SecureDataEncryption

v2.0 Technical Documentation & Security Audit

Quantum-Resistant Multi-Cipher Encryption Tool

with Hybrid ML-KEM-768 Post-Quantum Key Encapsulation

Classification: Internal Technical Document

Author: 404SecurityNotFound

Security Audit: Independent Code Review

Date: February 2026

Version: 2.0.0

Table of Contents

1. Executive Summary

SecureDataEncryption v2.0 is a text encryption tool designed for people who need to protect sensitive information -- credentials, private notes, configuration secrets, API keys, or any block of text -- without that data ever touching a disk.

Unlike general-purpose tools like GPG or age, this tool is purpose-built for a specific workflow: encrypt a block of text, get back a string you can safely store or transmit, and decrypt it later with your password. The encrypted output is displayed once and automatically cleared after 60 seconds.

The tool offers four encryption modes of increasing strength, from single-cipher AES-256-GCM through to a maximum-security combination of cipher chaining plus hybrid post-quantum key encapsulation using NIST FIPS 203 ML-KEM-768.

This document serves three purposes:

- A complete technical reference for developers and security professionals
- An independent security audit with actionable findings
- A plain-English guide for non-technical users who need to understand what the tool does and why they should trust it

2. Why This Tool Exists

The Problem

You have a block of sensitive text. Maybe it's an API key for a production system. Maybe it's a password list, a private note, or a configuration file with database credentials. You need to encrypt it so you can store or transmit it safely.

Your options today are surprisingly poor:

- GPG: Powerful but complex. Designed for email, not quick text encryption. The learning curve is steep, the output format is opaque, and there's no post-quantum support.
- age: Simple and elegant, but file-oriented. No GUI, no cipher selection, no PQ, no auto-clear. Your encrypted output sits in terminal scrollback forever.
- openssl enc: Uses CBC mode by default (no authentication!). No KDF tuning. A single wrong flag and your encryption is silently broken.
- Online encryption tools: Your plaintext hits someone else's server. That's not encryption, that's trust.

What We Built Instead

A tool that is obsessive about three things:

1. Cryptographic depth -- not one algorithm, but layers. AES-256-GCM and ChaCha20-Poly1305 can be chained so that if either is broken, your data survives. ML-KEM-768 hybrid mode adds post-quantum resistance on top.

2. Ephemeral by design -- encrypted output appears once, counts down from 60 seconds, then vanishes. Clipboard is wiped. Key material is zeroed. Nothing is written to disk. The tool treats your data like a self-destructing message.

3. Accessible security -- a modern terminal GUI with dropdown cipher selection, real-time password strength feedback, and one-click operation. You don't need to know what AES-GCM means to use it safely. The defaults are secure.

Who Is This For?

- Security professionals who need to encrypt text blocks without touching disk
- Developers storing secrets that don't belong in plaintext config files
- System administrators sharing credentials through secure channels
- Privacy-conscious users who want encryption they can verify and understand
- Anyone preparing for the post-quantum era who wants hybrid protection today

3. How It Works -- Plain English

The Lock-and-Key Analogy

Think of encryption like a lockbox with a unique lock:

Your text is the item inside. Your password is the key. The encrypted output is the locked box -- anyone can carry it, but nobody can see inside without the key.

What makes this lockbox special is that every time you lock something, the lock itself changes. Even if you lock the same item with the same key twice, the two locked boxes look completely different from the outside. An observer can't tell if two boxes contain the same item or different items.

What Happens When You Hit 'Encrypt'

Step 1 -- Key Strengthening (takes about 1 second)
Your password is deliberately put through a slow, memory-intensive process called Argon2id. This turns your human-memorable password into a 256-bit cryptographic key. The slowness is the point -- it means an attacker trying millions of passwords would need millions of seconds.

Step 2 -- Encryption
Your text is encrypted with AES-256-GCM (the same algorithm banks and governments use). A random "nonce" (number used once) ensures the output is unique every time. A 16-byte authentication tag is appended that detects any tampering.

Step 3 -- Packaging
The salt (used in Step 1), the nonce (used in Step 2), and the encrypted text are bundled into a single base64 string safe for copy/paste.

Step 4 -- Display and Forget
The result appears in the output area. A 60-second countdown begins. When it reaches zero, the output is erased, the clipboard is cleared, and the key material is overwritten with zeros in memory.

What Happens When You Hit 'Decrypt'

The process reverses: the tool reads the header to determine which cipher and KDF were used (this is automatic -- you don't need to remember your settings), derives the same key from your password plus the stored salt, and decrypts. If the password is wrong or the data has been tampered with, the authentication tag check fails and you get a generic error message. The tool deliberately does not tell you which one went wrong -- that would help attackers.

4. Architecture Deep Dive

Module Structure

```
secure_encryption/
  __init__.py        Package version (2.0.0)
  __main__.py        Entry point -- auto-detects GUI vs CLI
  gui.py             Textual TUI (733 lines)
  cli.py             Argparse CLI (224 lines)
  core/
    ciphers.py       Cipher strategy pattern -- AES-GCM, ChaCha20
    kdf.py           KDF strategy pattern -- Argon2id, Scrypt
    pipeline.py      Orchestration -- chaining, hybrid PQ, key mgmt
    formats.py       Versioned binary format -- serialize/deserialize
    memory.py        mlock, secure zeroing, SecureBuffer
    validation.py    Password scoring (0-100), input checks
```

Design Principles

1. Strategy Pattern for Ciphers and KDFs

Each cipher (AES-256-GCM, ChaCha20-Poly1305) and each KDF (Argon2id, Scrypt) implements an abstract base class. The pipeline doesn't know or care which concrete implementation it's using. This means adding a new cipher (e.g., AES-256-CBC-HMAC for legacy compat) requires only a new class and a registry entry -- zero changes to the pipeline.

2. Self-Describing Ciphertext Format

The 6-byte header encodes everything needed to decrypt: format version, cipher ID, KDF ID, and flags. The decryptor reads the header and configures itself automatically. This means the tool can always decrypt ciphertexts from any version 2.x configuration without the user needing to remember or specify settings.

3. Separation of Encryption from Presentation

The core/ package has zero dependency on GUI or CLI. You can import EncryptionPipeline in a script, a web app, or a test harness. The GUI and CLI are thin wrappers that handle user interaction and delegate all crypto to the pipeline.

5. Cryptographic Internals

5.1 Ciphertext Binary Format (Version 2)

```
Byte 0:     0x02 (version)
Byte 1:     cipher_id (0x01=AES-GCM, 0x02=ChaCha20, 0x03=Chained)
Byte 2:     kdf_id    (0x01=Scrypt, 0x02=Argon2id)
Byte 3:     flags     (bit0=chained, bit1=hybrid_pq)
Bytes 4-5:  reserved  (0x0000)
Bytes 6+:   payload

Single cipher payload:  [16B salt][12B nonce][ciphertext + 16B tag]
Chained payload:        [16B salt][12B nonce1][12B nonce2][ciphertext + tag]
Hybrid PQ prefix:       [2B kem_ct_len][kem_ciphertext]  (prepended to payload)
```

The header fields are authenticated via the AAD (Associated Authenticated Data) parameter of the AEAD cipher. The AAD is constructed as: pack('!BBBB', version, cipher_id, kdf_id, flags). This cryptographically binds the algorithm choices to the ciphertext -- an attacker cannot modify the header to trick the decryptor into using a different algorithm without the authentication tag verification failing.

5.2 AES-256-GCM

NIST SP 800-38D. 256-bit key, 96-bit random nonce, 128-bit authentication tag. Hardware-accelerated via AES-NI on modern x86/ARM processors. Provides IND-CCA2 security under the assumption that the block cipher is a PRP.

Nonce generation: os.urandom(12) -- 96 bits of cryptographic randomness per encryption. The birthday bound for nonce collision is 2^48 encryptions under the same key, which is not a concern for our use case (each encryption derives a unique key from a random salt).

5.3 ChaCha20-Poly1305

RFC 8439. 256-bit key, 96-bit random nonce, 128-bit Poly1305 tag. Constant-time software implementation -- immune to cache-timing side channels that can affect AES table lookups on hardware without AES-NI. Preferred by WireGuard, Google QUIC, and Cloudflare.

5.4 Cipher Chaining

When chaining is enabled, the pipeline always uses the fixed order AES-256-GCM (inner) then ChaCha20-Poly1305 (outer). Two independent 256-bit keys are derived from the master key using HKDF-Expand (RFC 5869) with distinct info strings:

```
  master_key = KDF(password, salt)
  key_aes    = HKDF-Expand(master_key, info='cipher-key-0', length=32)
  key_chacha = HKDF-Expand(master_key, info='cipher-key-1', length=32)
```

The inner encryption produces ciphertext_1 = AES-GCM(key_aes, plaintext, aad). The outer encryption wraps it: ciphertext_2 = ChaCha20-Poly1305(key_chacha, ciphertext_1, aad). Both layers use independent nonces.

5.5 Key Derivation

Argon2id (default): RFC 9106, OWASP recommended. Parameters: time_cost=3, memory_cost=65536 (64 MiB), parallelism=4. Argon2id combines Argon2i (data-independent memory access, resists side-channel attacks in the first pass) with Argon2d (data-dependent access, resists GPU attacks in subsequent passes).

Scrypt (alternative): RFC 7914. Parameters: n=2^17 (131072), r=8, p=1. Memory-hard via sequential memory access patterns. Well-established but Argon2id is preferred for new designs.

Both KDFs produce a 256-bit key from the password and a random 128-bit salt.

5.6 Hybrid Post-Quantum (ML-KEM-768)

NIST FIPS 203 (August 2024). ML-KEM (Module-Lattice Key Encapsulation Mechanism) is based on the hardness of the Module Learning With Errors (MLWE) problem -- a lattice problem that no known quantum algorithm can solve efficiently.

ML-KEM-768 provides 192-bit classical security and is the recommended parameter set for general use. Key sizes: public key 1184 bytes, secret key 2400 bytes, ciphertext 1088 bytes, shared secret 32 bytes.

Our hybrid construction:

1. password_key = Argon2id(password, salt)
2. (kem_ct, kem_ss) = ML-KEM-768.Encaps(public_key)
3. final_key = HKDF(password_key || kem_ss, salt, info='hybrid-pq-v1')
4. ciphertext = AES-GCM(final_key, plaintext)

The HKDF extract step combines entropy from both the password-derived key and the KEM shared secret. An attacker must break BOTH the password AND ML-KEM to recover the plaintext. If either layer holds, the data is safe. This is the NIST-recommended approach to hybrid PQ migration: don't replace classical crypto, layer PQ on top.

5.7 Memory Protection Model

The memory.py module provides best-effort protection via:

- mlock(2): Prevents the OS from swapping buffer pages to disk. Uses ctypes to call libc directly. Non-fatal if it fails (e.g., due to RLIMIT_MEMLOCK).

- Secure zeroing: Overwrites bytearray buffers byte-by-byte after use. Resistant to dead-store elimination in CPython because the interpreter doesn't optimize at that level.

- SecureBuffer context manager: Combines mlock + auto-zero in a with-statement pattern.

Limitation: Python str objects are immutable and cannot be reliably zeroed. The password, once passed as a string, exists in the Python heap until GC. See the Security Audit for the full analysis of this limitation.

6. Security Audit Report

Methodology: Manual line-by-line code review of all 8 source modules (1,736 lines total). Reviewed for: cryptographic misuse, memory safety, information leakage, injection vectors, format parsing vulnerabilities, timing attacks, and backdoors. No automated SAST tools were used -- this is a human adversarial review.

6.1 Findings

[CRITICAL] Key Zeroing Creates Copies, Does Not Zero Originals

`Location: pipeline.py:213-216, 96-98, 298-301`

In pipeline.py lines 213-216, the key cleanup code does:

 for k in keys: buf = bytearray(k); secure_zero(buf)

This creates a NEW bytearray from the immutable bytes object 'k', then zeros the copy. The original key bytes remain in the Python heap. The same pattern appears at lines 96-98 and 298-301. This gives a false sense of security -- the actual key material is never zeroed.

Recommendation: Use bytearray throughout the key derivation chain instead of bytes. Modify KDF.derive() and HKDF to return bytearray, and modify cipher.encrypt/decrypt to accept bytearray keys. This is a fundamental CPython limitation but can be mitigated.

[CRITICAL] Password String Is Immutable and Cannot Be Zeroed

`Location: pipeline.py:170, kdf.py:63, gui.py:544`

The password parameter flows as a Python str through the entire codebase: gui.py reads it from Input.value (str), passes it to EncryptionPipeline.encrypt(plaintext, password) which passes it to KDF.derive(password, salt) which calls password.encode('utf-8'). At no point is the password stored in mutable memory. The str object persists in CPython's heap until garbage collection, and even then the memory page may not be cleared.

Recommendation: Accept password as bytes or bytearray at the API boundary. The GUI/CLI should encode immediately and pass bytearray, which can be zeroed after use. This is the standard mitigation in Python crypto tools (e.g., paramiko does this).

[MEDIUM] secure_key Context Manager Yields Immutable Copy

`Location: memory.py:117-128`

In memory.py:126, secure_key yields bytes(buf.data), creating an immutable copy. The SecureBuffer correctly zeros buf.data on exit, but the yielded bytes object is a separate immutable allocation that cannot be zeroed. Code using 'with secure_key(k) as key' would hold an unzeroable reference for the duration of the block.

Recommendation: Yield buf.data (the bytearray) directly, or document that the context manager protects against swap only, not heap persistence.

[MEDIUM] No Payload Length Validation During Decrypt Parsing

`Location: pipeline.py:255-280`

In pipeline.py:255-280, the payload is parsed using sequential offset reads without checking that sufficient bytes remain. A truncated ciphertext could produce empty salt, nonce, or KEM fields via silent slice truncation. While this fails safely (the auth tag check will reject it), the error message will be 'incorrect password' rather than 'truncated ciphertext', which makes debugging harder.

Recommendation: Add explicit length checks: 'if len(payload) < offset + expected_size: raise ValueError("Truncated ciphertext")'

[LOW] libc Loader Re-Attempts on Every Call After Failure

`Location: memory.py:25-48`

In memory.py:28, '_load_libc()' checks 'if _libc is not None: return'. If the initial load fails, _libc stays None, causing every subsequent call to re-attempt the dlopen. This is a minor performance issue, not a security issue.

Recommendation: Use a sentinel value (e.g., _libc = False) to distinguish 'not yet attempted' from 'attempted and failed'.

[LOW] KEM Ciphertext Length Field Limited to 65535 Bytes

`Location: pipeline.py:201, formats.py:19`

The KEM ciphertext length is encoded as a 2-byte unsigned short (!H). ML-KEM-768 ciphertext is 1088 bytes, well within range. However, future KEMs (e.g., Classic McEliece with ~200KB ciphertexts) would overflow this field silently.

Recommendation: Document this limitation. If Classic McEliece support is planned, use a 4-byte length field (!I) in format version 3.

[LOW] Clipboard Clear May Not Defeat Clipboard Managers

`Location: gui.py:677`

pyperclip.copy('') overwrites the system clipboard, but clipboard history managers (macOS Universal Clipboard, Windows Clipboard History, KDE Klipper, various third-party tools) may retain the previous value in a separate history store.

Recommendation: Document this limitation. Users with clipboard managers should disable history for sensitive operations, or use the manual copy method.

[INFO] No Rate Limiting on Decryption Attempts

`Location: pipeline.py:222-303`

An attacker with the ciphertext can attempt unlimited offline password guesses. The KDF cost (Argon2id: ~1 second per attempt, 64MB per attempt) provides substantial protection, but there is no exponential backoff or lockout mechanism.

Recommendation: This is inherent to offline encryption (no server to enforce rate limits). The mitigation is KDF cost + password strength requirements. Document that users should use 16+ character passwords for high-value data.

[INFO] Reserved Header Bytes Not Validated on Decode

`Location: formats.py:64`

The 2-byte reserved field (bytes 4-5) is read but not checked to be zero. A modified ciphertext with non-zero reserved bytes would still decrypt successfully. This is not a vulnerability because the reserved field is not part of the AAD.

Recommendation: No action needed. Non-zero reserved bytes are forward-compatible (a future version may use them). If strict validation is desired, add a check.

[INFO] Exception Handling in Decrypt Catches All Exceptions

`Location: gui.py:587-591`

gui.py:587 catches bare 'except Exception' and returns a generic error. This is correct security behavior (prevents information leakage / padding oracle attacks) but may mask bugs during development.

Recommendation: Add debug-mode logging (disabled in production) that logs the actual exception type for diagnostic purposes.

6.2 Positive Findings

[POSITIVE] Cryptographically Secure Random Generation

`Location: ciphers.py:58,75; kdf.py:42`

All nonces and salts use os.urandom(), which reads from /dev/urandom on Linux and CryptGenRandom on Windows. No weak PRNGs.

[POSITIVE] Contextual AAD Binds Algorithm Choices

`Location: formats.py:38-40`

The AAD includes version, cipher_id, kdf_id, and flags. An attacker cannot modify the header to downgrade the cipher without the auth tag failing. This prevents algorithm confusion attacks.

[POSITIVE] HKDF-Expand Uses Distinct Info Strings

`Location: pipeline.py:89-93`

Chained mode derives keys using 'cipher-key-0' and 'cipher-key-1' as HKDF info parameters. This provides proper domain separation -- knowing key_0 does not help derive key_1.

[POSITIVE] Generic Error Messages on Decrypt Failure

`Location: gui.py:588-591; cli.py:218-221`

Both GUI and CLI return 'incorrect password or corrupted data' without distinguishing between wrong password, tampered ciphertext, or format errors. This prevents padding oracle and error oracle attacks.

[POSITIVE] Password Never Accepted via CLI Arguments by Default

`Location: cli.py:71-73, 158-164`

The -p/--password flag is suppressed (hidden from help) and triggers a visible stderr warning when used. The default path always uses getpass for non-echoing interactive input.

[POSITIVE] No Backdoors, No Telemetry, No Network Access

`Location: All modules`

Complete code review confirms: no outbound network calls, no telemetry, no analytics, no logging to files, no temp file creation, and no eval/exec/import of dynamic code. The tool is fully offline.

[POSITIVE] Argon2id Parameters Follow OWASP 2024 Guidelines

`Location: kdf.py:57, kdf.py:85`

Default Argon2id params (t=3, m=64MiB, p=4) match the OWASP 2024 recommendation for interactive applications. Scrypt default n=2^17 also matches OWASP guidance.

7. Testing & Verification

Automated Test Suite: 86 Tests

Run with: python -m pytest tests/ -v

All tests pass in 1.01 seconds.

Test File	Count	Coverage
`test_ciphers.py`	18	AES/ChaCha encrypt-decrypt, wrong key, tampered data, unique nonces, empty/large payloads
`test_kdf.py`	12	Argon2id/Scrypt derivation consistency, cross-password independence, custom key lengths
`test_formats.py`	9	Serialize-deserialize roundtrip, flag preservation, invalid base64, wrong version, truncated input
`test_validation.py`	14	Password scoring (weak/fair/strong/excellent), missing char classes, empty input, Unicode, 10MiB limit
`test_pipeline.py`	28	All cipher+KDF combos, chaining, hybrid PQ, wrong password, wrong KEM key, self-describing format
`test_memory.py`	5	Secure zeroing, SecureBuffer context manager, buffer sizing

Manual Verification Procedures

Roundtrip Test: Encrypt text with a known password, then decrypt with the same password. Verify the output matches the input exactly, including whitespace and special characters.

Wrong Password Test: Encrypt, then attempt to decrypt with a different password. Verify the tool returns 'incorrect password or corrupted data' and does not leak partial plaintext.

Tamper Detection Test: Encrypt, modify a single character in the base64 output, attempt to decrypt. Verify rejection.

Uniqueness Test: Encrypt the same text with the same password twice. Verify the two encrypted outputs are different (different random salt and nonce).

Cross-Cipher Test: Encrypt with AES-256-GCM, then decrypt (the tool auto-detects the cipher from the header). Encrypt with ChaCha20-Poly1305, decrypt the same way. Both should work without specifying the cipher during decryption.

Hybrid PQ Test: Generate an ML-KEM-768 keypair. Encrypt with the public key. Decrypt with the secret key and correct password. Verify success. Then try with the wrong secret key. Verify rejection.

8. Competitive Comparison

Feature matrix comparing SecureDataEncryption v2.0 against common alternatives:

Feature	This	age	GPG	Pico.	openssl
Hybrid PQ (ML-KEM-768)	Yes	No	No	No	No
Cipher chaining	Yes	No	No	No	No
One-time auto-clear	Yes	No	No	No	No
Memory locking (mlock)	Yes	No	Part.	No	No
Modern TUI / GUI	Yes	No	No	Yes	No
Argon2id KDF	Yes	No	No	Yes	No
Scrypt KDF	Yes	Yes	No	No	No
AES-256-GCM	Yes	No	Yes	No	Yes
ChaCha20-Poly1305	Yes	Yes	No	Yes	No
Self-describing format	Yes	Yes	Yes	Part.	No
Password strength check	Yes	No	No	No	No
Text-block optimized	Yes	No	No	No	No
No disk writes	Yes	No	Yes	No	No
Authenticated encryption	Yes	Yes	Yes	Yes	CBC!

Key insight: No other tool in this category combines post-quantum hybrid encryption, cipher chaining, and ephemeral output in a user-friendly interface. GPG is the closest in cryptographic depth but has a steep learning curve and no PQ support. age is the closest in simplicity but lacks cipher selection and PQ. Picocrypt has a GUI and Argon2 but no PQ and no cipher chaining.

The unique position of this tool is at the intersection of modern cryptography, post-quantum readiness, and usability -- targeting the specific workflow of encrypting and transmitting sensitive text blocks.

9. Deployment & Operations

Installation

```
git clone https://github.com/404securitynotfound/SecureDataEncryption.git
cd SecureDataEncryption
python -m venv venv && source venv/bin/activate
pip install -r requirements.txt
python -m pytest tests/ -v       # verify 86 tests pass
python secure_data_encryption.py  # launch GUI
```

Operational Security Recommendations

- Use in a terminal that supports secure erase or clearing scrollback history
- Disable clipboard managers during sensitive operations
- Use 16+ character passwords for high-value data (24+ for critical data)
- Enable cipher chaining for data that must remain secure for 10+ years
- Enable hybrid PQ for data that must resist future quantum attacks
- Generate ML-KEM keypairs fresh for each session -- do not reuse across sessions
- Verify the test suite passes before first use: python -m pytest tests/ -v
- Pin dependency versions in production (see requirements.txt)
- Run on a full-disk-encrypted system to protect against cold boot attacks

Dependencies and Supply Chain

The tool depends on five Python packages, all widely used and audited:

cryptography (pyca) -- The standard Python crypto library. Backed by the Python Cryptographic Authority. Wraps OpenSSL/BoringSSL. Regularly audited.

argon2-cffi -- Python bindings for the reference Argon2 C implementation. Winner of the Password Hashing Competition.

textual -- Terminal GUI framework by Textualize. No native code, pure Python.

pyperclip -- Clipboard access. Thin wrapper around platform-native clipboard APIs.

pqcrypto (optional) -- Post-quantum cryptography. Wraps PQClean C reference implementations of ML-KEM. The reference implementation is NIST-validated.

End of Document

SecureDataEncryption v2.0

404SecurityNotFound

February 2026

"The only truly secure system is one that is powered off, cast in a block of concrete, and sealed in a lead-lined room with armed guards." -- Gene Spafford