PassDork: Architecture and Security Analysis of an Offline Password Manager

Bhavya Sharma[1], Km Prachi Kasana[2], Sneha Kathayat[3], Ms. Priyanka Verma[4]
[1,2,3]*Department of Computer Science and Engineering, Mahatma Gandhi Missions's College of Engineering and Technology Noida, India*
[4]*Under the guidance, Mahatma Gandhi Missions's College of Engineering and Technology Noida, India*

***Abstract*—This paper presents a technical analysis of PassDork, an offline password manager that generates strong passwords and stores them using encrypted local file storage. Rather than transmitting credentials over networks like cloud- based alternatives, PassDork performs all cryptographic operations locally on the user's machine. This design choice fundamentally changes the threat model no remote interception possible, no third-party server exposure. The analysis examines the system's implementation: PBKDF2-based key derivation with 100,000 iterations, Fernet symmetric encryption for vault storage, bcrypt hashing for master password verification, and the password generation algorithm using Sanskrit transliterated words. Entropy calculations show passwords achieve approximately 85-95 bits depending on format. The authentication system includes account lockout after 5 failed attempts and an optional data destruction feature triggered after configurable thresholds. This paper focuses on the engineering decisions that make offline password management secure rather than the linguistic aspects of the Sanskrit word list, which are covered in prior work [8].**

***Index Terms*—password manager, offline security, encrypted storage, PBKDF2, Fernet, entropy**

I. INTRODUCTION

1.1 The Problem

Users today maintain credentials for dozens of online services. Remembering unique, strong passwords for each is cognitively impossible. The common solution reusing passwords creates a single point of failure. When one service gets breached, attackers use the same credentials to access other accounts. This cascading failure is why credential stuffing attacks are so effective. Password managers address this by storing unique, high- entropy passwords in an encrypted vault, protected by a single master password. The user only needs to remember one strong password; the manager handles the rest.

1.2 Cloud vs. Local

Most mainstream password managers (1Password, LastPass, Bitwarden) store encrypted vaults on cloud servers. This enables cross-device synchronization but introduces network-based risks. Even when encrypted, credentials traverse networks, creating interception opportunities. The cloud provider becomes a target.

Offline managers like PassDork store everything locally. No network traffic means no interception risk. The vault exists only on the user's machine. This approach sacrifices synchronization credentials don't automatically appear on other devices but removes an entire category of attack vectors.

1.3 This Paper's Focus

The prior publication on PassDork [8] examined the Sanskrit word generation methodology and its memorability advantages. This paper analyzes the implemented system: the encryption architecture, the key derivation process, the authentication mechanisms, and the security properties of local-only operation. The goal is to understand how the engineering decisions support secure password management.

II. BACKGROUND

2.1 Sanskrit-Based Generation

PassDork generates passwords using Sanskrit words in transliterated form as memorable base components. The password format combines one to three words with special character separators and

four random digits. For example: 'Pustaka@Deepa#1234'. Prior work established that this linguistic approach aids memorability while providing adequate entropy [8]. This paper focuses on the cryptographic and security implementation rather than the linguistic methodology.

2.2 Cryptographic Foundations

Use Three cryptographic primitives form the backbone of PassDork's security:

- PBKDF2 transforms the master password into an encryption key through iterative hashing. Each iteration adds computational cost, making brute-force attacks more expensive. The iteration count is a trade- off: too few provides weak protection; too many creates unacceptable authentication delay. PassDork uses 100,000 iterations with SHA-256.
- Fernet is the encryption format used for the vault. It combines AES-128-CBC with HMAC-SHA256 for authenticated encryption both confidentiality and integrity protection. If an attacker modifies the encrypted vault, decryption fails rather than producing garbage.
- Bcrypt hashes the master password for authentication. Unlike PBKDF2, bcrypt is specifically designed for password storage with a cost factor that can be increased over time [7]. PassDork uses cost factor 12, requiring roughly 250ms per verification.

III. IMPLEMENTATION ANALYSIS

3.1 Module Architecture

The application is organized into five primary modules:

- auth_system.py: Master password creation and verification using bcrypt
- password_vault.py: Encrypted storage of credential entries
- password_generator.py: Password creation using the Sanskrit word list
- entropy_calculator.py: Password strength analysis
- gui_manager.py: User interface coordination

This separation isolates cryptographic operations from presentation logic. The GUI never directly accesses encryption keys; it delegates to the vault module.

3.2 File Storage

PassDork stores all data in `~/.passdork/` with permissions set to 0o700 (owner-only read/write). Three files exist:

- auth.json: Contains the bcrypt hash of the master password
- vault.enc: The encrypted password database
- salt: 16 bytes of random data for PBKDF2 key derivation the code explicitly sets file permissions after writing:

```python
self.auth_file.chmod(0o600)
self.vault_file.chmod(0o600)
salt_file.chmod(0o600)
```

This prevents other users on multi-user systems from reading sensitive data.

3.3 Key Derivation

The `_derive_cipher()` method in password_vault.py derives the encryption key:

```python
kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(), length=32,
                salt=salt, iterations=100000,
)
            key = base64.urlsafe_b64encode(
    kdf. Derive (self.master_password.encode('utf-8'))
)
self.cipher = Fernet(key)
```

The 32-byte (256-bit) key exceeds AES-128 requirements, providing margin. The salt is generated once on first run and persisted critical because changing the salt would make the stored vault unencryptable.

3.4 Encryption Process

When adding a password, the vault:

1. Loads existing entries from `vault.enc` (decrypted in memory)
2. Appends the new entry with timestamp
3. Serializes to JSON
4. Encrypts using the Fernet cipher
5. Writes back to `vault.enc`

The same process applies to retrieval decryption happens in memory, never written to disk. This "decrypt-in- memory" pattern is standard for password

managers but worth noting: the plaintext never touches persistent storage after the initial encryption.

IV. PASSWORD GENERATION

4.1 Algorithm

The password_generator.py module creates passwords in three formats:

Format	Structure	Example
Single	Word + special character + 4 digits	Pustaka@1234
Two words	Word + special character + Word + special character + 4 digits	Pustaka@Deep a#1234
Three words	Word + special character + Word + special character + Word + special character + 4 digits	Pustaka@Deep a#Pushpa$1234

The special character set is limited to seven: `@ # $ % & !
*`. This restricted set avoids characters that cause problems in certain web forms while still providing sufficient variety for complexity requirements.

4.2 Entropy Calculation

The entropy calculator computes Shannon entropy:
$\log_2$(charset_size^length). For the three-word format:

- 18 lowercase letters from Sanskrit words (~4.2 bits per character average accounting for word selection)
- 3 special characters (~2.8 bits each)
- 4 digits (~1.4 bits each)

Total entropy: approximately 85-95 bits depending on word lengths. This exceeds the 80-bit threshold commonly considered secure against brute-force.

4.3 Strength scoring

The entropy calculator combines multiple factors:

- Entropy (40%): The mathematical search space
- Length (30%): Penalizes passwords under 12 characters
- Variety (20%): Rewards using all character types
- Pattern (10%): Penalizes sequential characters like "abc" or repetition like "aaa"

The weighted score produces a 0-100 rating with recommendations for improvement. This is more informative than entropy alone because it catches predictable patterns that reduce effective security even when theoretical entropy is adequate.

V. AUTHENTICATION SECURITY

5.1 Master Password Verification

The auth_system.py module stores the bcrypt hash:

```python
salt = bcrypt.gensalt(rounds=12)
hashed = bcrypt.hashpw(password.encode("utf-8"), salt)
```

To verify later:

```python
is_valid = bcrypt.checkpw(password_bytes, stored_hash)
```

Bcrypt's cost factor of 12 means approximately 250ms per verification on modern hardware. This creates meaningful computational barriers for attackers while remaining acceptable for legitimate users.

5.2 Brute-Force Protection

The authentication system implements progressive lockout:

- Maximum 5 failed attempts before lockout
- 300-second (5-minute) lockout duration
- All attempts logged with timestamps

After 5 failures, the user must wait 5 minutes before attempting again. This makes exhaustive search impractical if an attacker could try 10,000 passwords per second, the 5- attempt limit means they'd need thousands of lockout periods to exhaust a reasonable search space.

5.3 Data Destruction Feature

A more aggressive option exists: automatic data deletion after failed attempts. Configurable from 1-5 attempts, this feature deletes both auth.json and vault.enc when triggered:

```Python
If delete_threshold > 0 and attempts_data["failed_attempts"] >= delete_threshold:
self.delete_all_data()
```

This defends against offline attacks where an attacker has physical access or root privileges and can attempt unlimited passwords without network rate limiting. The vault destroys itself rather than permitting unlimited guesses.

VI. SECURITY TRADE-OFFS

6.1 What Offline Removes

The local-only design eliminates several attack categories:

- MITM attacks: No network traffic means no interception
- Cloud breaches: No remote server to compromise
- DNS hijacking: No external connections to hijack
- Service downtime: No dependency on third-party availability

6.2 What Remains

Offline operation doesn't protect against:

- Physical access to the device
- Malware with user-level privileges
- Memory extraction attacks
- Shoulder surfing

The master password must still be strong if an attacker can run password-guessing software on the local machine, the 5-attempt lockout doesn't apply.

6.3 Privacy Advantages

Local storage means no data leaves the user's machine:

- No usage statistics sent anywhere
- No credential metadata exposed
- No behavioral analysis possible
- No "freemium" data harvesting

This matters for users with privacy concerns. The only data that exists is what the user explicitly saves.

VII. SECURITY CONSIDERATIONS

7.1 Current Strengths

The implementation demonstrates good practices:

- Authenticated encryption prevents tampering
- PBKDF2 with 100k iterations exceeds minimum recommendations
- Python's secrets module provides cryptographically secure randomness
- Explicit file permissions restrict access
- Lockout limits guessing attempts

7.2 Potential Improvements

Several areas could be enhanced:

1. Memory exposure: The master password remains in memory during the session. While standard for GUI applications, malicious code or memory dumps could extract it. More aggressive memory clearing after use would help.
2. No memory locking: Python doesn't support mlock(), so sensitive data might swap to disk on systems with enabled swap. This is a limitation of the runtime rather than the implementation.
3. Character set expansion: Generated passwords lack uppercase letters. Adding mixed case would increase entropy while maintaining the word-based structure.
4. Master password strength: The minimum 8-character requirement is minimal. The entropy calculator could enforce stronger master passwords.

VIII. CONCLUSION

PassDork implements secure offline password management using established cryptographic primitives. The PBKDF2 key derivation with 100,000 iterations, Fernet authenticated encryption, and bcrypt master password hashing meet modern security standards. Passwords achieve 85-95 bits of entropy in the three-word format well above the 80-bit threshold for high-security applications [1].

The offline-first architecture fundamentally changes the threat model by eliminating network-based attacks. This trade-off sacrificing cross-device synchronization for reduced attack surface aligns with the needs of privacy- conscious users or those with elevated threat models.

The implementation is not without areas for potential enhancement: memory protection, stronger master password enforcement, and character set expansion could strengthen the system further. However, these represent incremental improvements rather than fundamental flaws. This analysis focused on the engineering decisions enabling secure local password management. The Sanskrit word generation methodology and its memorability advantages are covered in prior work [8]. Together, the two papers present a complete picture of PassDork's design: linguistic foundation and security implementation.

REFERENCES

[1] J. Bonneau, C. Herley, P. C. van Oorschot, and F. Stajano, “The quest to replace passwords: A framework for comparative evaluation of web authentication schemes,” in Proc. IEEE Symp. Security and Privacy (SP), 2012, pp. 553–567.

[2] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith, "Why Eve and Mallory love Android: An analysis of Android SSL (in)security," in Proc. ACM Conf. Computer and Communications Security (CCS), 2012, pp. 50–61.

[3] C. Herley, "Passwords: A case history," IEEE Security & Privacy, vol. 14, no. 5, pp. 8–10, 2016.

[4] M. L. Mazurek, P. G. Kelley, S. Komanduri, T. Vidas, L. F. Cranor, and N. Christin, "The password divide: Differences in security behavior across demographic groups," in Proc. ACM Conf. Computer and Communications Security (CCS), 2013, pp. 293–304.

[5] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch, "PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2)," RFC 8018, Jan. 2017.

[6] Python Cryptographic Authority, "Fernet (symmetric encryption)," [Online]. Available: https://cryptography.io/en/latest/fernet/. [Accessed: Apr. 13, 2026].

[7] R. Morris and K. Thompson, "Password security: A case history," Communications of the ACM, vol. 22, no. 11, pp. 594–597, Nov. 1979.

[8] B. Sharma, K. P. Kasana, S. Kathayat, and N. Shrivastava, "Leveraging Sanskrit for strong and memorable passwords," International Journal of Innovative Research in Technology (IJIRT), vol. 6, no. 6, 2026.