International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

Secure Encrypted Database System

A Multi-Layer Cryptographic Approach to Local Data Security

Vedant Kinarkar, Jayendra Kondekar, Yog Chikram
Kshitij Niwate, Aryan Tadke, Mr. Mohammad Sajid
Department of Computer Science and Engineering

Abstract - This paper presents the design, implementation, and security analysis of a command-line encrypted database system developed in Python. The system addresses the need for lightweight, locally-deployed secure data storage without reliance on external infrastructure. It employs a three-layer cryptographic architecture: file-level encryption using Fernet with PBKDF2-HMAC-SHA256 key derivation, record-level encryption with unique per-record keys derived from a random salt and sixteen master keys, and credential protection using salted PBKDF2 hashing at 600,000 iterations. The system is modularised across eleven Python files following separation of concerns, achieves O(1) record access via a binary index, and parallelises encryption and decryption using Python's ThreadPoolExecutor. Security analysis demonstrates resistance to rainbow table attacks, ciphertext correlation attacks, and offline brute-force attempts. Performance benchmarks show sub-second response times for individual record operations and linear scaling for bulk insertion. The system represents a practical model for applying layered cryptographic principles to local data management.

Keywords: cryptography, Fernet, PBKDF2, encrypted database, Python, key derivation, record-level encryption, secure storage, modular design

1. INTRODUCTION

The proliferation of sensitive personal and organisational data stored on local devices has created a pressing need for secure, accessible, and maintainable data storage solutions. Existing database systems — including SQLite, PostgreSQL, and MongoDB — are not encrypted by default, leaving data at risk from physical access, theft, or unauthorised process access. While full-disk encryption solutions such as BitLocker and VeraCrypt address the storage medium, they offer no protection once the disk is unlocked and the process is running.

This project presents a purpose-built encrypted database system that maintains data confidentiality at the record level throughout its lifecycle — at rest, during read operations, and during write operations — without dependence on operating system encryption features. The system is implemented entirely in Python using the widely audited cryptography library and requires no database server, network connection, or external key management infrastructure.

The primary contributions of this work are: (1) a three-layer encryption architecture that provides defence-in-depth; (2) per-record unique key derivation that prevents ciphertext correlation across records; (3) a binary framed database format with an external index enabling O(1) record access; (4) a clean modular implementation across eleven files with well-defined interfaces; and (5) a salted PBKDF2-based credential system resistant to offline dictionary attacks.

1.1 Motivation

Existing lightweight encrypted storage solutions either rely on a single encryption key shared across all records (e.g., SQLCipher), employ proprietary binary formats with limited auditability, or require network-accessible key management services. The present system was motivated by the need for a transparent, auditable, self-contained solution suitable for sensitive personal records, research data, or small-team deployments where a database server is impractical.

1.2 Scope and Limitations

The system is designed for single-user, local deployment on a trusted machine. It does not provide multi-user access control, network-accessible APIs, or replication. Record values are restricted to Python dict literals, and the current implementation does not support structured querying beyond sequential scan. These constraints are acknowledged as deliberate trade-offs in favour of simplicity and auditability.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

2. RELATED WORK

Encrypted database research spans two broad categories: server-side encrypted databases and client-side encrypted storage. CryptDB [1] pioneered the use of onion-layered encryption on a PostgreSQL backend, allowing SQL queries over encrypted data at the cost of query expressiveness and a complex key management hierarchy. Cipherbase [2] extended this to hardware-assisted query processing. These approaches require significant infrastructure and are ill-suited to local deployment.

SQLCipher [3] provides transparent AES-256 encryption for SQLite databases using a single database-level key. While widely deployed in mobile applications, the shared-key model means that compromise of the key exposes all records simultaneously, and there is no per-record isolation. The present system addresses this limitation explicitly.

Key derivation standards have been extensively studied. PBKDF2 [4], bcrypt [5], and Argon2 [6] are the principal password-based key derivation functions in contemporary use. PBKDF2 with HMAC-SHA256 and a minimum of 600,000 iterations is the current OWASP recommendation [7] and is adopted throughout this system for all password-to-key derivations. Fernet [8], the symmetric encryption scheme used for record payloads, combines AES-128-CBC with HMAC-SHA256 for authenticated encryption, providing both confidentiality and integrity.

The concept of per-record key diversification appears in hardware security module (HSM) design, where a key derivation function is applied to a master key and a per-object identifier to produce object-specific keys [9]. This system applies the same principle in software, using a random 16-byte salt as the diversifier.

3. SYSTEM ARCHITECTURE

The system is structured as eleven Python modules with a strict dependency hierarchy. Figure 1 illustrates the layered architecture. All configuration constants are centralised in constants.py, ensuring a single point of change for cryptographic parameters and file paths. Encryption and decryption operations are separated into dedicated modules to enforce unidirectional data flow.

3.1 Module Dependency Hierarchy

Module	Depends On	Depended On By
constants.py	*stdlib only*	all modules
encryption.py	*constants, cryptography*	decryption, auth, backup, db_index, adddata, viewdata, deletedata, main
decryption.py	*constants, encryption*	auth, db_index, viewdata, deletedata
db_index.py	*constants, encryption, decryption*	adddata, viewdata, deletedata
auth.py	*constants, encryption, decryption, backup*	main
backup.py	*constants, encryption*	auth, main
adddata.py	*constants, encryption, db_index*	viewdata, main
viewdata.py	*encryption, decryption, db_index, adddata*	main
deletedata.py	*decryption, encryption, db_index*	main
helpandcreds.py	*none*	main
main.py	*all modules*	— (entry point)

Table 1: Module dependency matrix

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

3.2 Three-Layer Encryption Architecture

The system implements three distinct and independent encryption layers, each protecting a different scope of data:

- Layer 1 — Credential Layer: User credentials and the sixteen master Fernet keys are encrypted with PBKDF2-derived keys and stored in auth.enc and key.enc respectively. Each file prepends its own random 32-byte PBKDF2 salt.
- Layer 2 — File Layer: The database file (database.db) and its index (database.idx) are encrypted as complete binary blobs using a file key derived from the XOR combination of all sixteen master keys and a persistent file salt. Plaintext working copies exist only during an authenticated session and are wiped on exit.
- Layer 3 — Record Layer: Each individual record is encrypted with a unique key derived from SHA256(XOR(master_keys) ‖ random_16_byte_salt). The salt is stored prepended to the ciphertext. The per-record key is ephemeral — derived on demand and discarded after use.

This layered design ensures that an attacker who obtains the database file without the master keys sees only ciphertext at the file level. An attacker who obtains the master keys but not the individual record salts cannot decrypt any specific record without reconstructing the derivation. An attacker who compromises one record's decrypted plaintext learns nothing about any other record.

3.3 Database File Format

Records are stored in a binary framed format within database.db. Each frame consists of an 8-byte header (4-byte big-endian record ID followed by 4-byte big-endian payload length) and the variable-length encrypted payload. The payload is [16-byte salt][Fernet ciphertext].

```
database.db binary layout:

┌──────────────────────────────────────────────────────────────┐
│ Frame 0: [4B id][4B length][16B salt][ciphertext]            │
│ Frame 1: [4B id][4B length][16B salt][ciphertext]            │
│ ...                                                          │
└──────────────────────────────────────────────────────────────┘

database.idx (JSON):
{ "0": {"offset": 0, "length": 284, "deleted": false}, ... }
```

The companion index file (database.idx) stores a JSON map from record ID to byte offset, payload length, and deletion flag. This enables O(1) seek-based access to any record without scanning the database file. Deletion is a soft operation — only the index flag is updated, with zero writes to the database file itself.

4. CRYPTOGRAPHIC DESIGN

4.1 Key Derivation

All password-to-key derivations use PBKDF2-HMAC-SHA256 with 600,000 iterations and a 256-bit (32-byte) random salt, conforming to OWASP 2023 guidance [7]. The output is 32 bytes, formatted as a URL-safe base64 string for compatibility with the Fernet specification. The derivation function is defined as:

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

```
key = PBKDF2-HMAC-SHA256(
    password  = user_password.encode('utf-8'),
    salt      = random_bytes(32),
    iterations= 600_000,
    dkLen     = 32
)
fernet_key = base64.urlsafe_b64encode(key)
```

Four independent PBKDF2 derivations are performed in the system: one for the auth master password (protecting auth.enc), one for the credential password hash (stored inside auth.enc), one for the key master password (protecting key.enc), and one for the backup password (protecting key_backup.enc). Each derivation uses an independent random salt, ensuring that the same password used in two roles produces different keys.

4.2 Per-Record Key Diversification

The sixteen master Fernet keys loaded from key.enc serve as the cryptographic root for all record operations. They are combined into a single 32-byte value by XOR, then used as the 'password' input to a per-record PBKDF2 derivation keyed by a freshly generated 16-byte salt:

```
combined = XOR(raw_key_0, raw_key_1, ..., raw_key_15)   # 32 bytes


record_key = PBKDF2-HMAC-SHA256(
    password  = combined,
    salt      = random_bytes(16),   # unique per record per write
    iterations= 600_000,
    dkLen     = 32
)


ciphertext = Fernet(base64(record_key)).encrypt(json(record))
stored     = salt_16 + ciphertext
```

Because the salt is unique per write operation, modifying a record and writing it back produces a completely different ciphertext, even if the plaintext is unchanged. This provides forward ciphertext unlinkability: an observer who records database.db snapshots over time cannot determine whether a record was modified between snapshots.

4.3 File-Level Key Derivation

The file key used to encrypt the entire database.db and database.idx files is derived using the same XOR-combined master key material as the record key, but with a different salt (stored permanently in database.salt) and the domain separator b'file'. This ensures that the file key and any record key can never collide even if the random salts were to repeat.

4.4 Fernet Authenticated Encryption

Fernet [8] is used for all symmetric encryption operations. Each Fernet token encodes: a version byte, a timestamp (8 bytes), an IV (16 bytes), the AES-128-CBC ciphertext, and an HMAC-SHA256 authentication tag (32 bytes). The authentication tag covers all preceding fields, ensuring that any tampering with the stored ciphertext is detected before decryption. This provides authenticated encryption with associated data (AEAD) semantics at the record level.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

4.5 Fernet Instance Cache

Constructing a Fernet object involves key scheduling operations that are non-trivial in cost. Since the sixteen master keys are fixed for the duration of a session, the system constructs all sixteen Fernet instances once at login and stores them in a module-level list. The raw key bytes are simultaneously stored for XOR combination. This eliminates repeated key scheduling overhead on every record operation, yielding significant performance improvements for bulk operations.

5. IMPLEMENTATION

5.1 Technology Stack

Component	Choice / Version
Language	Python 3.11+
Symmetric encryption	Fernet (cryptography library ≥ 41.0)
Key derivation	PBKDF2-HMAC-SHA256 (hashlib, stdlib)
Serialisation	JSON (stdlib) — replaces earlier pickle usage for safety
Parallelism	concurrent.futures.ThreadPoolExecutor (stdlib)
Binary framing	struct.pack / struct.unpack, big-endian uint32 (stdlib)
Random generation	secrets.token_bytes (stdlib, cryptographically secure)
Index format	JSON on disk, dict in memory
Test environment	CPython 3.11, Ubuntu 24.04

Table 2: Technology stack

5.2 Session Lifecycle

The session lifecycle follows a strict sequence:

1. User invokes main.py. auth.py checks for auth.enc and key.enc.
2. If absent: setup() creates credentials, generates 16 Fernet keys, writes to disk. Program exits; user relaunches.
3. On relaunch: login() prompts Master Password → validates auth.enc → prompts Username + Password → validates → prompts Master Key → decrypts key.enc.
4. build_fernet_cache() constructs all 16 Fernet instances. decrypt_db_files() decrypts database.db.enc and database.idx.enc into working plaintext files.
5. User interacts via menu. After every write: sync_to_disk() re-encrypts .enc files without deleting plaintext (crash protection).
6. On exit (option 9): _secure_exit() calls encrypt_db_files() which re-encrypts and deletes plaintext files.

5.3 Parallel Encryption and Decryption

For bulk add operations and view/modify/delete operations that require decrypting multiple records, the system uses Python's ThreadPoolExecutor to parallelise encryption and decryption. Since the Global Interpreter Lock (GIL) is released during cryptographic operations in the cryptography library's C extension, genuine parallelism is achieved for

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

I/O-bound and compute-bound crypto workloads. A thread pool with the default worker count (min(32, cpu_count + 4)) is used.

5.4 Soft Delete and Compaction

Record deletion is implemented as a soft delete: the index entry's deleted flag is set to True and the index is saved. The database file is not written during deletion, making it an O(1) operation. The compact_database() function, accessible from the menu, performs a full rewrite of database.db keeping only non-deleted records, updating offsets in the index accordingly. This design separates the common case (fast deletion) from the maintenance case (space reclamation).

5.5 Modify Without ID Change

Record modification appends the new encrypted payload to the end of database.db and updates the index entry for that record ID in-place (new offset and length). The original bytes at the old offset become unreachable orphans, cleaned up by the next compact operation. This approach avoids rewriting the entire database file on every modify, while preserving record ID stability — the record's logical identity does not change across modifications.

6. SECURITY ANALYSIS

6.1 Threat Model

The system defends against the following threat classes:

- Offline file access: An attacker obtains copies of the database files from disk without knowing any passwords or keys.
- Partial credential compromise: An attacker obtains one of the four passwords (master auth, user password, master key, backup password) but not the others.
- Ciphertext analysis: An attacker accumulates multiple snapshots of the database file over time and attempts to infer changes.
- Brute-force: An attacker attempts offline dictionary or brute-force attacks against the credential files.

The system does not defend against: a fully compromised runtime environment (malware with memory access), side-channel attacks on the host CPU, or physical coercion. These are outside the stated scope.

6.2 Rainbow Table Resistance

Every password-derived key in the system is produced by PBKDF2 with a unique 32-byte random salt. Precomputed rainbow tables are computationally infeasible against salted PBKDF2 outputs: a table covering even a 64-bit salt space at one entry per nanosecond would require approximately 585 years to build. The 256-bit salt used here renders this attack completely infeasible.

6.3 Brute-Force Resistance

At 600,000 PBKDF2 iterations, a single password verification requires approximately 0.3–0.5 seconds on a modern CPU. An attacker capable of 10 billion SHA-256 operations per second (GPU-accelerated) would be reduced to approximately 33,000 PBKDF2 evaluations per second. Against a 12-character random alphanumeric password (entropy ≈ 71 bits), exhaustive search would require approximately 10^7 years.

6.4 Record Isolation

Because each record is encrypted with a unique key derived from a random salt, compromise of one record's decryption key — whether through cryptanalysis or key extraction — reveals nothing about any other record's key. The XOR combination of the sixteen master keys means that compromise of any strict subset of master keys is insufficient to derive any record key; all sixteen are required.

6.5 Ciphertext Unlinkability

Re-encrypting a record (on modification) generates a new 16-byte salt, producing an entirely different ciphertext. An observer comparing two snapshots of database.db cannot determine which records were modified or whether the

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

plaintext changed. The file-level Fernet encryption also re-randomises the IV on every sync_to_disk() call, further obscuring file-level patterns.

6.6 Known Limitations

The XOR combination of master keys introduces a known weakness: if all sixteen keys are identical (which cannot occur given Fernet.generate_key() uses os.urandom), the XOR result would be zero. In practice, the probability of even two identical random 32-byte keys is negligible (approximately 2^-256). The use of XOR is a design choice for simplicity; a more robust approach would use HKDF [10] with all sixteen keys as input key material, which is noted as a direction for future work.

Property	This System	SQLCipher	Plain SQLite
At-rest file encryption	Yes	Yes	No
Per-record key isolation	Yes	No	No
Salted password hashing	Yes	Yes	N/A
PBKDF2 ≥ 600k iterations	Yes	Config	N/A
Ciphertext unlinkability	Yes	No	No
Authenticated encryption (AEAD)	Yes	Yes	No
No external infrastructure	Yes	Yes	Yes
Structured query support	No	Yes	Yes

Table 3: Security property comparison

7. PERFORMANCE EVALUATION

7.1 Benchmark Methodology

Benchmarks were conducted on a mid-range development machine (Intel Core i5-1235U, 16 GB RAM, SSD storage, Python 3.11.6, cryptography 41.0.7). Each measurement represents the mean of ten trials after a two-trial warm-up to account for OS file cache effects. PBKDF2 timing is inherently dominated by the iteration count and was measured separately from record I/O.

7.2 Results

Operation	Mean Time	Notes
Login (PBKDF2 × 2)	**~0.8 s**	*Auth + key derivation*
Add 1 record (encrypt + write)	**~0.35 s**	*Includes PBKDF2 for record key*
Add 100 records (parallel)	**~4.2 s**	*ThreadPoolExecutor, 100 PBKDF2 calls*
View 100 records (parallel decrypt)	**~4.1 s**	*ThreadPoolExecutor*

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

View 1 record (seek + decrypt)	**~0.34 s**	*O(1) index lookup*
Soft delete (index only)	**~0.05 s**	*No DB file write*
Compact (100 records)	**~4.5 s**	*Full rewrite + re-encrypt*
sync_to_disk() after write	**~0.12 s**	*Re-encrypt DB + index .enc files*

Table 4: Performance benchmark results

7.3 Discussion

The dominant performance cost is PBKDF2 at 600,000 iterations, which accounts for approximately 0.33 seconds per derivation call. Each record encryption/decryption requires one PBKDF2 call for the record key derivation. For workloads requiring high-throughput bulk operations, the iteration count could be reduced at the cost of brute-force resistance, or the record key derivation could be replaced with a faster KDF such as HKDF, which does not iterate.

The parallelism gain from ThreadPoolExecutor is limited by the number of available CPU cores. On the test machine (10 cores), 100-record operations exhibit approximately a 10× speedup versus sequential processing. View and add operations for databases of up to 1,000 records complete within 45 seconds, which is acceptable for the intended use case.

8. DESIGN DECISIONS AND TRADE-OFFS

8.1 JSON over Pickle for Serialisation

Early versions of the system used Python's pickle module to serialise the master key dictionary. Pickle was replaced with JSON for two reasons: (1) pickle is susceptible to arbitrary code execution if a maliciously crafted payload is deserialised [11], creating an attack vector if key.enc were replaced by an adversary; (2) JSON is human-readable and language-agnostic, facilitating future interoperability.

8.2 Sixteen Master Keys

The choice of sixteen master keys rather than one is a design legacy retained for its contribution to key diversity in the XOR combination step. A single 256-bit master key would be cryptographically equivalent if HKDF were used for record key derivation. The current design was preserved as it was already implemented and tested; migration to HKDF is noted as future work.

8.3 Soft Delete with External Index

The decision to use a soft delete model was driven by the observation that rewriting the entire database file on every deletion is O(n) in database size and involves re-encrypting all records — a prohibitive cost for large databases. The external index adds a small amount of metadata overhead but enables O(1) deletions and O(1) random access, representing a favourable trade-off.

8.4 File-Level Encryption as Outer Layer

The file-level Fernet encryption of database.db was added to prevent metadata leakage: without it, the binary frame headers (record IDs and payload lengths) would be visible in plaintext, revealing the number of records and the approximate size of each. The outer encryption eliminates this information channel. The cost is an additional encrypt/decrypt pass on the entire file at session boundaries.

9. FUTURE WORK

Several directions for future development have been identified:

- HKDF-based record key derivation: Replace the PBKDF2 per-record derivation with HKDF-Expand, reducing per-record key derivation from ~0.33s to microseconds while maintaining cryptographic soundness.

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 15 Issue 04 , April - 2026

- Argon2 for credential hashing: Migrate from PBKDF2 to Argon2id, which provides memory-hard hashing and resistance to GPU-accelerated brute force attacks.
- Structured querying: Implement an in-memory query engine that decrypts records on demand and filters by field value, enabling basic search without exposing plaintext to disk.
- Multi-user access control: Introduce per-user key envelopes, allowing multiple users with different credentials to access the same database with role-based record visibility.
- GUI front-end: Develop a Tkinter or web-based interface to improve accessibility for non-technical users.
- Integrity tree: Implement a Merkle tree over record hashes to detect unauthorised tampering with individual records even within an authenticated session.

10. CONCLUSION

This paper has presented the design, implementation, and security analysis of a multi-layer encrypted database system implemented in Python. The system achieves defence-in-depth through three independent encryption layers: credential protection with salted PBKDF2, file-level Fernet encryption of the database at rest, and per-record unique key derivation that provides record isolation and ciphertext unlinkability.

The modular eleven-file implementation demonstrates that strong cryptographic properties can be achieved in a lightweight, dependency-minimal Python application without reliance on database servers or network-accessible key management. The binary index design enables O(1) record access and O(1) soft deletion, and the ThreadPoolExecutor-based parallelism provides practical performance for databases of hundreds to thousands of records.

Security analysis confirms resistance to offline brute-force attacks, rainbow table attacks, and ciphertext correlation across records and over time. The system represents a practical and auditable reference implementation of layered cryptographic principles applied to local data management, suitable for sensitive personal records, research data custody, and small-team deployments.

REFERENCES

[1] Popa, R. A., Redfield, C. M., Zeldovich, N., & Balakrishnan, H. (2011). CryptDB: Protecting Confidentiality with Encrypted Query Processing. Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11), pp. 85–100.

[2] Arasu, A., et al. (2013). Orthogonal Security with Cipherbase. CIDR 2013.

[3] Zetetic LLC. (2023). SQLCipher: Open Source Full Database Encryption for SQLite. https://www.zetetic.net/sqlcipher/

[4] Kaliski, B. (2000). PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898. Internet Engineering Task Force.

[5] Provos, N., & Mazières, D. (1999). A Future-Adaptable Password Scheme. Proceedings of the USENIX Annual Technical Conference, FREENIX Track, pp. 81–91.

[6] Biryukov, A., Dinu, D., & Khovratovich, D. (2015). Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications. IEEE European Symposium on Security and Privacy.

[7] OWASP Foundation. (2023). Password Storage Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

[8] Percival, C., & Josefsson, S. (2016). Fernet Specification. https://github.com/fernet/spec/blob/master/Spec.md

[9] National Institute of Standards and Technology. (2020). Recommendation for Key Derivation Using Pseudorandom Functions. NIST Special Publication 800-108 Rev. 1.

[10] Krawczyk, H., & Eronen, P. (2010). HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869. Internet Engineering Task Force.

[11] Python Software Foundation. (2023). pickle — Python Object Serialization: Warning on Untrusted Data. https://docs.python.org/3/library/pickle.html