JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

A MULTI-ALGORITHM HYBRID CRYPTOGRAPHIC FILE SYSTEM

[1]Diksha Gawand, [2]Trupti Asolkar
[1] MSc CS Specialization in Cybersecurity, [2]Assistant Professor
[1,2]Department of Advanced Computing
[1,2]Nagindas Khandwala College
[1,2]University of Mumbai, Maharashtra, India

ABSTRACT
The increasing use of digital documents and sensitive files has created a need for secure systems that provide both confidentiality and integrity. Conventional file encryption protects data from unauthorized access, but encryption alone does not continuously detect unauthorized modification of stored files. This paper presents a Multi-Algorithm Hybrid Cryptographic File System with Automated Integrity Verification, developed using Python and Streamlit for secure file management. The proposed system combines symmetric encryption for file protection with RSA-2048-OAEP for secure protection of the generated symmetric encryption key. The system supports multiple symmetric encryption algorithms, including AES-256-GCM, ChaCha20-Poly1305, Camellia-256, ARIA-256, Twofish, Serpent, and IDEA. SHA-256 is used to generate integrity hashes for encrypted files. An automated monitoring mechanism periodically recalculates the hash of stored encrypted files and compares it with the previously recorded value. Any mismatch or missing encrypted file is identified as an integrity failure and recorded through the audit logging mechanism. The system also provides controlled decryption after successful integrity verification. The proposed approach integrates multi-algorithm encryption, asymmetric key protection, integrity verification, automated monitoring, tamper detection, and audit logging into a unified file-security system.

Keywords: Hybrid Cryptography, File Security, AES-256-GCM, ChaCha20-Poly1305, Camellia-256, ARIA, Twofish, Serpent, IDEA, RSA-2048-OAEP, SHA-256, File Integrity Monitoring, Tamper Detection, Secure File Storage.

I. INTRODUCTION

Digital files are continuously created, exchanged, and stored across personal computers, organizational systems, cloud environments, and other digital platforms. These files may contain personal, academic, financial, business, and other sensitive information. Unauthorized access, modification, corruption, or deletion of such files can result in information disclosure, loss of data integrity, and operational risks. Therefore, secure file-storage mechanisms must provide appropriate protection against both unauthorized access and unauthorized modification.

Cryptographic techniques provide important mechanisms for protecting digital information. Encryption is primarily used to provide confidentiality by transforming readable information into ciphertext. Symmetric encryption algorithms are particularly suitable for file protection because they provide efficient encryption and decryption of large amounts of data. However, the encryption key itself must also be protected securely. Asymmetric cryptography can be used to protect or exchange symmetric keys without

directly applying computationally expensive public-key encryption to the complete file.

Although encryption protects the confidentiality of stored files, it does not by itself provide continuous visibility into whether an encrypted file has been modified after storage. A file may remain encrypted while its encrypted contents are altered because of accidental corruption or unauthorized activity. Therefore, an additional integrity-verification mechanism is required to determine whether the stored encrypted file remains unchanged.

The proposed system addresses this requirement by combining multiple cryptographic algorithms with automated integrity monitoring. The system uses symmetric encryption to protect file contents, RSA-2048-OAEP to protect the generated symmetric key, and SHA-256 to generate integrity hashes for encrypted files. The stored hash is periodically compared with a newly calculated hash to identify changes in the encrypted file.

The system is implemented using Python and Streamlit and provides a unified workflow for file upload, file analysis, encryption algorithm selection, key generation, encrypted storage, integrity verification, automated monitoring, tamper detection, audit logging, and controlled decryption.

The main contribution of this work is the integration of multi-algorithm encryption, asymmetric key protection, automated file integrity monitoring, tamper detection, and audit logging within a single file-security application.

II. PROBLEM STATEMENT

Traditional file protection mechanisms may provide confidentiality without continuous verification of file integrity. There is a need for a unified file-security system that can encrypt files using cryptographic algorithms, store the protected data securely, verify file integrity using cryptographic hashes, and automatically identify unauthorized modifications.

Therefore, there is a need for an integrated file-security system that can:

- encrypt files using multiple cryptographic algorithms;
- securely protect the generated symmetric encryption key;
- generate and store cryptographic integrity hashes;
- automatically monitor encrypted files;
- detect modification or disappearance of protected files;
- maintain security-related audit records; and
- allow decryption only after successful integrity verification.

The proposed system addresses these requirements through a hybrid cryptographic architecture combining symmetric encryption, RSA-based key protection, SHA-256 integrity verification, automated monitoring, tamper detection

III. OBJECTIVES

The primary objectives of the proposed system are:

1. To design and implement a secure cryptographic file-storage system.
2. To provide multiple encryption algorithms for protecting sensitive files.
3. To generate secure symmetric encryption keys for file protection.
4. To protect symmetric encryption keys using RSA-2048-OAEP.
5. To generate SHA-256 integrity hashes for encrypted files.
6. To automatically monitor stored encrypted files for integrity changes.
7. To detect unauthorized modification, corruption, or missing encrypted files.
8. To maintain integrity-verification records and security audit logs.
9. To prevent normal decryption when the encrypted file fails integrity verification.
10. To provide a user-friendly Streamlit interface for secure file-management operations

IV. LITERATURE REVIEW

Research in secure file storage has investigated combinations of symmetric and asymmetric cryptography, hash-based integrity verification, and secure key management. Symmetric algorithms are generally efficient for encrypting file contents, while asymmetric cryptography can be used for protecting or exchanging cryptographic keys. Hash functions such as SHA-256 and SHA-3 provide a mechanism for detecting changes because a modification to the input produces a different digest.

Recent secure-storage approaches also emphasize integrity verification in addition to confidentiality. The proposed work extends this general approach by integrating encryption, SHA-based verification, automated monitoring, tamper detection, and audit logging into one application-oriented file-security system.

4.1 Research Gap

Existing approaches provide important individual security mechanisms such as encryption, hashing, and key management. However, a complete file-

security workflow requires these mechanisms to operate together. A system that encrypts a file but does not continuously verify its stored state may not detect subsequent modification. Similarly, an integrity mechanism without strong confidentiality and key protection does not provide complete protection for sensitive files.

The research gap addressed by this work is the integration of multi-algorithm file encryption, asymmetric protection of symmetric keys, automated integrity verification, tamper detection, and audit logging into a single application-oriented file-security system.

V. PROPOSED SYSTEM

The proposed system is a Multi-Algorithm Hybrid Cryptographic File System with Automated Integrity Verification designed to provide confidentiality, secure key management, integrity verification, and continuous monitoring of protected files. The system is implemented using Python with Streamlit as the application interface and a database for maintaining file metadata, integrity records, and audit information.

The proposed system follows a hybrid cryptographic approach in which symmetric encryption is used to protect the actual file contents, while asymmetric encryption is used to protect the symmetric encryption key. This approach avoids applying computationally expensive asymmetric encryption directly to complete files while providing a secure mechanism for protecting the generated symmetric key.

When a user uploads a file, the system first performs file analysis and records relevant information such as filename, extension, MIME type, and file size. The user then selects an encryption algorithm from the available cryptographic techniques. A symmetric key is generated and used to encrypt the file. The resulting encrypted file is stored in the protected storage location.

The generated symmetric key is protected using an RSA-2048 public key with OAEP padding and SHA-256. The protected key is stored along with the associated file metadata so that it can later be recovered using the corresponding RSA private key during authorized decryption.

After encryption, a SHA-256 hash is calculated for the encrypted file. This hash is stored as the expected integrity value in the database. The integrity value provides a reference against which future versions of the encrypted file can be verified.

The system incorporates an automated integrity-monitoring mechanism that periodically checks the protected files. During each verification cycle, the current SHA-256 hash of an encrypted file is calculated and compared with its previously stored hash. If both values match, the file is considered to have passed integrity verification. If the values differ, the system identifies a possible modification or corruption of the encrypted file.

The system also handles missing encrypted files. If an encrypted file cannot be found in its expected storage location, the monitoring mechanism records the condition as an integrity failure. Integrity failures are recorded in the integrity log and relevant security events are recorded in the audit log.

For decryption, the system first verifies the integrity of the encrypted file. After successful verification, the protected symmetric key is decrypted using the RSA private key. The recovered symmetric key is then supplied to the corresponding symmetric decryption algorithm to recover the original file.

The complete proposed workflow therefore integrates file analysis, multi-algorithm encryption, symmetric-key generation, RSA-2048-OAEP key protection, secure encrypted storage, SHA-256 integrity verification, automated monitoring, tamper detection, audit logging, and integrity-controlled decryption into a unified file-security system.

VI. SYSTEM ARCHITECTURE

The proposed architecture follows a sequential security workflow. The user first logs into the application and uploads a file. The file is analyzed to obtain basic metadata such as filename, extension, MIME type, and size. The user then selects a supported symmetric encryption algorithm. A symmetric key is generated and used to encrypt the file. The generated symmetric key is protected using the RSA-2048 public key with OAEP padding and SHA-256. The encrypted file, protected key, and associated metadata are stored. A SHA-256 integrity hash is generated for the encrypted file and stored as the expected integrity value. The automated monitoring component periodically calculates the current hash and compares it with the stored value. If the values match, the file is considered verified. If they differ, or if the encrypted file is missing, an integrity failure is recorded and an audit event is generated. Decryption is performed only after the integrity verification succeeds.

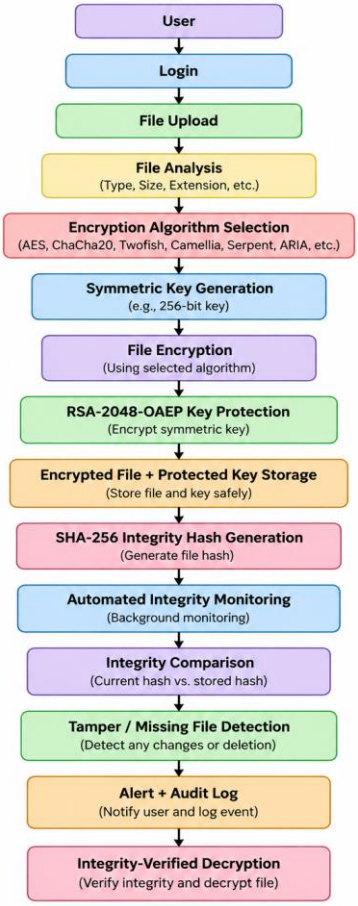

Fig. 6.1 : SYSTEM ARCHITECTURE

VII. METHODOLOGY

7.1 File Upload and Analysis

The user selects a file through the Streamlit interface. Before encryption, the application records relevant metadata including the original filename, extension, MIME type, file size, and stored filename. The metadata is associated with the encrypted file record for subsequent management and verification.

7.2 Symmetric Encryption

The selected symmetric encryption algorithm is used to protect the file contents. A separate symmetric key is generated for the encryption operation. The encrypted output is stored in the application's encrypted-file storage location. In the current implementation, AES-256-GCM and ChaCha20-Poly1305 provide authenticated encryption, while Camellia-256 is implemented with CBC encryption together with an integrity-protection construction.

7.3 RSA-2048-OAEP Key Protection

The generated symmetric key is not stored as plaintext. It is encrypted using the RSA-2048 public key with OAEP padding and SHA-256. The resulting protected key is stored with the file metadata. During decryption, the RSA private key is used to recover the symmetric key before the selected symmetric algorithm decrypts the file.

7.4 SHA-256 Integrity Verification

A SHA-256 digest is calculated for the stored encrypted file after encryption. This digest represents the expected integrity value. During subsequent verification, the application calculates a new SHA-256 digest and compares it with the stored value. Matching values indicate that the encrypted file has not changed since the recorded reference was created; a mismatch indicates a possible modification or corruption.

7.5 Automated Integrity Monitoring

The monitoring component periodically checks the encrypted files associated with the user. Each verification operation records the expected hash, current hash, verification status, and verification time. The monitoring workflow therefore provides a historical record of integrity checks instead of relying only on a one-time manual verification.

7.6 Tamper Detection and Audit Logging

When the current hash differs from the expected hash, the system records an integrity failure and creates an audit event. A missing encrypted file is also treated as an integrity failure. The audit record provides information about the event and supports later review of security-related activity.

7.7 Integrity-Controlled Decryption

Before decryption, the encrypted file is verified against its stored integrity value. When verification succeeds, the protected symmetric key is decrypted using the RSA private key and the original file is recovered using the corresponding symmetric decryption function. This prevents normal decryption from proceeding when the stored encrypted file has failed the integrity check.

VIII. CRYPTOGRAPHIC TECHNIQUES

Technique	Category	Purpose
AES-256	Symmetric encryption	File confidentiality
ChaCha20	Symmetric encryption	File confidentiality
Twofish	Symmetric encryption	File confidentiality
Camellia	Symmetric encryption	File confidentiality
Serpent	Symmetric encryption	File confidentiality
ARIA	Symmetric encryption	File confidentiality
IDEA	Symmetric	File

	encryption	confidentiality
SHA-256	Cryptographic hash	Integrity verification

Table. 8.1: cryptographic techniques

IX. IMPLEMENTATION

The application is developed in Python with Streamlit as the user interface framework. The implementation is organized into components for authentication, file handling, cryptographic processing, database operations, integrity verification, monitoring, and logging. The database stores file-related metadata and security records, while encrypted files and key-related data are maintained in the designated storage structure.

9.1 File Encryption Implementation

When a file is uploaded, the system first extracts its basic metadata, including filename, extension, MIME type, and file size. The user then selects one of the supported symmetric encryption algorithms.

9.2 Hybrid Key Management

The system follows a hybrid encryption model. The selected symmetric algorithm encrypts the actual file, while RSA-2048-OAEP protects the generated symmetric key. RSA-2048-OAEP with SHA-256 is used for protecting the symmetric encryption key rather than encrypting the complete file. This separates bulk-data encryption from key protection and reduces the need to perform asymmetric encryption over large file contents. Hybrid encryption approaches combining symmetric and asymmetric techniques have also been investigated in cloud-storage security research.

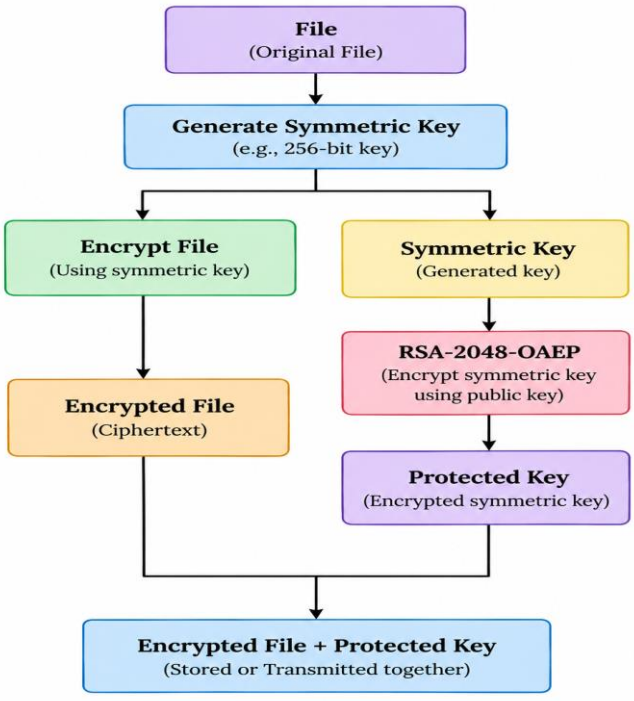

FIG. 9.2 : Hybrid Key Management

9.3 Integrity Verification Implementation

After encryption, a SHA-256 hash is calculated for the encrypted file. The generated hash is stored in the database as the expected integrity value.

During automated verification, the system generates a new SHA-256 hash from the current encrypted file and compares it with the previously stored value.

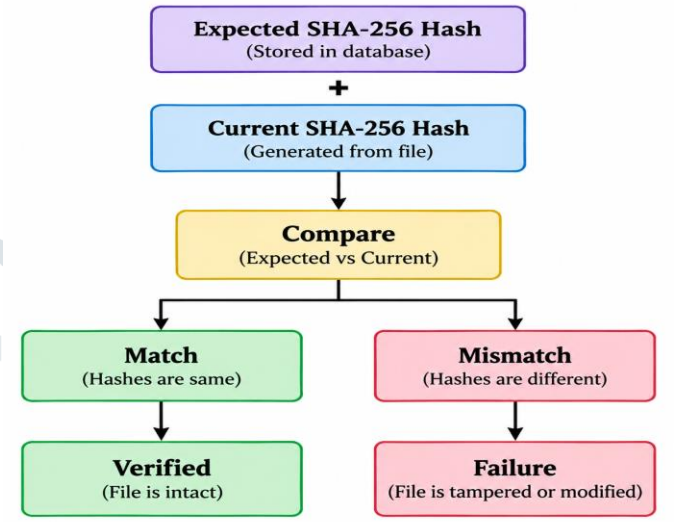

Fig 9.3: Integrity Verification Implementation

A matching value indicates that the file contents have not changed relative to the stored integrity reference. A mismatch indicates possible modification or corruption.

9.4 Audit Logging

The system maintains an audit trail of important security operations. Events recorded by the system include:

- File encryption
- Integrity verification
- Integrity failure
- Key decryption
- File decryption
- Key decryption failure
- File decryption failure

This provides traceability of important cryptographic and integrity-related operations.

X. RESULTS AND DISCUSSION

The completed implementation was evaluated using functional tests covering encryption, key protection, integrity verification, automated monitoring, tamper detection, logging, and decryption.

The results demonstrated that the system successfully integrated confidentiality and integrity mechanisms into a single file-security workflow. Files could be encrypted using the supported symmetric algorithms, while the corresponding

symmetric keys could be protected using RSA-2048-OAEP.

The SHA-256-based integrity mechanism successfully provided a reference value for detecting changes to encrypted files. During normal verification, an unchanged file generated the same hash value as the stored reference. During the controlled tampering experiment, modification of the encrypted file resulted in a hash mismatch and an integrity-failure event.

The automated monitoring mechanism reduced the requirement for manual integrity verification by periodically checking the stored encrypted files. The integrity results were recorded in the database, providing a historical record of verification operations.

The audit-log mechanism further improved traceability by recording important events such as encryption, successful integrity verification, integrity failure, key operations, and decryption operations.

10.1 Functional Results

Function	Expected Result	Observed Result
File Upload	File accepted and analyzed	Successful
AES-256-GCM Encryption	File encrypted	Successful
ChaCha20-Poly1305 Encryption	File encrypted	Successful
Camellia-256 Encryption	File encrypted	Successful
ARIA-256 Encryption	File encrypted	Successful
Twofish Encryption	File encrypted	Successful
Serpent Encryption	File encrypted	Successful
IDEA Encryption	File encrypted	Successful
RSA Key Protection	Symmetric key protected	Successful
SHA-256 Verification	Hash correctly compared	Successful
Automated Monitoring	Periodic checking performed	Successful
Tamper Detection	Modified file detected	Successful
Audit Logging	Security event recorded	Successful
Authorized Decryption	Original file recovered	Successful

Table. 10.1: Functional Results

10.2 Security Results

The implementation provides several security properties:

- **Confidentiality:** File contents are encrypted before protected storage.
- **Key protection:** Symmetric keys are protected using RSA-2048-OAEP.
- **Integrity:** SHA-256 provides a stored reference value for integrity comparison.
- **Tamper detection:** Changes to encrypted files produce hash mismatches.
- **Monitoring:** Files are checked automatically at periodic intervals.
- **Traceability:** Important security events are recorded in audit logs.
- **Controlled decryption:** Integrity verification is performed before decryption.

XI. TEST CASES

Test Case	Test Scenario	Expected Result	Result
TC01	Upload valid file	File uploaded successfully	Pass
TC02	Analyze uploaded file	Metadata displayed/stored	Pass
TC03	Encrypt using AES-256-GCM	Encrypted file generated	Pass
TC04	Protect symmetric key	RSA-protected key generated	Pass
TC05	Generate SHA-256 hash	Hash stored successfully	Pass
TC06	Verify unchanged file	Integrity status shown as verified	Pass
TC07	Modify encrypted file	Integrity failure detected	Pass
TC08	Automated monitoring	Periodic verification performed	Pass
TC09	Decrypt verified file	Original file recovered	Pass

Table 10: Test Cases

XII. ADVANTAGES

• Supports secure storage of encrypted files.

•Provides multiple cryptographic encryption options.

• Uses SHA-based integrity verification.
•Detects unauthorized modification through integrity comparison.
• Provides automated monitoring and tamper alerts.
•Maintains audit/event information for security-related activities.
•Provides a simple web-based interface using Streamlit.

XIII. LIMITATIONS

• The security of the overall system depends on proper protection of cryptographic keys.
• Hash-based verification detects changes but does not by itself identify the person who made the change.
• Monitoring effectiveness depends on the monitoring configuration and system availability.
• Performance can vary with file size and the selected encryption technique.

XIV. FUTURE SCOPE

Future development may include stronger centralized key-management mechanisms, additional authentication protections, broader file-system monitoring, improved alert delivery, secure backup and recovery, and extended benchmarking across different file sizes and algorithms.

XV. CONCLUSION

This paper presents a Hybrid Cryptographic File System with Automated Integrity Verification for protecting sensitive digital files. The system combines encryption, key-based decryption, SHA-based integrity verification, automated monitoring, tamper detection, alerts, and audit logging in a single application. The approach demonstrates how confidentiality and integrity controls can be integrated into a practical file-security workflow. The proposed system can serve as a foundation for further development of secure file-management solutions.

REFERENCES

[1] M. Al-Masri, A. Al-Mahmoud, and others, "Implementation and Performance Analysis of Hybrid Cryptographic Schemes applied in Cloud Computing Environment," *Procedia Computer Science*, vol. 194, pp. 165–172, 2021. DOI: 10.1016/j.procs.2021.10.070.

[2] Y. Zhang, A. V. Vasilakos, and J. Shen, "Cloud data integrity checking with an identity-based auditing mechanism from RSA," *Future Generation Computer Systems*, vol. 62, pp. 85–91, 2016. DOI: 10.1016/j.future.2016.02.003.

[3] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou, "Toward Secure and Dependable Storage Services in Cloud Computing," *IEEE Transactions on Services Computing*, vol. 5, no. 2, pp. 220–232, 2012. DOI: 10.1109/TSC.2011.24.

[4] C. Wang, Q. Wang, K. Ren, and W. Lou, "Enabling Public Auditability and Data Dynamics for Storage Security in Cloud Computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 5, pp. 847–859, 2011. DOI: 10.1109/TPDS.2010.183.

[5] C. Wang, S. S. M. Chow, Q. Wang, K. Ren, and W. Lou, "Privacy-Preserving Public Auditing for Secure Cloud Storage," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 362–375, 2013. DOI: 10.1109/TC.2011.245.

[6] Y. Zhu, H. Wang, Z. Hu, G.-J. Ahn, H. Hu, and S. S. Yau, "Dynamic Audit Services for Integrity Verification of Outsourced Storage in Clouds," *Proceedings of the 2011 ACM Symposium on Applied Computing*, 2011. DOI: 10.1145/1982185.1982514.

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable Data Possession at Untrusted Stores," *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 598–609, 2007. DOI: 10.1145/1315245.1315318.

[8] A. Juels and B. S. Kaliski Jr., "PORs: Proofs of Retrievability for Large Files," *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 584–597, 2007. DOI: 10.1145/1315245.1315317.

[9] Y. Zhang, J. Shen, and others, "One secure data integrity verification scheme for cloud storage," *Future Generation Computer Systems*, vol. 96, pp. 376–385, 2019. DOI: 10.1016/j.future.2019.01.054.

[10] "Full integrity and freshness for cloud data," *Future Generation Computer Systems*, vol. 80, pp. 640–652, 2018. DOI: 10.1016/j.future.2016.06.013.

[11] "An efficient data integrity auditing protocol for cloud computing," *Future Generation Computer Systems*, vol. 109, pp. 306–316, 2020. DOI: 10.1016/j.future.2020.03.032.