log_plaintext_length)+
"\nLength of Ciphertext : "+str(log_ciphertext_length)+
"\nLength of Password : "+str(log_password_length)+
"\nEncryption Time (sec) : "+str(log_duration))
# Password for Keys
password = input('Enter Password: ')

log_password_length = len(password)
log_ciphertext_length = len(hexlify(ciphertext))

log_start_time = datetime.now()

# LSB Steg
unhide_encrypted_keys_and_iv = lsb.reveal("/content/drive/MyDrive/SEM-
6/CDSS/test/stego_image.png").encode()

hash = hashlib.sha1()
hash.update(password.encode())
password_decryption_cipher = AES.new( hash.hexdigest()[:16].encode() ,
AES.MODE_CBC, iv= '16bitAESInitVect'.encode())

decrypted_keys_iv =
json.loads(unpad(password_decryption_cipher.decrypt(unhexlify(unhide_encry
pted_keys_and_iv)), AES.block_size))
```

```
#Initializations
decryption_key_aes = unhexlify(decrypted_keys_iv['aes_key'])
decryption_iv_aes = unhexlify(decrypted_keys_iv['aes_iv'])
decryption_key_rsa = RSA.construct(rsa_components =
(decrypted_keys_iv['rsa_n'] , decrypted_keys_iv['rsa_e'] ,
decrypted_keys_iv['rsa_d']))
decryption_iv_blowfish = unhexlify(decrypted_keys_iv['blowfish_iv'])
decryption_key_blowfish = unhexlify(decrypted_keys_iv['blowfish_key'])


aes_cipher_decryption = AES.new(decryption_key_aes, AES.MODE_CBC,
iv=decryption_iv_aes)
rsa_cipher_decryption = PKCS1_OAEP.new(decryption_key_rsa)
blowfish_cipher_decryption = Blowfish.new(decryption_key_blowfish,
Blowfish.MODE_CBC, iv=decryption_iv_blowfish)

# AES DECRYPTION
ciphertext_rsa = unpad(aes_cipher_decryption.decrypt(ciphertext),
AES.block_size)
# RSA DECRYPTION
ciphertext_blowfish = bytearray()
for i in range(0, len(ciphertext_rsa),256):
    ciphertext_rsa_segment = ciphertext_rsa[i:i+256]

ciphertext_blowfish.extend(rsa_cipher_decryption.decrypt(ciphertext_rsa_se
gment))

# BLOWFISH DECRYPTION
decrypted_plaintext =
unpad(blowfish_cipher_decryption.decrypt(ciphertext_blowfish),
Blowfish.block_size)

log_end_time = datetime.now()
log_duration = str(log_end_time - log_start_time)
log_plaintext_length = len(hexlify(decrypted_plaintext))

with open('/content/drive/MyDrive/SEM-6/CDSS/logs/decryption-log.txt',
'a+') as log_file:
    log_file.write( "\n| "   +str(log_ciphertext_length)
                +"        | "+str(log_plaintext_length)
                +"        | "+str(log_password_length)
                +"       | "+log_start_time.strftime("%H:%M:%S")
                +"   | "+log_end_time.strftime("%H:%M:%S")
                +"  | "+str(log_duration)
                +" |"
               )

print("Decrypted Ciphertext: ", decrypted_plaintext.decode())
print('File Decryption Complete!')
```