

PDManager: A Privacy-First, Cross-Platform Encrypted Vault for Credentials and Documents with Layered Cryptography and User-Controlled Backups

Kalyani Bachhav

Computer Science, Indira College Malegoan,
India

Abstract: Password managers protect credentials but usually leave sensitive documents to separate, often cloud-based tools, which fragments personal data across applications. This paper presents PDManager (project name passwordmaster), a cross-platform vault built with Flutter and Dart that stores passwords, secure notes and documents and lets users edit, convert and organize files without leaving the encrypted environment. The design applies defense in depth: per-field AES-256-GCM encryption, whole-database encryption with SQLCipher, an Argon2id master-password verifier, hardware-backed key storage (Android Keystore / iOS Keychain), biometric quick unlock, a rate-limited lockout and root/jailbreak detection that triggers a secure wipe of all application data. Backups are encrypted on the device under an independent passphrase before being written to a local folder or Google Drive. We describe the architecture, the cryptographic workflows and the password generator and strength scoring, and analyse the security properties against common threats. Testing so far has been manual, so we report design-level expectations rather than benchmarks and state the limitations and future work openly.

Keywords—password manager, AES-256-GCM, Argon2id, SQLCipher, Flutter, secure document management, encrypted backup

I. INTRODUCTION

Individuals now manage dozens of online accounts alongside a growing collection of sensitive digital documents such as identity cards, bank statements and contracts. Two habits amplify the resulting risk: reusing passwords across services, which turns one breach into a chain of account takeovers (credential stuffing), and choosing weak passwords that automated guessing defeats in fractions of a second. Password managers address both problems by generating and remembering unique credentials, and current guidance supports their use [10].

Security tooling is nevertheless fragmented. A user typically keeps credentials in one application but relies on separate, frequently unencrypted or cloud-dependent tools to edit, convert or split sensitive documents. Every hand-off uploads private material to another party and widens the attack surface. At the same time, cloud-centric managers concentrate many users' vaults on provider servers; the 2022 LastPass incident showed how a server-side compromise exposes stored vault copies to offline cracking attempts [11].

This paper presents PDManager, a privacy-first vault that unifies credential management and secure document processing behind one master password. It is written in Dart with the Flutter framework [8] and targets Android, iOS, Windows, macOS, Linux and the web. The main contributions are:

- A unified encrypted vault for passwords, secure notes and documents, with in-vault tools (image editor, PDF-to-DOCX converter, PDF organizer/splitter) so that sensitive files need not be exposed to external apps.
- A layered cryptographic design combining per-field AES-256-GCM, SQLCipher database encryption, an Argon2id master-password verifier, hardware-backed key storage and a secure wipe of all application data when a rooted or jailbroken device is detected.
- User-controlled backups encrypted on the device, under an independent passphrase, before they reach a local folder or Google Drive.
- A transparent account of what has and has not been verified, together with limitations and future work.

II. BACKGROUND AND PROBLEM ANALYSIS

A. Weaknesses of Existing Password Managers

Four recurring weaknesses motivate this work. Centralized cloud vulnerability: purely cloud-based managers hold many vaults in one place, so a single breach exposes all of them to offline cracking. Subscription lock-in: synchronizing

across devices often requires a recurring fee. Limited user control: users must trust the provider's infrastructure and zero-knowledge claims without being able to verify how data is handled. Weak offline support: some managers restrict functionality without an internet connection.

B. Security Problems Addressed

PDManager targets three problems: (i) credential stuffing and password reuse, countered by making unique, strong passwords easy to generate and store; (ii) local device compromise, where someone with physical access must still be unable to read stored secrets, countered by biometric locking and strong database encryption and, on rooted or jailbroken devices, by a secure wipe of all application data; and (iii) interception of data or theft of the storage medium, countered by encrypting data before it is stored or transmitted.

C. Why Plaintext Storage and Weak Passwords Fail

Plaintext storage gives an attacker every credential the moment a malicious app, a rooted or stolen device or a compromised operating system exposes the storage; it relies entirely on OS sandboxing and provides no defense in depth. Reused passwords create a domino effect: a breach of a poorly secured forum yields credentials that open a bank account. Weak passwords such as “password123” fall to brute-force and dictionary attacks, and passwords built from pet names or birth dates are open to social engineering through public profiles. These failure modes shape the requirements below.

III. SYSTEM OVERVIEW

A. Purpose and Intended Users

PDManager is an encrypted digital vault for credentials and documents. Its intended users are privacy-conscious individuals, professionals and everyday users who handle sensitive documents and want to manage passwords, notes and files without exposing them to external cloud-processing services. What sets it apart from typical managers, which mostly attach static files to vault items, is a document-processing suite: an image editor, a PDF-to-DOCX converter and a PDF organizer/splitter that run inside the protected environment.

B. Stored Data and Features

Each password entry holds a title, username, password, website URL, notes and a category; documents and images are handled by separate document and image/PDF models. Further features are a customizable password generator, a strength indicator, biometric unlock, image compression and encrypted backup and restore. Backups use the .pwm format for passwords and the .sdm format for documents.

C. Technology Stack

TABLE I. TECHNOLOGY STACK

Layer	Technology
Language / UI	Dart, Flutter
Local database	SQLite with SQLCipher (sqflite_sqcipher)
Cryptography	cryptography package: AES-256-GCM, Argon2id, PBKDF2
Key storage	flutter_secure_storage (Android Keystore / iOS Keychain)
Authentication	Master password, biometrics (local_auth), Clerk, Google Sign-In
Cloud services	Firebase (Core, Firestore, Functions); Google Drive (googleapis)
Documents	pdf, printing, image, flutter_image_compress; isolated Python script for PDF-to-DOCX
Platforms	Android, iOS, Windows, macOS, Linux, Web

D. Requirements and Traceability

Table II maps the requirements derived from Section II to the mechanisms and sections that address them.

TABLE II. REQUIREMENTS TRACEABILITY

Requirement	Mechanism	Section
Confidentiality at rest	Field-level AES-256-GCM; SQLCipher	IV-B, IV-D

Key protection	Hardware keystore; no keys in database	IV-C
Response to a compromised host	Root/jailbreak detection; secure wipe of keys and all app data	IV-G, V-D
Integrity	128-bit GCM authentication tags	IV-B
Requirement	Mechanism	Section
Resistance to guessing	Argon2id verifier; 5-attempt lockout	V-B
Recoverability	Encrypted local / Drive backups	V-C
Private document handling	In-vault local tools; no AI or uploads	III-A, III-E
Credential hygiene	Generator and strength meter	VI
Platform reach	Flutter, six targets	III-C

E. Design Decision: No Artificial Intelligence

The application uses no AI model, LLM or machine-learning API. All processing, including document conversion and image compression, runs locally through standard Dart libraries and bundled tools using deterministic algorithms. No vault data, personally identifiable information or password is therefore ever sent to an AI service. Communication with the authentication and cloud services that are used is protected by TLS, and vault content is encrypted before it leaves the device.

IV. ARCHITECTURE AND SECURITY DESIGN

A. Layered Architecture

Fig. 1 shows the architecture. The Flutter presentation layer exposes the vault screens, the password generator and the document tools. Beneath it, security services handle authentication, cryptography, key custody, rate limiting and device-integrity checking: CryptoService implements AES-256-GCM and the key-derivation functions, KeystoreService wraps the platform keystore and RateLimitingService tracks failed logins, while the root/jailbreak check (Section IV-G) runs before the vault can be opened. Persistent data lives in an encrypted SQLite database [5], while keys are held separately in the hardware-backed keystore [6], [7]. Google Drive, Firebase [9] and the local PDF-to-DOCX converter are optional components; the two cloud services never receive plaintext vault content.

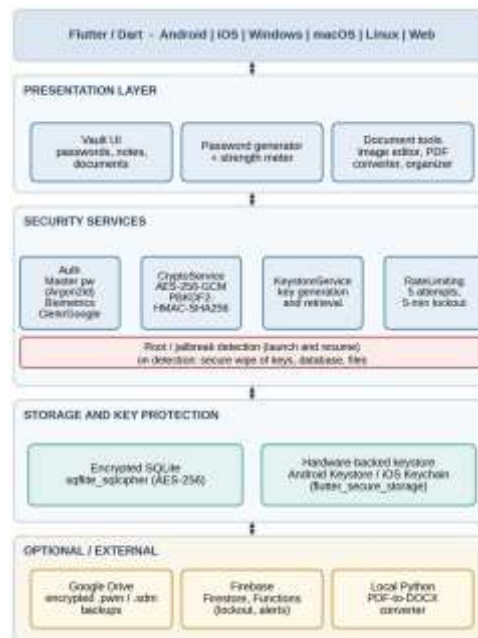


Fig. 1 Layered architecture of PDManager. Keys are kept in the hardware-backed keystore, separately from the encrypted database. The root/jailbreak check sits in the security-services layer.

B. Cryptographic Primitives

Table III summarizes the primitives. AES-256-GCM is an authenticated mode [1]: alongside the ciphertext it produces a 128-bit tag that is verified on decryption, and altered data raises an AuthenticationTagException instead of being decrypted. A fresh 96-bit IV is generated for every field encryption and is never reused. Argon2id is a memory-hard

function [2] configured with the OWASP-recommended cost of 19 MiB and two iterations [4]; PBKDF2-HMAC-SHA256 [3] derives keys for backup passphrases and serves as a legacy fallback.

TABLE III
CRYPTOGRAPHIC PRIMITIVES AND PARAMETERS

Purpose	Primitive	Parameters
Field and document encryption	AES-256-GCM	256-bit key; 96-bit IV per field; 128-bit tag
Database at rest	SQLCipher (AES-256)	Separate device_db_key
Master-password verifier	Argon2id	19 MiB, 2 iterations, 16-byte random salt
Backup key derivation	PBKDF2 (HMAC-SHA256)	100,000 iterations
Key generation	CSPRNG	Random.secure(); 256-bit keys

C. Key Management

Three secrets are held in the Android Keystore or iOS Keychain through flutter_secure_storage: the SQLCipher key (device_db_key), the field-encryption key (vault_field_key) and the biometric quick-unlock key. On supported devices these keys are non-exportable at the hardware level. No key is hardcoded or stored inside SQLite, and the master password is never stored, in plaintext or encrypted; only its Argon2id verifier is kept.

D. Defense in Depth

Encryption is applied at two levels. Field-level AES-256-GCM protects sensitive values such as passwords, usernames, notes and document files individually and detects tampering per value. SQLCipher additionally encrypts the entire database file, so table structure and metadata are also unreadable at rest. If one layer were bypassed, the other would still stand.

E. Access Control

Access requires the master password or, when enabled, a biometric prompt (fingerprint, Face ID or Windows Hello via local_auth). Clerk and Google Sign-In support account authentication. RateLimitingService limits guessing as described in Section V-B. The current version has no inactivity-based auto-lock; locking is driven only by the failed-attempt limiter

F. Rationale for Local-First Storage

The application supports both local and cloud components, but the vault itself is stored locally and the cloud is only an optional destination for encrypted backups. Four reasons drive this choice. *Privacy and control*: the user holds the only keys, so the data belongs to the user rather than to a third party. *Zero-knowledge behavior*: because data is encrypted on the device before it can touch the internet, a breach of the cloud provider yields only unreadable ciphertext. *Offline availability*: access does not depend on connectivity or server outages. *Reduced attack surface*: compromising individual devices at scale is much harder than attacking one centralized server.

G. Root and Jailbreak Detection with Secure Wipe

Every layer above assumes that the host operating system enforces app sandboxing and keystore access control. On a rooted (Android) or jailbroken (iOS) device that assumption no longer holds, so PDManager treats such a device as compromised. At launch, and each time the app returns to the foreground, a device-integrity check looks for common indicators: superuser binaries and management apps, a writable system partition, modified or test-key builds and jailbreak artefacts. A positive result is not a dismissible warning. The app performs a secure wipe of all application data before any vault screen is shown, so the vault cannot be opened and its keys cannot be read through the normal code path. The wipe covers:

- the three keystore secrets (device_db_key, vault_field_key and the biometric quick-unlock key) and the Argon2id verifier and salt;
- the SQLCipher database with its journal and write-ahead-log files, overwritten and then deleted;
- app-private storage: preferences, caches and the temporary or decrypted working copies produced by the document tools (image editor, converter, PDF organizer);
- stored sign-in state for Clerk, Google Sign-In and Google Drive, so the app can no longer reach the user's cloud account.

Keys are destroyed first. This is a crypto-erase: once the field and database keys are gone, any ciphertext that survives, for example because flash wear levelling prevents a reliable overwrite, cannot be decrypted. Encrypted backups kept

outside the app sandbox (a local folder or Google Drive) are not touched; they stay protected by their independent passphrase and are the only route to recovery (Section V-C).

V. CORE WORKFLOWS

A. Creating a Password Entry

When the user saves an entry (Fig. 2), the sensitive fields are first encrypted individually with AES-256-GCM under `vault_field_key`. The resulting ciphertext, with its IV and authentication tag, is inserted into SQLite, and the whole database file is additionally encrypted by SQLCipher under `device_db_key`.

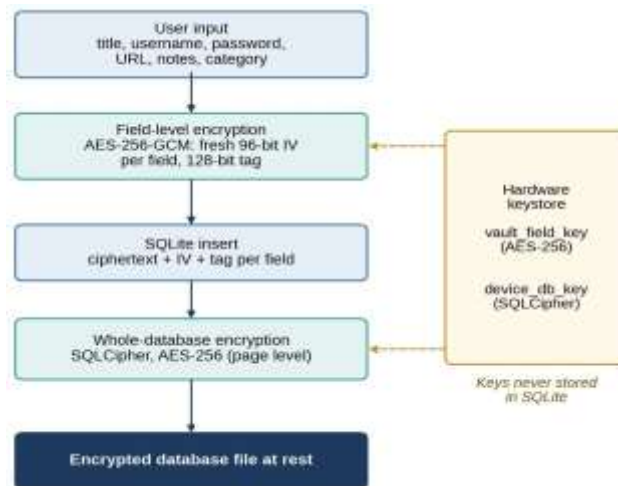


Fig. 2 Data flow when a password entry is created: field-level encryption precedes database-level encryption.

B. Unlocking the Vault

The master password is fed into Argon2id with a stored random 16-byte salt to derive a verifier, which is compared with the value held in the Keystore/Keychain (Fig. 3). Only on a match does the application retrieve the two vault keys: the database key unlocks SQLCipher, and the field key stays in memory to decrypt fields as they are viewed. After five failed attempts the app locks the user out for five minutes. The lock state is also synchronized to Firebase Cloud Functions for an authoritative, server-backed lockout, and a security-alert e-mail can be sent to the registered Gmail address.

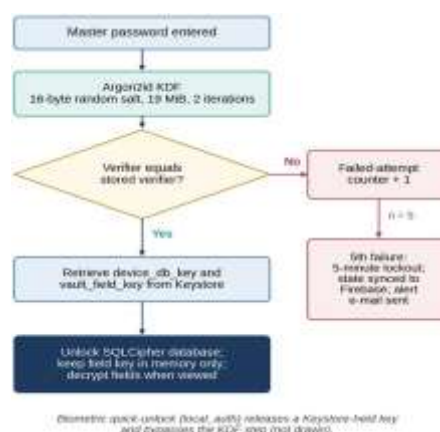


Fig. 3 Vault unlock and lockout workflow.

C. Backup and Restore

Backups are offline-first. The user can save to a local directory (for example `SecureVault_Backups`) or to a dedicated PDManager folder in their own Google Drive, either manually or through silent automatic backup. Before any file is written or uploaded, the payload is encrypted on the device with AES-256-GCM under a key derived from a passphrase (Fig. 4). This passphrase, the Auto Backup Passphrase or a custom export passphrase, is independent of the master password;

without it nobody can open the backup. To restore, the user opens the Import section, selects a .pwm or .sdm file locally or from Google Drive and enters the passphrase, after which the app verifies, decrypts and restores the data into the local vault.

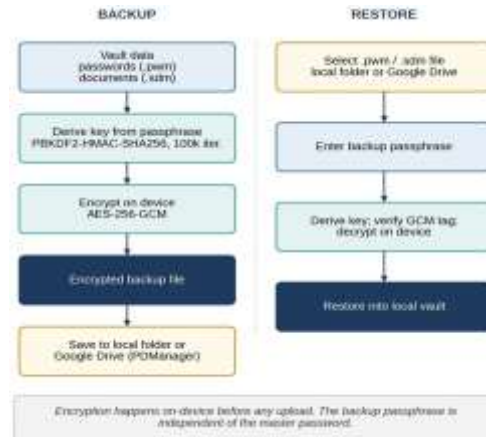


Fig. 4 Encrypted backup and restore. Plaintext is never uploaded.

Two failure scenarios matter. If the device is lost, its local vault cannot be recovered because the keys live in that device's hardware keystore; the user can install the app on a new device and restore from a backup with the backup passphrase, and without a backup the data is lost. If the master password is forgotten, no server-side reset exists, so recovery depends on holding a backup whose passphrase is still known.

D. Secure Wipe on Root Detection

When the integrity check of Section IV-G fails, the app (1) writes a persistent wipe-pending marker, so that a wipe interrupted by a crash or power loss resumes at the next launch before any vault screen is shown; (2) deletes the keystore secrets and the Argon2id verifier; (3) overwrites and deletes the database files and all working files; (4) clears preferences, caches and sign-in state; and (5) clears the marker and returns the app to its first-run state with a notice explaining why the data was erased. The sequence runs locally and needs no network connection. It is irreversible and has the same recovery path as a lost device (Section V-C): data can be restored only from a .pwm or .sdm backup, on a device that passes the check.

VI. PASSWORD GENERATION AND STRENGTH ANALYSIS

A dedicated PasswordGeneratorPanel lets the user set the password length with a slider and toggle uppercase letters, lowercase letters, numbers and symbols. Generation runs locally on a cryptographically secure random source and is an $O(N)$ operation in the password length.

Strength is scored between 0.0 and 1.0. The score gains 0.25 for each criterion met: length of at least eight characters, an uppercase letter, a digit and a symbol. The PasswordStrengthIndicator widget shows the result in red (Weak), yellow (Medium) or green (Strong), and a score below 0.3, meaning one criterion or none, is explicitly flagged as Weak (Fig. 5).

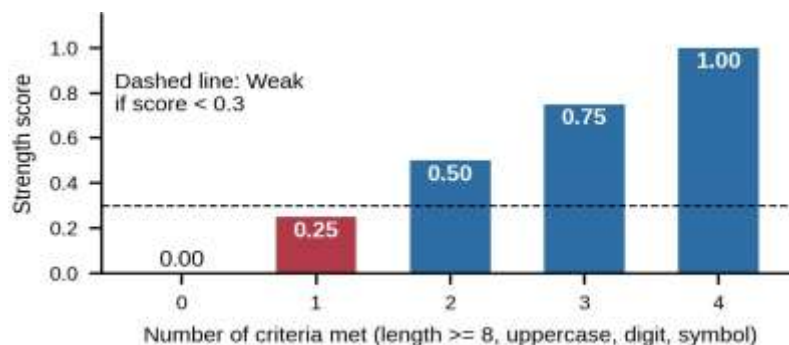


Fig. 5 Strength score as a function of the number of criteria met.

The scheme is a composition heuristic and has two known gaps. It cannot recognize predictable patterns such as Summer2024!, and it does not screen against common or breached passwords, which NIST guidance recommends [10]. The vault also does not yet detect reused passwords, and although each entry records createdAt and updatedAt timestamps, no feature flags old passwords for rotation.

VII. EVALUATION

A. Verification Status

Each feature was tested manually by the author against a written checklist of 46 test cases covering encryption, authentication and lockout, root detection and wipe, backup and restore, the generator and strength meter, the document tools and the six platform builds. Table IV summarizes the outcome: all 46 cases passed. These are manual functional checks, not an automated suite: the standard test/ directory does not exist, the results are reported by the author and have not been independently repeated, and no timing or benchmark data were collected. We therefore make no empirical claims about speed beyond the design analysis that follows.

TABLE IV
VERIFICATION STATUS BY FEATURE

Feature	Test cases	Result (manual)
Encryption, tamper detection, key storage	6	All passed
Master password, lockout, biometrics	9	All passed
Root detection and secure wipe	7	All passed
Backup and restore (local, Drive, wrong passphrase, tampering)	8	All passed
Password generator and strength meter	5	All passed
Document tools (editor, compression, PDF-to-DOCX, organizer)	5	All passed
Platform builds (six targets)	6	All passed
Database corruption, failed migration	0	Not tested
Performance (memory, scrolling, crypto timing)	0	No formal benchmarks

B. Design-Level Expectations (Not Measured)

Because SQLite stores an entry of about 1–2 KB, capacity is bounded only by device storage; roughly 500,000 entries fit per gigabyte. Lists are built lazily (ListView.builder), and modern mobile processors accelerate AES in hardware, so per-field encryption is expected to take well under a millisecond and vault opening to feel instantaneous. Password generation, being linear in length, takes microseconds. These are expectations derived from the design, not benchmark results, and formal measurement is planned.

C. Comparative Positioning

D.

Table V positions PDManager against typical cloud-centric managers. The comparison is qualitative and reflects design goals; it is not a head-to-head benchmark against a specific product.

TABLE V QUALITATIVE COMPARISON

Aspect	Typical cloud-centric manager	PDManager
Primary storage	Provider-hosted vault	Local SQLCipher database
Backup / sync	Provider servers; multi-device sync often paid	User-controlled encrypted files (local / Drive)
Offline use	Sometimes limited	Fully functional
Documents	Mostly static attachments	In-vault edit, convert, split
Multi-device sync	Built in	Planned
Maturity	Established, widely reviewed	Prototype; not independently audited

E. Security Properties Achieved

- *Data at rest*: the database is AES-256 encrypted with SQLCipher, so a stolen or rooted device yields an unreadable

file without the vault keys.

- *Ciphertext-only backups*: only encrypted payloads reach Google Drive or the local backup folder.
- *Limited plaintext lifetime*: the field key is held in memory during a session and individual fields are decrypted only when viewed; keys and biometric material sit in the platform keystore.
- *Compromised-device response*: on a rooted or jailbroken device the keys and all application data are erased before the vault can be opened; only independently encrypted backups survive.

VIII. SECURITY ANALYSIS

TABLE VI THREATS AND MITIGATIONS

Threat	Mitigation in PDManager
Stolen or rooted device; file access	SQLCipher plus field-level AES-GCM; keys held in hardware keystore, not in the database
Rooted or jailbroken device (key extraction)	Integrity check at launch and resume; keys destroyed and all app data wiped before vault access (detection can be evaded)
Offline guessing of master password	Memory-hard Argon2id with random 16-byte salt; only a verifier is stored
Online guessing	Lockout after 5 failures for 5 minutes, mirrored in Firebase
Tampering with stored data	128-bit GCM tag verified on every decryption
Cloud or backup-provider breach	Backups encrypted on-device; provider holds only ciphertext
Weak or reused passwords	Generator and strength meter (reuse detection not yet implemented)
Network interception	TLS in transit; data encrypted before transmission

Residual risks remain. Security depends on the integrity of the host device, since a rooted or jailbroken device or a bypassed secure enclave weakens every layer. Root detection with secure wipe (Section IV-G) reduces this exposure but is heuristic: tools that conceal root or hook the integrity check can evade it, and an attacker who extracted the keys before detection is not stopped. Conversely, a false positive, such as a developer device or custom ROM, erases the vault irreversibly, so users must keep current backups. The absence of auto-lock leaves an unlocked session exposed to anyone with physical access. Finally, a backup is only as strong as its passphrase, and PBKDF2 with 100,000 iterations is less costly to attack than Argon2id, so users should choose long, random passphrases.

IX. LIMITATIONS

- *Verification gap*: manual functional tests only (Table IV); no automated tests, no independent replication and no formal performance or robustness testing.
- *Session security*: no inactivity auto-lock; no reuse or password-age detection; strength scoring is heuristic.
- *Compromised-device response*: root and jailbreak detection is heuristic and platform specific; the secure wipe is irreversible and applies to Android and iOS, since the desktop and web targets have no equivalent integrity signal; bypass resistance and false-positive rates have not been evaluated beyond the manual test cases.
- *Recovery*: zero-knowledge design means a lost master password cannot be reset, and simultaneous edits on two offline devices can cause sync conflicts that need manual resolution.
- *Dependencies*: Google Drive backups are bound by the user's quota and service availability; Firebase and Clerk are third-party services.
- *Cross-platform variance*: biometrics (Windows Hello, Touch ID, fingerprint, Face ID) and secure storage (Keystore, Keychain, Windows Credential Manager) differ in behavior and hardware backing across platforms.
- *Integration*: no native OS autofill or browser extensions.
- *Missing features*: hardware security keys (WebAuthn/U2F), breach monitoring, emergency access and cryptographically secure vault sharing.

X. FUTURE WORK

- Browser extensions for auto-fill without weakening security.

- Passkey support as a phishing-resistant alternative to master passwords.
- Multi-device synchronization across mobile, desktop and web.
- Breach monitoring through services such as Have I Been Pwned.
- Local, offline AI for categorization and strength analysis so that data never leaves the device.
- *Engineering*: inactivity auto-lock, reuse detection, and automated unit, integration and performance tests, including the root-detection wipe.

CONCLUSION

PDManager shows that credential management and document processing can share one encrypted vault. Its layered design (field-level AES-256-GCM, SQLCipher, an Argon2id verifier, hardware-backed keys, a secure wipe on root detection and independently encrypted backups) keeps sensitive data protected at rest and ciphertext-only when it leaves the device, while local tools remove the need to upload documents to third-party sites. The main gap is verification: automated testing, benchmarking and an independent security review are the priority before broader deployment, followed by auto-lock, reuse detection and the features listed in Section X.

REFERENCES

1. M. Dworkin, "Recommendation for block cipher modes of operation: Galois/Counter Mode (GCM) and GMAC," NIST SP 800-38D, 2007.
2. A. Biryukov, D. Dinu, D. Khovratovich and S. Josefsson, "Argon2 memory-hard function for password hashing and proof-of-work applications," RFC 9106, 2021.
3. K. Moriarty, B. Kaliski and A. Rusch, "PKCS #5: Password-based cryptography specification version 2.1," RFC 8018, 2017.
4. OWASP Foundation, "Password Storage Cheat Sheet," OWASP Cheat Sheet Series.
5. Zetetic LLC, "SQLCipher design," sqlcipher.net/design.
6. Google, "Android Keystore system," Android Developers documentation.
7. Apple Inc., "Keychain services," Apple Developer documentation.
8. Google, "Flutter documentation," docs.flutter.dev.
9. Google, "Firebase documentation: Cloud Firestore and Cloud Functions," firebase.google.com/docs.
10. P. A. Grassi et al., "Digital identity guidelines: Authentication and lifecycle management," NIST SP 800-63B, 2017.
11. LastPass, "Notice of recent security incident," LastPass Blog, Dec. 2022.