International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 11 | Nov - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

Secure Cloud File Storage with Encryption

Prof. D. Nandurkar[1],
Avinash Banote[2], Sakshi Kasture[3], Vanshika Kherde[4], Prasanna Lanje[5], Mitesh Sahare[6]
Department of Computer Science and Engineering-II, Priyadarshini Bhagwati College of Engineering, Nagpur, Maharashtra, India

Abstract
Cloud storage enables convenient, scalable access to digital files, but placing sensitive information on a remote service introduces risks involving unauthorized access, disclosure, and modification. This paper presents the design of a secure cloud file-storage system in which files are encrypted before they are uploaded. The proposed workflow uses AES-256 for bulk file encryption, RSA public-key cryptography to protect the randomly generated AES key, and SHA-256 to calculate a file digest for integrity checking. User authentication and controlled file-management operations provide an additional access layer. The paper describes the system requirements, threat assumptions, architecture, upload and download workflows, key-handling considerations, and a test plan for evaluating confidentiality, integrity, and usability. The design follows a hybrid-cryptography approach: symmetric encryption protects file contents efficiently, while asymmetric cryptography protects the data-encryption key. SHA-256 supports change detection when the expected digest is stored in a trusted location; it does not by itself authenticate the source of a digest. The supplied project material does not contain implementation measurements or experimental results, so no performance figures or claims of completed testing are reported. The paper therefore serves as a design-and-methodology research draft that should be updated with verified implementation details and test results before journal submission.

Keywords— cloud security, secure file storage, AES-256, RSA, SHA-256, hybrid cryptography

I. INTRODUCTION

Cloud computing has changed how individuals and organizations store, share, and retrieve digital information. Cloud services offer on-demand capacity, remote access, and reduced dependence on local storage infrastructure. These benefits are useful for academic records, business documents, personal files, and organizational backups. At the same time, data placed on a remote platform may be exposed through compromised accounts, misconfigured permissions, insecure applications, insider misuse, or a provider-side security incident. Files may also be altered or deleted without the owner's knowledge.

Encryption before upload reduces the need to trust the storage provider with readable file contents. In a client-side encryption model, the file is transformed into ciphertext before it leaves the user's device. The cloud service stores and returns the ciphertext, while decryption keys remain under the control of the authorized user or the system's key-management component. This approach does not remove every risk: account security, key protection, metadata exposure, availability, and secure application implementation remain important.

The project, titled Secure Cloud File Storage with Encryption, proposes a combined security workflow using AES-256, RSA, SHA-256, user authentication, and cloud file management. AES is intended to protect the file data; RSA is intended to protect the AES key; and SHA-256 provides a digest that can help detect changes. The design is aimed at providing confidentiality, integrity checking, and access control through a simple user interface.

The main contribution of this paper is a structured design for the proposed project, including its threat assumptions, cryptographic workflow, functional modules, and a reproducible evaluation plan. Since the supplied project presentation does not report a completed implementation or measured results, this paper distinguishes design expectations from demonstrated outcomes.

A. Problem Statement

Cloud storage makes files available from different devices, but sensitive content can be exposed if access controls fail or the cloud account is compromised. Data may also be modified in transit or while stored. Weak handling of encryption keys can undermine otherwise strong encryption. A secure storage workflow is therefore required in which files are encrypted before upload, encryption keys are protected separately, user access is authenticated, and integrity is checked when files are retrieved.

B. Aim and Objectives

The aim is to design a secure cloud file-storage system that protects user files from unauthorized disclosure and supports detection of file modification. The objectives are:

- Encrypt files locally with AES-256 before sending them to cloud storage.
- Protect the AES data-encryption key using RSA public-key encryption.
- Calculate a SHA-256 digest to support file-integrity checking.
- Require user authentication before file-management operations.
- Support secure upload, storage, download, verification, and decryption workflows.

- Define tests that can evaluate correctness, security properties, and usability without inventing performance outcomes.

C. Scope and Assumptions

The proposed system covers authenticated file upload and retrieval, encryption before cloud storage, protection of the data-encryption key, and integrity verification. The cloud is treated as a storage service that may be exposed to unauthorized inspection or accidental modification; it is not assumed to be fully trusted with plaintext. The design assumes that users' endpoints and cryptographic libraries are not already compromised. It does not claim to prevent denial-of-service, guarantee cloud availability, conceal all file metadata, or protect a file after it has been decrypted on an infected device.

Cloud-storage security can be considered as a layered problem rather than a single encryption task. The storage layer must preserve the availability of objects, the application layer must enforce identities and permissions, and the cryptographic layer must protect the content and keys. A failure in any one layer can weaken the overall system. For example, strong file encryption does not prevent an attacker from deleting ciphertext, and secure authentication does not guarantee confidentiality if plaintext files are uploaded directly. This layered perspective informs the separation of modules in the proposed architecture.

An important distinction in reviewing cryptographic systems is the difference between a hash function and a message-authentication mechanism. SHA-256 produces a digest that can be used to compare data against a trusted expected value, but it does not use a secret key and does not prove who created the digest. Authenticated encryption, a message authentication code, or a digital signature can provide stronger guarantees depending on the threat model. Therefore, the SHA-256 component in this project should be treated as an integrity-checking aid, while the final implementation should ensure that an attacker cannot replace both the stored file and its reference digest without detection.

II. LITERATURE REVIEW

Hybrid cryptography combines symmetric encryption for bulk data with asymmetric cryptography for key protection. This arrangement avoids using a computationally expensive public-key operation on every block of a large file. Saginayev et al. describe a cloud-storage approach that combines AES-256 for file encryption, RSA for key management, and SHA-256 for integrity verification [1]. Their work is closely related to the cryptographic choices in the present project and supports using separate mechanisms for content encryption, key protection, and integrity checking.

Lai and Heng present a cloud file-storage approach that combines AES with ElGamal-based key protection, a SHA-2 integrity check, and two-factor authentication [2]. Their study highlights that encryption alone is not sufficient if an account can be taken over; authentication is an important complementary control. The present project includes user authentication, while multi-factor authentication is considered a possible improvement rather than a confirmed implemented feature.

Liu et al. investigate a key-aggregate cryptosystem for selective group data sharing with user revocation in cloud storage [3]. This work addresses more advanced sharing and revocation requirements than the initial project scope. It suggests that future versions of a file-storage system should define permissions, sharing lifetimes, and revocation behavior explicitly rather than treating every authenticated user as having identical access.

Selvi and Sakthivel describe a hybrid ECC-AES framework for secure cloud-based data protection [4]. Their work explores elliptic-curve cryptography as an alternative approach to public-key operations in a hybrid design. The current project retains RSA because it is specified in the supplied project material; comparing RSA-based key wrapping with alternative key-establishment approaches would require a controlled evaluation.

These studies indicate that a secure cloud-storage design should address several related but distinct concerns: confidentiality of file contents, protection of cryptographic keys, integrity verification, authentication, and sharing permissions. The proposed project brings the first four into a single workflow. Its current limitation is that the presentation does not provide implementation details, test datasets, measurements, or a detailed access-control model. The evaluation plan in this paper is intended to address these gaps during implementation.

The system's data flow should minimize the time during which plaintext exists outside the user's control. File encryption should take place on the client before network transfer, and temporary plaintext copies should be removed safely where practical. Error messages should avoid including file contents, encryption keys, private-key material, or sensitive metadata. If a backend service performs encryption instead of the client, the design must explicitly identify that trust boundary because the backend would then handle plaintext and key material.

Each uploaded file should be associated with a unique identifier rather than relying only on its visible filename. The metadata record should identify the cryptographic algorithm and mode, nonce or initialization value, wrapped key, ciphertext size, and version. If a file is updated, the system should create a new encryption version with a fresh key or a carefully defined key-rotation policy. Reusing a nonce where the selected encryption mode prohibits it can compromise security; the implementation should use a cryptographic library's recommended nonce-

generation procedure and validate uniqueness requirements.

Algorithm 1 summarizes the intended upload operation in implementation-neutral form. First, authenticate the user and check permission to create an object. Second, read the selected file and generate a fresh random data-encryption key. Third, encrypt the file using the selected AES-256 mode and retain all required nonce and authentication-tag values. Fourth, protect the AES key using the authorized RSA public key and a secure padding scheme. Fifth, calculate the integrity digest over the precisely documented byte sequence and store the expected value in access-controlled metadata. Finally, upload the ciphertext and associated metadata over TLS. The application should report success only after the cloud service confirms that the object and its metadata were stored.

Algorithm 2 summarizes retrieval. The user authenticates and requests a file identifier; the application checks ownership or sharing permissions before returning any object. The ciphertext and metadata are downloaded over TLS. The system validates metadata structure and verifies the integrity information available to it. The user's authorized private key then unwraps the AES key, and the local cryptographic module decrypts the ciphertext. If the authenticated-encryption tag or integrity check fails, the process terminates with an error and does not treat the output as a valid file. Logs should record the event without exposing the key or plaintext.

Unlike the existing approaches discussed in the literature, the proposed system focuses on combining AES-256 file encryption, RSA-based encryption key protection, SHA-256 integrity checking, and user authentication within a single cloud file-storage workflow. The proposed design emphasizes client-side encryption so that files are protected before being uploaded to cloud storage. It also defines a structured evaluation plan to assess confidentiality, integrity, access control, and performance after implementation.

III. METHODOLOGY

A. System Architecture

The proposed system contains a user interface, an authentication component, a client-side encryption module, a key-protection module, a cloud-storage interface, and a verification/decryption module. The client-side cryptographic modules should process files before upload so that the cloud receives ciphertext rather than plaintext. The storage service holds the encrypted file and the associated protected key and metadata. The authenticated user initiates retrieval; the system downloads the ciphertext, checks integrity, unwraps the AES key, and decrypts the file only after verification succeeds.

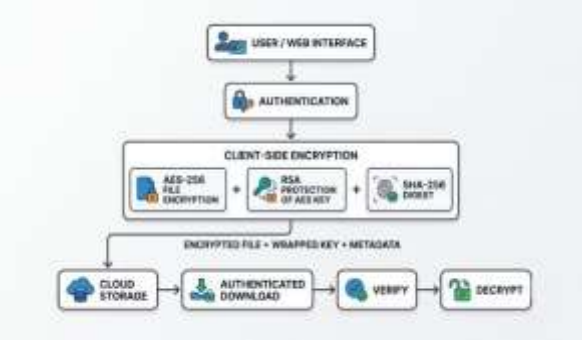


Fig. 1. Proposed logical architecture of the secure cloud file-storage system.

B. Functional Modules

1) User authentication: The application validates user credentials before allowing upload, listing, download, deletion, or sharing actions. Passwords should be stored as salted password hashes using a dedicated password-hashing algorithm, not as plaintext or as a plain SHA-256 digest. Sessions should expire appropriately, and repeated failed logins should be rate-limited.

2) File selection and metadata: The user selects a file and initiates upload. The system records only necessary metadata, such as an opaque file identifier, original filename if needed, size, upload time, ciphertext location, wrapped key, and integrity information. Metadata should be access-controlled because filenames and timestamps can themselves reveal sensitive information.

3) File encryption: A fresh, cryptographically secure random AES key is generated for each file or upload version. The file is encrypted locally before upload. For a production-quality implementation, an authenticated encryption mode such as AES-GCM is preferable to an unauthenticated mode because it detects ciphertext alteration during decryption. If the implementation must follow the project's stated AES-256 choice, the mode, nonce handling, and authentication tag must be specified and implemented correctly [5].

4) Key protection: The AES key is encrypted using the intended recipient's RSA public key with a secure padding scheme such as RSA-OAEP [7]. The matching private key is required to recover the AES key. Private keys must not be uploaded in plaintext alongside the encrypted files. Key backup, recovery, rotation, and loss handling need a documented policy.

5) Integrity verification: SHA-256 computes a 256-bit digest of a defined byte sequence [6]. The design must specify whether the digest is calculated over the original plaintext, ciphertext, or both. During retrieval, the recalculated digest is compared with the expected digest. The expected digest must be protected from unauthorized replacement; a digest stored beside a file in the same writable location is not sufficient against an attacker who can replace both. Authenticated encryption can additionally detect ciphertext modification.

6) Cloud storage and retrieval: Only ciphertext, the wrapped AES key, and the minimum necessary metadata are sent to the storage service. Retrieval requires authentication and authorization. After download, the system verifies the file, recovers the AES key through RSA, and decrypts the file locally. Verification failure should stop the download from being presented as trusted plaintext and should produce a clear warning.

7) File management and sharing: The interface should list the user's files and provide upload, download, delete, and (if implemented) share operations. Sharing requires a defined recipient-key process. Simply sending an encrypted file to another user does not grant access unless that recipient can securely obtain the appropriate decryption key.

C. Cryptographic Workflow

Upload workflow: (i) authenticate the user; (ii) select a file; (iii) generate a fresh random AES-256 key and a unique nonce/initialization value required by the chosen mode; (iv) encrypt the file locally; (v) encrypt or wrap the AES key using the authorized RSA public key and an appropriate padding scheme; (vi) calculate and store the defined SHA-256 digest in a protected metadata record; and (vii) upload the ciphertext, wrapped key, nonce, authentication tag where applicable, and required metadata.

Download workflow: (i) authenticate the user and authorize access to the file; (ii) download ciphertext and associated metadata; (iii) verify the expected digest and, when using authenticated encryption, verify the authentication tag; (iv) use the authorized RSA private key to recover the AES key; (v) decrypt locally; and (vi) report success only if verification and decryption complete correctly. The implementation should fail closed if key recovery, authentication, or integrity verification fails.

RSA should not be used to encrypt an entire large file. Its role in this design is to protect a small AES key. The implementation must use a maintained cryptographic library and avoid designing custom cryptographic algorithms. Key generation, random-number generation, nonce uniqueness, and error handling are security-critical parts of the implementation.

D. Security Requirements and Threat Model

The main security goals are confidentiality, integrity, and controlled access. Confidentiality means that a person who obtains only the cloud-stored ciphertext should not be able to read the file without the required key. Integrity means unauthorized changes should be detected under the system's stated assumptions. Access control means file operations are limited to authenticated and authorized users.

The threat model includes an attacker who can inspect cloud-stored objects, obtain a copy of ciphertext, attempt to access another user's file, or alter stored bytes or metadata. The design also considers accidental corruption and compromised account passwords. It does not assume protection against a fully compromised client device, stolen plaintext after decryption, or a malicious administrator who also obtains the user's private key. These limits should be made clear in project documentation.

Important controls include TLS for data in transit, strong authentication, least-privilege cloud permissions, server-side validation, protected private keys, secure secret storage, input and file-size validation, audit records that avoid logging sensitive content, and routine dependency updates. Authentication and encryption solve different problems: a logged-in user must still be authorized to access a particular file, and a cloud account permission does not replace client-side encryption.

E. Proposed Evaluation Plan

Because no implementation logs or measured results were included in the supplied project presentation, evaluation is specified as a plan rather than reported as completed work. Tests should be performed only after the system is implemented, using synthetic or non-sensitive test files.

- Functional correctness: upload, store, retrieve, decrypt, delete, and authorized-sharing flows behave as intended.
- Confidentiality check: inspect stored objects to confirm that file contents are ciphertext and that the AES key is not stored in plaintext.
- Integrity test: alter a copy of stored ciphertext or metadata and verify that the system detects the change or rejects authenticated decryption.
- Access-control test: attempt access using an unauthenticated session and a different user's account; confirm denial.
- Key-handling test: verify that private keys are not exposed in cloud objects, logs, or client error messages.
- Performance measurement: for several file sizes, record upload/download duration, encryption/decryption duration, CPU and memory use, and file sizes before and after encryption.
- Usability test: record whether users can complete upload and retrieval correctly and understand verification errors.

For performance measurements, the same machine, cloud region, network conditions, software versions, and test-file set should be documented. Each measurement should be repeated and summarized using a mean or median with the number of runs stated. No numerical result should be included unless it is produced by the implemented system and can be reproduced.

A useful evaluation should report both functional correctness and operational cost. Functional tests determine whether the application preserves the file contents across a complete upload-download cycle and rejects invalid access. Security-oriented tests examine

whether altered ciphertext, incorrect keys, corrupted metadata, and unauthorized requests are rejected. Performance tests should measure encryption and decryption separately from network transfer so that local cryptographic overhead is not confused with cloud latency. The evaluation should also report the hardware, operating system, programming language, cryptographic library version, cloud service or local test environment, network conditions, file types, file sizes, and number of repetitions.

At least three classes of test files are recommended: small text documents, medium-size mixed files, and larger binary files. For each file, calculate a reference digest before upload and compare the recovered file with the original after download. A matching digest supports byte-for-byte equality for the tested file, but it does not alone establish that every security property is satisfied. The test report should include negative cases, such as a modified ciphertext or a user attempting to access a file owned by another account. Results should be presented in a compact table with columns for test identifier, procedure, expected outcome, observed outcome, and pass/fail status.

IV. RESULTS

The supplied project presentation defines the intended cryptographic components and workflow but does not provide source code, screenshots of a working implementation, test logs, datasets, or measured performance values. Consequently, this draft does not claim that encryption, RSA key protection, integrity verification, or cloud upload has been experimentally validated. The current deliverable is a system design and evaluation specification.

At the design level, the workflow separates the protection of file contents from the protection of encryption keys and includes a defined integrity-checking step. These are properties of the proposed architecture, not proof that a completed application meets them. Implementation evidence should be added here after functional and security tests are performed. A final version should include a table of test cases with expected and observed outcomes, and a performance table containing actual measurements.

TABLE I. PLANNED VALIDATION TESTS

Test	Expected outcome	Status
Valid upload/download	File is recovered correctly after decryption	Not tested
Ciphertext modification	Integrity/authentication check rejects altered data	Not tested
Unauthorized account	Access is denied	Not tested
Missing/invalid key	Decryption fails safely; no plaintext is released	Not tested
Performance by file size	Timings and resource use are recorded	Not tested

V. DISCUSSION

The proposed architecture is practical as a baseline for a student cloud-security project because it separates the tasks of file encryption, key protection, authentication, storage, and integrity verification. Hybrid encryption is suitable for files of varying sizes because AES handles the bulk data while RSA protects the relatively small AES key. The architecture also allows the cloud-storage provider to remain independent of the chosen encryption library, provided the provider can store and return binary ciphertext objects and metadata.

Several design choices require care before the system can be described as secure. First, AES-256 alone does not specify the mode of operation. A production implementation should use a well-reviewed authenticated-encryption mode and correctly manage nonces. Second, RSA must use a secure padding scheme and an appropriate key size; textbook RSA is not suitable. Third, SHA-256 detects a difference only when the expected digest is trusted. If an attacker can replace both the stored file and its unprotected digest, the comparison may not reveal the attack. Fourth, account authentication must be paired with file-level authorization and secure session management.

Key lifecycle management is another limitation. Loss of the RSA private key may make stored files unrecoverable, while exposure of that key may compromise all files whose AES keys it protects. The implementation should document how private keys are generated, stored, backed up, rotated, and revoked. Recovery mechanisms must be designed carefully so that convenience does not create an undocumented path around encryption.

Scalability also depends on how the application handles large files and concurrent users. Reading an entire file into memory may be acceptable for a small demonstration but can become inefficient for larger objects. A production-oriented design may use authenticated chunked encryption, with each chunk protected by a construction that prevents reordering, truncation, or substitution. Such a change must be designed carefully because simply encrypting independent chunks without binding their sequence and context can introduce integrity gaps. The current project description does not state whether streaming or chunked processing has been implemented, so this remains a design consideration.

Availability and recovery deserve separate treatment from confidentiality. Encryption can protect content while leaving the file unavailable if a storage account is suspended, an object is deleted, or a private key is lost. A practical deployment should define backup and restoration procedures for ciphertext and key material, test recovery, and establish who is authorized to perform recovery. Backup copies of private keys require strong protection and access auditing. These operational requirements should be documented alongside the cryptographic design because users may otherwise mistake encryption for a complete backup or disaster-recovery solution.

The project presentation lists secure file sharing as an advantage, but a sharing mechanism is not fully specified. To support sharing safely, the system needs

a method to wrap the file's AES key for each authorized recipient, or another documented key-distribution mechanism. It should also define what happens when a recipient loses access or is removed. Role-based access control, multi-factor authentication, audit logging, and multi-cloud storage are reasonable extensions, but they should be reported as future work unless they are implemented and tested.

The main limitation of this draft is the absence of implementation evidence. The proposed design can guide development, but journal-ready claims about security, usability, or efficiency require repeatable tests and an explicit comparison with relevant baselines. A careful final paper should report both positive and negative findings, describe the test environment, and avoid presenting expected behavior as an observed result.

VI. CONCLUSION

This paper presents a design for Secure Cloud File Storage with Encryption using a hybrid cryptographic workflow. The proposed system encrypts file contents with AES-256 before cloud upload, protects the AES key using RSA, computes a SHA-256 digest for integrity checking, and requires user authentication for file operations. The architecture separates client-side cryptographic processing from remote storage and outlines upload, download, verification, and decryption steps.

The design can provide a foundation for a secure file-storage application, but its actual security depends on correct algorithm configuration, key management, authentication, authorization, and implementation quality. SHA-256 must be paired with trusted digest storage or authenticated encryption to provide meaningful tamper detection. Future work includes implementation, functional and adversarial testing, measured performance evaluation, recipient-based sharing, role-based permissions, multi-factor authentication, and mobile access. Actual results must be added before the manuscript is represented as an experimentally evaluated system.

ACKNOWLEDGMENT

The authors acknowledge the guidance of Prof. D. Nandurkar and the Department of Computer Science and Engineering-II, Priyadarshini Bhagwati College of Engineering, Nagpur, for support of the project.

REFERENCES

[1] E. Saginayev, A. Niyazov, A. Razaque, Zh. Kalpeyeva, and A. Urazgaliyeva, “Secure File Storage on Cloud Using Hybrid Cryptography,” Computing & Engineering, vol. 2, no. 2, pp. 1–7, 2024, doi: 10.51301/ce.2024.i2.01.

[2] J.-F. Lai and S.-H. Heng, “Secure File Storage on Cloud Using Hybrid Cryptography,” Journal of Informatics and Web Engineering, vol. 1, no. 2, pp. 1–18, 2022, doi: 10.33093/jiwe.2022.1.2.1.

[3] J. Liu, J. Qin, X. Zhang, and H. Wang, “Efficient Key-Aggregate Cryptosystem With User Revocation for Selective Group Data Sharing in Cloud Storage,” IEEE Transactions on Knowledge and Data Engineering, vol. 36, no. 11, pp. 6042–6055, 2024, doi: 10.1109/TKDE.2024.3397721.

[4] P. Selvi and S. Sakthivel, “A hybrid ECC-AES encryption framework for secure and efficient cloud-based data protection,” Scientific Reports, vol. 15, Art. no. 30867, 2025, doi: 10.1038/s41598-025-01315-5.

[5] National Institute of Standards and Technology, Advanced Encryption Standard (AES), FIPS PUB 197, updated May 2023. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.197

[6] National Institute of Standards and Technology, Secure Hash Standard (SHS), FIPS PUB 180-4, Aug. 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.180-4

[7] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, “PKCS #1: RSA Cryptography Specifications Version 2.2,” RFC 8017, Internet Engineering Task Force, Nov. 2016. [Online]. Available: https://doi.org/10.17487/RFC8017