SecureShare: Secure File Sharing Using Hybrid Cryptography, Client-Side Data Loss Prevention and Hash-Chained Audit Logs

Dr. C. Shyamala[1], Hemchudesh A[2], Manivarman A N[3], and Prabukumar V[4]
[1]*Associate Professor, Department of Information Technology K. Ramakrishnan College of Technology, Trichy, Tamil Nadu, India*
[2,3,4]*Student, Department of Information Technology K. Ramakrishnan College of Technology, Trichy, Tamil Nadu, India*

***Abstract*—At most companies and college departments, files still move through an ordinary cloud drive, and whoever runs that drive can open whatever sits inside it. SecureShare is a web application we built for a narrower case: a single organization, such as a company team, a research lab or a college club, where one administrator already knows everyone involved. Each file is cut into 64 KB chunks and encrypted with a fresh 256-bit AES key, and that key is then wrapped separately for every recipient under a 2048-bit RSA key. The uploader also signs a SHA-256 digest of the file, and a download will not start until that signature checks out. Private keys never sit on disk unencrypted, and the things that matter — uploads, downloads, approvals, policy changes — are written into a hash-chained audit log an administrator can verify from the dashboard. Before anything is sent, a scanner running in the browser itself looks for personal data, financial data, passwords and cloud keys, so the server is never handed plaintext just to scan it. What follows is the design, the threat model we worked from, the limits we are aware of, and results from a working Flask implementation we tested on a development machine.**

***Index Terms*—Audit log, client-side data loss prevention, digital signature, hybrid cryptography, organization-scoped access control, secure file sharing, encrypted-at-rest storage.**

I. INTRODUCTION

Most people hand their files to a cloud drive without asking who else can open them. The provider keeps those files safe in transit and encrypts them on disk, but it still holds the keys. A leaked backup, or a misconfigured bucket, and files users assumed were private are suddenly not. Encrypting on the client side, so the provider only sees ciphertext, is not a new idea [1] — it is just surprisingly easy to get wrong, as attacks on a well-known commercial service showed [2].

SecureShare targets a smaller case than the public cloud: a company team or a college club, where one administrator already knows who belongs. Files go to named people rather than to whoever holds a link, and the boundary of the organization is something the code enforces, not a policy document nobody reads.

Two things about existing encrypted-sharing tools bothered us. One is integrity — a checksum next to a download proves little when the same person who can swap the file can swap the checksum too, so every download here checks a signature the uploader actually produced. The other is accidental leakage, where passwords, card numbers or cloud keys end up in a shared file. The usual fix is a DLP gateway [3], but it has to read the file to do its job, which quietly undoes the point of encrypting it. So we put the scan in the browser instead.

In short, this paper contributes the following.

1) Hybrid encryption. Files are split into 64 KB chunks, a key is wrapped per recipient, and nothing reaches a downloader until the signature is verified.
2) Organization-scoped access. Admin-approved membership, with revocation, expiry and download limits enforced per recipient in the service layer.
3) A verifiable audit log. A hash chain an administrator can check directly from the dashboard, rather than take on faith.

4) Browser-side DLP. A scanner that applies the organization's own rules locally and sends the server only metadata, never file content.

II. RELATED WORK

Kamara and Lauter [1] laid out cloud storage where the client encrypts before anything is uploaded. The MEGA analysis [2] shows this architecture alone guarantees nothing — the actual failures were in key handling, chosen modes and a few weaker building blocks. That is part of why we list every parameter in Table I and state what the design does not protect against (Section X).

Keys are wrapped with RSA [4] and OAEP [5], and signatures use PSS [6]. File contents use AES [7] in CBC mode [8]. CBC only gives confidentiality, and padding errors can leak information [9], so plaintext is never released until the signature check passes. The audit log is one chain of SHA-256 digests [10].

Most DLP surveys [3] look at content already on a machine the organization controls. SecureShare scans earlier, in the browser, before the file is encrypted. An isolation-forest anomaly module appears in later screenshots but was not part of what we set out to contribute here.

III. THREAT MODEL AND DESIGN GOALS

We built the design around five situations: someone getting hold of the database and storage folder, say from a stolen backup; a signed-in member, or an outsider, reading files nobody shared with them; someone swapping a stored ciphertext; an administrator editing history to cover their tracks; and an uploader sharing something sensitive by mistake.

We assume the server behaves honestly while running, and that TLS protects data in transit. The server briefly holds plaintext during an upload, and can unlock a signed-in user's private key, so a server turned malicious at either moment could read what passes through it — this is not a zero-knowledge design against a malicious server. What it does guarantee is narrower: nothing stored at rest can be decrypted without a password. Infected devices, denial of service and traffic analysis are not in scope.

Those five situations are where the goals come from: keep stored data confidential, make every delivered file intact and genuinely from its sender, keep organizations separate, make the log hard to edit quietly, and warn people early when a file looks sensitive.

IV. SYSTEM ARCHITECTURE

A. Components

SecureShare is a Flask app, built from a factory function, create_app(). It registers the auth, file and admin blueprints, plus three extensions — Flask-SQLAlchemy for the database, Flask-WTF for CSRF tokens, Flask-Limiter for rate limiting. The routes stay thin; the real work sits in a services layer covering crypto, file upload and download, access control, authentication, logging and DLP. The scanner is a JavaScript file running in the browser, and everything else goes over HTTPS.

B. Storage Layout

Two separate SQLite databases hold everything. One handles authentication: users, organizations, join requests, audit entries, login records, DLP policies. The other handles files: metadata, one share row per recipient, and the access and verification logs. Ciphertext sits under random UUID names on disk, and the real file name is kept only in the file database, so leaking one database does not hand over the other. Table I lists the parameters.

TABLE I SECURITY PARAMETERS USED BY SECURESHARE

Item	Setting
File cipher	AES-256-CBC, fresh random key and IV for every file
Key wrapping	RSA-2048 with OAEP and SHA-256, one wrapped key per recipient
Signature	RSA-PSS with MGF1-SHA-256 over the SHA-256 digest of the plaintext
Private key at rest	AES-256-CBC under a PBKDF2-HMAC-SHA256 key, 390,000 iterations, random 16-byte salt
One-time password	6 digits, SHA-256 hashed, 5 min expiry, 5 attempts, 60 s resend wait
Account lockout	15 min after 10 failed logins

Item	Setting
Session	30 min lifetime, HttpOnly, SameSite=Lax, CSRF tokens

C. Sessions and Transport

Sessions are signed with a key pulled from the environment, last 30 minutes, and use the HttpOnly and SameSite=Lax cookie flags. Any form that changes data needs a CSRF token. E-mail is used for exactly one thing, sending one-time passwords, over SMTP with STARTTLS.

V. CRYPTOGRAPHIC DESIGN

A. Per-File Hybrid Encryption

For each upload the server generates a fresh AES-256 key K_f and a random 128-bit IV, then encrypts the file in CBC mode, 64 KB at a time, using one cipher context so the chain carries across chunks. K_f is wrapped separately for every recipient, and the uploader is always counted among them:

$W_r = E_{pk}r(K_f)$ (1)

E here is RSA-2048 with OAEP and SHA-256 [5]. Each W_r lives in its own share row, so no secret is shared across recipients, and removing one person's access just means marking that one row.

B. Signing and Verification

While the file is being encrypted, the server also hashes the plaintext, H(M) with SHA-256, and the uploader's private key signs that digest with RSA-PSS, using MGF1 and SHA-256:

$\sigma = S_{sk}s(H(M))$ (2)

The signature is stored in Base64 with the file record. On download, the server recovers the plaintext, hashes it again to get M′, and only continues if

$V_{pk}s(H(M'), \sigma) = 1$ (3)

If that check fails, the transfer stops, SIGNATURE_INVALID is logged, and a verification entry is written. One check catches a changed ciphertext, a swapped file, or a wrong sender, because the signature covers the plaintext hash directly.

C. Protecting Private Keys

Every user gets an RSA-2048 key pair at registration. The public key is stored as-is; the private key is encrypted under a key derived from the password with PBKDF2-HMAC-SHA-256 [11]:

$k = PBKDF2(pw, s, N)$ (4)

pw is the password, s a random 16-byte salt, and N = 390,000 iterations. The PEM-encoded private key sk is then encrypted as

$C = E_k(sk)$ (5)

That high iteration count is deliberate — it makes offline guessing against a stolen database noticeably more expensive. Login is checked against a separate bcrypt hash [12], so the two salts are unrelated, and changing a password just re-encrypts the existing private key.

D. Streaming Large Files

Files never sit fully in memory — use depends only on the 64 KB chunk size, which is why our test deployment could push a 6.8 GB video through the same pipeline as a small document. Plaintext is written first to a temporary file, then removed in a finally block even if encryption fails. That temporary file is the only point where plaintext touches the server's disk.

A download goes through two passes. The first unwraps the key and decrypts in memory just to recompute the hash and check the signature; only then does a second pass stream the chunks to the client. It costs more CPU time, but no unverified plaintext is ever sent.

E. On-Demand Verification

A recipient can verify a file without downloading it, and each check is logged, valid or not.

VI. ACCESS CONTROL, AUTHENTICATION AND AUDITING

A. Organization Scoping

Every user belongs to exactly one organization. Someone can register as an individual, start a new organization and become its admin, or ask to join one, and that join request waits for an admin of the same organization to approve it. This check lives in the access-control module itself, not a page template, so a crafted request cannot route around it. Recipients must be approved members of the sender's organization.

A download is refused unless a share row exists, has not been revoked or expired, the download count is under its limit, and the sender belongs to the requester's organization. Every attempt is logged for the uploader and administrators to read.

B. Registration and Login

Registration happens in two steps: the form is validated and a one-time password is e-mailed, then nothing is written to the database until that code is entered. The code is a SHA-256 hash, expires after five minutes, and allows five attempts.

Ten failed password attempts lock an account for 15 minutes. Flask-Limiter also caps sensitive actions, for example 10 logins per minute and 3 private-key downloads per hour, and the wrong account type at sign-in is logged as LOGIN_TYPE_MISMATCH.

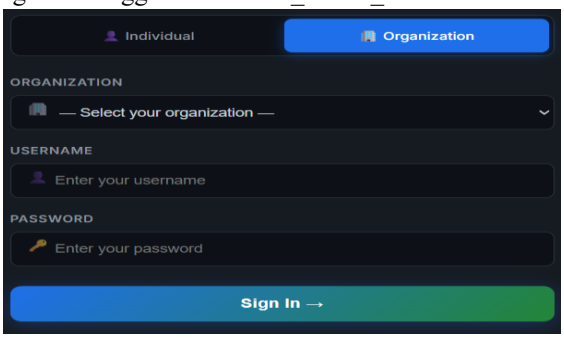


Fig. 1. Organization sign-in page, with the account-type switch.

C. Hash-Chained Audit Log

Logins, uploads, downloads, key views, approvals, removals, policy changes and DLP results all go into one table. Each row stores its fields plus the hash of the row before it. Writing m_i for the concatenated fields of row i, the hash stored with that row is

$$h_i = H(m_i \,\|\, h_{i-1}) \quad (6)$$

where h_0 = "GENESIS" and H is SHA-256. Edit or delete a row in the middle and every link after it breaks. Verification replays the table, recomputes each hash, and reports the first mismatch — though it cannot tell you if the newest rows were simply deleted, since nothing follows them to point back (Section X). Admins only ever see rows from their own organization.

VII. CLIENT-SIDE DATA LOSS PREVENTION

A. Why the Scan Runs in the Browser

A scanner on the server would see each file right before encryption, which defeats the confidentiality goal. Running it in the browser avoids that. The server exposes two small endpoints: one returns the policy, the other accepts a summary that never includes file content, matched text or line numbers.

B. Scanning

The scanner works through a file in 512 KB slices, running precompiled regular expressions on each. Anything over 100 MB is reported as too large to scan. Only plain-text formats are actually scanned — .txt, .csv, .json, .log, .env and similar — everything else is marked NOT_SCANNED and treated as a warning. Table II lists the four categories; some patterns target India specifically, such as Aadhaar-style numbers and PAN strings.

TABLE II DETECTION CATEGORIES OF THE BROWSER-SIDE SCANNER AND THE DEFAULT POLICY

Category	What is matched	Default action
PII	E-mail addresses; international phone numbers (+country code); Indian mobile numbers; Aadhaar-like 12-digit numbers; PAN strings	WARN
Financial	Payment card numbers of 13 to 19 digits; IBAN; IFSC codes	BLOCK
Credential	Password, API key, secret and token assignments; JSON Web Tokens; PEM private-key headers; generic environment secrets	BLOCK
Cloud	AWS access key IDs (AKIA…) and secret keys; GCP service-account JSON; Azure storage connection strings	BLOCK

C. Classification and Policy

Every scan also gets a sensitivity level: LOW when nothing is found, MEDIUM for personal data alone or HIGH above five matches, HIGH for any financial, credential or cloud match, and CRITICAL when two high-risk categories appear together. That level is a summary, though — the real decision comes from the policy. With D the categories found in file F, and p(c) the action assigned to category c (ordered ALLOW < WARN < BLOCK), the result is

$$a(F) = \max_{c \in D} p(c) \quad (7)$$

A switched-off rule counts as ALLOW, and a category with no rule falls back to the sensitivity level: LOW allows, MEDIUM warns, HIGH and CRITICAL block. NOT_SCANNED results and scan errors both count as WARN too, so a broken scanner never passes something through silently.

D. Gating the Upload

Pressing upload fetches the policy and runs the scan under a spinner. ALLOW submits at once. WARN lists the categories found and asks the user to confirm anyway. BLOCK shows a red banner naming the cause and keeps the file on the user's machine. Admins pick one action per category and can toggle rules from the DLP tab (Section VIII); by default personal data gets WARN and the rest get BLOCK, and changes are logged as DLP_POLICY_UPDATED.

None of this is real enforcement — it is a safety net. A determined user can bypass a check that runs on hardware they control. An upload with no matching scan entry in the audit log would, at least, stand out as odd.

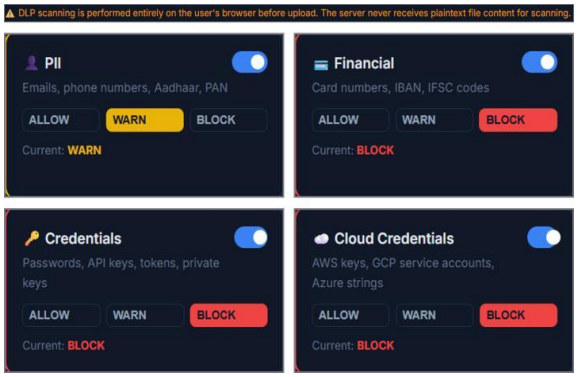


Fig. 2. DLP tab of the admin panel: warn on personal data, block the other categories.

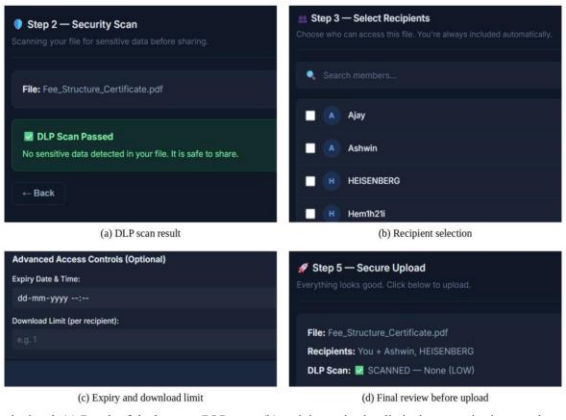
VIII. IMPLEMENTATION AND RESULTS

A. Setup

The server is Python, Flask and SQLite, and the scanner is plain ES6 JavaScript. We ran the system on a development machine with one test organization, LockByte, of 16 members and 16 managed files. We ran no load test, so we make no speed claims.

B. Upload Workflow

The upload wizard has five steps: pick the file, run the browser scan, choose recipients, review the protections with an optional expiry and download limit, and confirm.



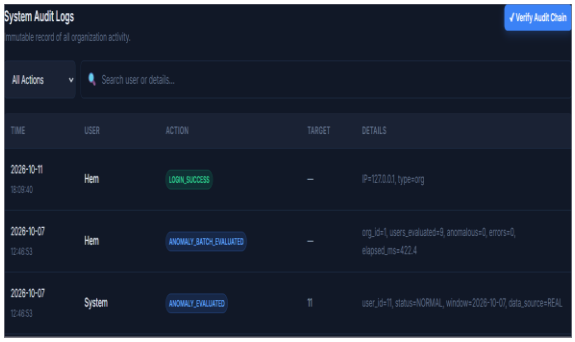
(a) DLP scan result (b) Recipient selection

(c) Expiry and download limit (d) Final review before upload

Fig. 3. Upload wizard: (a) DLP scan result, (b) recipient selection, (c) expiry and download limit, (d) final review.

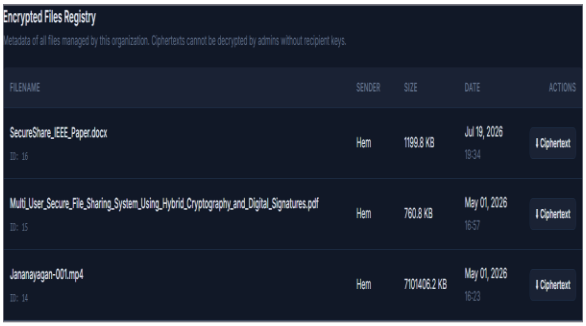
C. Administration

The admin panel has six tabs: join requests, members, files, audit logs, DLP policies and anomaly detection. The DLP tab shows the default policy, and the audit tab has an action filter, search and a Verify Audit Chain button. One batch anomaly evaluation covered nine users, flagged none, and took 422.4 ms. These screenshots show normal workflows only.



(a) Audit log tab

Fig. 4. Audit tab with action filter, search and Verify Audit Chain.



(b) Encrypted files registry

Fig. 5. Encrypted files registry: metadata visible, ciphertext-only download.

IX. COMPARISON WITH CONVENTIONAL SERVICES

SecureShare is worth setting next to consumer storage such as Google Drive or Dropbox — an architecture comparison, not a benchmark, since several vendors already offer audit export, DLP and customer-managed keys as paid add-ons.

On a typical consumer service the provider holds the keys, and revocation and audit are often paid extras. SecureShare is built the other way: only recipients hold keys, every download carries a mandatory signature check, and the audit trail is a verifiable hash chain — though every transfer costs server-side cryptography, a lost password means lost files, and the system has been tested far less than what it is compared against.

X. LIMITATIONS AND FUTURE WORK

1) Trust in the server. The server still sees the password and a temporary plaintext copy during uploads. Browser-side unwrapping with the Web Crypto API would remove this, at the cost of rewriting the streaming pipeline.
2) Unauthenticated CBC. Integrity rests on the signature alone. AES-GCM would authenticate each chunk.
3) OTP state. The pending registration and hashed OTP live in a client-side cookie by default, so a six-digit code could be brute-forced. A server-side store would fix this.
4) Audit tail. Deleting the newest rows goes unnoticed unless the latest hash is stored where the database operator cannot write.
5) DLP accuracy. Any 12-digit number resembles an Aadhaar number, so false positives are common. Binary files and Office documents are not inspected.
6) Deployment. Rate-limit counters live in memory and reset on restart, so multiple nodes need a shared store.
7) Recovery and quantum risk. A lost password means a lost key. Shamir secret sharing and a post-quantum key-encapsulation mechanism could help.
8) Measurement. We have not measured throughput, latency or memory use. Constant memory use follows from the design but has not been profiled.

XI. CONCLUSION

SecureShare shows that a small organization can share files securely, within a server-honest threat model, while keeping administrative control. Per-recipient key wrapping and mandatory signature checks turn access control and integrity into cryptographic properties. The browser scanner shows that DLP does not have to cost confidentiality, as long as it stays a safety net rather than strict enforcement. The design fits departments, labs, student bodies and small firms with a trusted administrator but an untrusted storage back end.

ACKNOWLEDGMENT

The authors thank the Department of Information Technology, K. Ramakrishnan College of Technology, Trichy, and the colleagues who took part as members of the test organization.

REFERENCES

[1] S. Kamara and K. Lauter, “Cryptographic cloud storage,” in Financial Cryptography and Data Security (Lecture Notes in Computer Science, vol. 6054). Berlin, Germany: Springer, 2010, pp. 136–149.

[2] M. Backendal, M. Haller, and K. G. Paterson, “MEGA: Malleable encryption goes awry,” in Proc. IEEE Symp. Security and Privacy (SP), San Francisco, CA, USA, May 2023.

[3] S. Alneyadi, E. Sithirasenan, and V. Muthukkumarasamy, “A survey on data leakage prevention systems,” J. Netw. Comput. Appl., vol. 62, pp. 137–152, Feb. 2016.

[4] R. L. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Commun. ACM, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[5] M. Bellare and P. Rogaway, “Optimal asymmetric encryption,” in Advances in Cryptology—EUROCRYPT ’94 (Lecture Notes in Computer Science, vol. 950). Berlin, Germany: Springer, 1995, pp. 92–111.

[6] M. Bellare and P. Rogaway, “The exact security of digital signatures: How to sign with RSA and Rabin,” in Advances in Cryptology—EUROCRYPT ’96 (Lecture Notes in Computer

Science, vol. 1070). Berlin, Germany: Springer, 1996, pp. 399–416.

[7] Advanced Encryption Standard (AES), National Institute of Standards and Technology, FIPS PUB 197, Nov. 2001.

[8] M. Dworkin, Recommendation for Block Cipher Modes of Operation: Methods and Techniques, NIST Special Publication 800-38A, Dec. 2001.

[9] S. Vaudenay, "Security flaws induced by CBC padding—Applications to SSL, IPSEC, WTLS …," in Advances in Cryptology—EUROCRYPT 2002 (Lecture Notes in Computer Science, vol. 2332). Berlin, Germany: Springer, 2002, pp. 534–545.

[10] Secure Hash Standard (SHS), National Institute of Standards and Technology, FIPS PUB 180-4, Aug. 2015.

[11] K. Moriarty, B. Kaliski, and A. Rusch, "PKCS #5: Password-based cryptography specification version 2.1," Internet Engineering Task Force, RFC 8018, Jan. 2017.

[12] N. Provos and D. Mazières, "A future-adaptable password scheme," in Proc. USENIX Annu. Tech. Conf., FREENIX Track, Monterey, CA, USA, Jun. 1999, pp. 81–91.