WHITE PAPER / VOLUME 3

Cryptographic Protocol & System Design

3.1 Zero-Knowledge Local-First Architecture

SecretMemoryLocker is engineered on the principle of **Total Data Sovereignty**. User data never leaves the local environment; all processing and storage occur strictly on the user's device.

The Stateless Model: The application operates as a "Pure Function." It initializes with a null state in volatile memory (RAM).

- **Volatile Session Memory:** Questions, answers, and sensitive data exist only during the active session.
- **Instant Memory Purge:** Upon successful encryption or session termination, the `reset_user_data()` routine securely wipes all sensitive variables from RAM, preventing memory-dump attacks.
- **Database-Free Design:** By eliminating traditional local databases, we remove the risk of "at-rest" data leaks and unauthorized access through third-party storage exploits.

3.2 Deterministic Key Derivation (KDF)

We utilize a deterministic approach to transform human memories into a high-entropy, cryptographically secure master key.

- **Argon2id Integration:** We employ Argon2id (the winner of the Password Hashing Competition) to provide memory-hard hashing. This ensures maximum resistance against GPU and ASIC-based brute-force attacks while maintaining perfect key reproducibility from identical inputs.
- **Authenticated Encryption (AEAD):** For data confidentiality and integrity, we use ChaCha20-Poly1305. This ensures that any unauthorized modification of the encrypted block is detected and rejected during the decryption process.

3.3 The Semantic Normalization Layer

To bridge the gap between rigid cryptography and the nuances of human memory, SecretMemoryLocker includes a Semantic Normalization Layer. This layer ensures that minor input variations do not lead to catastrophic data loss.

- **Human-Factor Mitigation:** The system automatically processes inputs by stripping leading/trailing whitespaces, unifying case sensitivity, and applying Unicode Normalization (NFKD).
- **Deterministic Output:** Whether a user inputs `"New York"` or `"new york "`, the layer normalizes the string before hashing, ensuring the resulting cryptographic key remains consistent across sessions.

3.4 Entropy Stacking & Salt Strategy

The master key is generated through a process of Entropy Stacking, combining multiple cognitive inputs with a physical security factor.

K_{master} = Argon2id($\bigoplus_{i=0}^{n} H(Answer_i)$, $Salt_{file}$, *T, M, P*) Where *T, M, P* are time, memory, and parallelism cost parameters.

The Mathematical Logic:

- **Primary Seed:** The first answer is combined with the hash of a local "secret file" (acting as a unique 256-bit salt).

- **Cascade Hashing:** Subsequent answers are hashed independently.
- **Final Derivation:** The system concatenates these hashes and passes them through a final SHA-256 round to generate the 256-bit Master Key (K_{master}).

> **Entropy Calculation:**
>
> Assuming each of the 5 answers provides ~40 bits of entropy (common for phrases or specific memories), the combined cognitive entropy is ~200 bits. When augmented with the 256-bit file-hash salt, the resulting key achieves full 256-bit cryptographic strength, making it computationally infeasible to crack.

3.5 "Phantom-Step" Mechanics (Cascading Encryption)

The "Phantom-Step" is our proprietary recursive encryption method that ensures data isolation within the question chain.

- **Recursive Dependencies:** Each question in the sequence is encrypted using the hash of the preceding answer.
- **Step-Wise Isolation:** The system is "blind" to Step *N+1* until Step *N* is correctly solved. This prevents an attacker from seeing the full scope of the security chain or identifying the final data payload.
- **Hash-Chaining:** All inputs are bound in a non-linear chain. Any deviation in a single answer breaks the entire recovery cascade, providing a robust defense against step-by-step guessing.

MVP & Reference Implementation

The current version of SecretMemoryLocker is a Reference Implementation (MVP) written in Python. This choice prioritizes logic transparency and rapid cryptographic auditing. Future production releases will migrate performance-critical and memory-sensitive modules to low-level languages (such as Rust) to further reduce the attack surface and enhance memory safety.