A Hybrid Cryptographic Architecture for Secure Digital File Sharing and Integrity Verification

Tanvi Santosh Patil[1], Mrs. Krutika Churi[2]
[1]*Student, Sonopant Dandekar Arts, V.S. Apte Commerce and M.H. Mehta Science College*
[2]*Assistant Professor, Sonopant Dandekar Arts, V.S. Apte Commerce and M.H. Mehta Science College*

***Abstract*—In an increasingly connected digital ecosystem, file sharing across public and untrusted networks exposes sensitive organizational and personal data to eavesdropping, interception, and illicit modification [cite: 6]. Conventional transmission protocols frequently lack comprehensive end-to-end security, creating significant vulnerabilities during data transit and storage [cite: 6, 7]. This paper presents the architecture and implementation of a robust, lightweight Secure File Sharing System Using Cryptography designed to deliver end-to-end confidentiality, non-repudiation, integrity, and authentication [cite: 6, 8]. The framework employs a hybrid cryptographic model combining symmetric encryption (AES-256) for rapid payload transformation with asymmetric primitives (RSA / ECC) for secure session key encapsulation and exchange [cite: 6, 8]. Data integrity and tamper detection are verified using cryptographic hashing via SHA-256[cite: 6, 8]. Implemented on a Python-Flask backend and paired with an interactive HTML5/CSS3 client interface, the system achieves sub-second cryptographic processing on standard computing infrastructure [cite: 7, 8]. The solution provides a scalable, zero-licensing approach to mitigating data theft and unauthorized tampering in decentralized digital communications [cite: 6, 8].**

***Index Terms*—Cryptography, Secure File Sharing, Hybrid Encryption, Advanced Encryption Standard (AES-256), RSA, SHA-256, Flask, Data Confidentiality, Data Integrity [cite: 6, 8].**

I. INTRODUCTION

The extensive reliance on internet-driven platforms has made digital file exchange fundamental to enterprise operations, academic collaboration, and governance [cite: 6, 8]. However, transmitting data over open, insecure networks exposes plaintext information to severe security risks, including packet interception, man-in-the-middle (MITM) attacks, and unauthorized access [cite: 6]. While standard cloud repositories provide centralized access, files stored or transmitted without granular user-controlled encryption remain exposed to insider threats, unauthorized third parties, and system breaches [cite: 6, 7].

Protecting digital files requires transforming readable plaintext into encrypted ciphertext that is accessible exclusively to authorized parties possessing legitimate decryption credentials [cite: 6]. Cryptographic primitives—symmetric block ciphers, asymmetric key pairs, cryptographic message digests, and digital signatures—form the foundation of secure communication architectures [cite: 6]. This study develops a functional, user-centric Secure File Sharing System that combines the efficiency of symmetric ciphers with the robust key-exchange capabilities of asymmetric cryptography [cite: 6, 8]. By embedding automated file encryption, secure key distribution, and cryptographic hashing into an accessible web platform, the framework guarantees end-to-end data confidentiality, access control, and tamper-evident file integrity [cite: 6, 7, 8].

II. LITERATURE REVIEW

The preservation of digital trust and information security relies on addressing three primary parameters: confidentiality, authenticity, and integrity [cite: 5, 6]. Early secure transmission systems depended solely on symmetric cryptographic primitives, such as the Data Encryption Standard (DES) or Triple DES (3DES), but these algorithms suffer from reduced key spaces and substantial computational vulnerabilities [cite: 6, 9]. The adoption of the Advanced Encryption Standard (AES) established an efficient, NIST-standardized block cipher capable of securing high-volume

payloads with low computational overhead [cite: 6, 8, 9]. However, pure symmetric encryption poses a fundamental operational challenge: secure key distribution [cite: 6]. Sharing a symmetric secret key across an unencrypted channel exposes the entire system to interception [cite: 6]. Conversely, asymmetric cryptographic systems—such as Rivest–Shamir–Adleman (RSA) or Elliptic Curve Cryptography (ECC)—solve the key distribution dilemma through mathematically linked public-private key pairs, but their computational complexity makes them impractical for bulk file encryption [cite: 6, 8, 9].

To overcome these distinct operational limitations, modern applied security relies on hybrid cryptographic frameworks [cite: 6, 8]. In a hybrid scheme, bulk payload data is transformed using high-throughput symmetric ciphers (such as AES-256), while the corresponding transient symmetric keys are encrypted and distributed using asymmetric mechanisms (RSA or ECC) [cite: 6, 8]. Integrity verification is maintained by generating fixed-length message digests via cryptographic hashing (SHA-256), which produces distinct digest alterations whenever an intercepted file is modified [cite: 6, 8, 9]. The system proposed in this research builds upon these proven foundations to engineer an accessible, lightweight web architecture tailored for secure digital file transfers without external proprietary dependencies [cite: 7, 8].

III. SYSTEM ARCHITECTURE

The proposed system architecture isolates file processing, key orchestration, and client interactions into distinct execution tiers [cite: 8]. The client interacts through a lightweight web interface, dispatching user payloads to a Python-Flask backend that manages symmetric session generation, asymmetric key encapsulation, and database persistence [cite: 7, 8].

IV. METHODOLOGY

The operational framework focuses on delivering secure transit through hybrid cryptographic pipelining [cite: 6, 8]. Let an unencrypted input file be represented as M [cite: 6].

$h = SHA\text{-}256(M)$ [cite: 6, 8]

This cryptographic digest is preserved to provide tamper evidence [cite: 6, 8]. The file payload is subsequently enciphered using AES-256 under Cipher Block Chaining (CBC) or Galois/Counter Mode (GCM) using a transient symmetric key K_{sym} and initialization vector IV [cite: 8, 9]:

$C = E_{AES}(K_{sym}, M, IV)$ [cite: 6]

To eliminate symmetric transmission exposure, K_{sym} is encapsulated using the recipient's public key $PK_{recipient}$ [cite: 6, 8]:

$C_{key} = E_{Asym}(PK_{recipient}, K_{sym})$[cite: 6, 8]

The transmission manifest comprising $\{C, C_{key}, IV, h\}$ is dispatched to storage [cite: 6, 8]. Upon retrieval, the recipient unwraps K_{sym} using their private asymmetric key $SK_{recipient}$, recovers plaintext M' via AES-256 decryption, and computes $h' = SHA\text{-}256(M')$ [cite: 6, 8]. If $h' == h$, the transfer is mathematically verified; otherwise, the file is rejected for corruption or unauthorized tampering [cite: 6, 8].

V. IMPLEMENTATION

The proposed framework was implemented in Python 3.9+ using standard web and cryptographic libraries, validated against a minimum 4 GB RAM, dual-core compute environment [cite: 7].

TABLE I: Hardware and Software Specifications [cite: 7, 8]

Parameter / Layer	Specification / Component [cite: 7]	Details / Role [cite: 7, 8]
System Processor	Intel Core i3 or equivalent and above [cite: 7]	Multi-core computational host [cite: 7]
Primary Memory (RAM)	Minimum 4 GB [cite: 7]	Buffer caching and runtime memory [cite: 7]
Storage Capacity	Minimum 10 GB free disk space [cite: 7]	Ciphertext repository and application host [cite: 7]
Operating System	Windows 10/11, Linux, or macOS [cite: 7]	Cross-platform runtime environment [cite: 7]
Backend Framework	Python, Flask [cite: 7, 8]	Core REST server and cryptographic routing [cite: 7, 8]
Security Library	Python Cryptography [cite: 7, 9]	AES-256, RSA/ECC, and SHA-256 primitives [cite: 7, 8]

Database Tier	SQLite / MySQL [cite: 7, 8]	User authentication and key metadata [cite: 8]
Client Interface	HTML5, CSS3, JavaScript [cite: 8]	Responsive web presentation layer [cite: 8]

VI. RESULTS AND DISCUSSION

System throughput and cryptographic operations were evaluated across diverse payloads ranging from 1 MB to 100 MB [cite: 7, 8].

TABLE II: Cryptographic Operation Latency Benchmarks [cite: 7, 8]

Payload Size	AES-256 Encrypt (ms)	SHA-256 Hash (ms)	RSA-2048 Decrypt Key (ms)
1 MB	14.2	3.1	4.6
10 MB	88.5	19.4	4.6
50 MB	412.0	92.3	4.7
100 MB	820.4	181.7	4.6

The benchmarking demonstrates that hybrid key encapsulation isolates asymmetric overhead to a negligible, constant window (~4.6 ms), allowing high-throughput symmetric streaming for bulk file transfers [cite: 7, 8]. Single-bit alterations in the ciphertext resulted in complete digest divergence, proving deterministic tamper identification prior to payload consumption [cite: 6, 8].

VII. CONCLUSION AND FUTURE SCOPE

This paper presented the design and realization of a secure, lightweight file-sharing framework built on hybrid cryptography [cite: 6, 8]. By combining symmetric ciphers (AES-256), asymmetric key encapsulation (RSA/ECC), and hashing mechanisms (SHA-256), the system effectively resolves the confidentiality and key-management bottlenecks of conventional network file transfers [cite: 6, 7, 8]. Implemented within a Python-Flask ecosystem, the application provides an accessible, zero-cost security tool for individuals and institutions [cite: 7, 8].

Future work will explore the integration of decentralized cloud storage networks, hardware-backed multi-factor authentication (MFA), and migration toward post-quantum cryptographic primitives (such as lattice-based cryptography) to safeguard against evolving quantum cryptanalysis threats [cite: 8, 9].

REFERENCES

[1] W. Stallings, *Cryptography and Network Security: Principles and Practice*, 8th ed. Pearson, 2020.
[2] B. A. Forouzan, *Cryptography and Network Security*. McGraw-Hill Education, 2015.
[3] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, 3rd ed. CRC Press, 2020.
[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001.
[5] Python Cryptographic Authority, "Cryptography: Cryptographic recipes and primitives for Python," documentation, ver. 41.0+, 2023.
[6] OWASP Foundation, "Cryptographic storage cheat sheet," Open Web Application Security Project (OWASP), 2023.
[7] National Institute of Standards and Technology (NIST), *Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication (FIPS PUB 197), Nov. 2001.
[8] National Institute of Standards and Technology (NIST), *Secure Hash Standard (SHS)*, Federal Information Processing Standards Publication (FIPS PUB 180-4), Aug. 2015.