CRYPTIFY

The domain of the Project:

Cyber Security

Team Mentors (and their designation):

Mr. Derick(Trainer)

Team Members:

Mr. Dalveer Singh
Mr. Akashdeep Singh

Period of the project

October 2025 to December 2025

Declaration

The project titled "Cryptify" has been mentored by Mr. Derick organised by SURE Trust, from October 2025 to December 2025, for the benefit of the educated unemployed rural youth for gaining hands-on experience in working on industry relevant projects that would take them closer to the prospective employer. I declare that to the best of my knowledge the members of the team mentioned below, have worked on it successfully and enhanced their practical knowledge in the domain.

Team Members:

Mr. Dalveer Singh DS Signature
Mr. Akashdeep Singh AS Signature

Mentor's Name
Mr. Derick

Prof. Radhakumari
Executive Director & Founder
SURE Trust

Table of contents

Executive Summary

1. Project Objective:

To deliver a next-generation file security utility that addresses the threat of future quantum computing. Cryptify provides Post-Quantum Cryptography (PQC) for key exchange, ensuring that data encrypted today remains secure even against future quantum-scale decryption attempts.

2. Methods & Approach:

- **Post-Quantum Key Exchange:** Implements the **ML-KEM (Kyber768)** algorithm via the liboqs library to safely encapsulate shared secrets between users.
- **Hybrid Encryption Architecture:** Uses **AES-GCM** for high-speed symmetric data encryption once the shared secret is established, providing both confidentiality and integrity (authenticated encryption).
- **Integrated Data Compression:** Employs the **Zstandard (zstd)** algorithm to compress data *before* encryption, reducing storage footprint and improving transmission speeds.
- **Memory Safety:** Features a secure_erase utility that proactively overwrites sensitive buffers (like shared secrets and keys) in memory to prevent forensic recovery.
- **Streaming & Performance:** Utilizes a **chunked processing** model (256 KB chunks) and optimized write buffers (4 MB) to handle multi-gigabyte files without crashing system RAM.

3. Key Findings:

The hybrid approach—combining PQC for key protection and AES-GCM for data throughput—successfully provides high security without the typical performance penalties associated with post-quantum algorithms.

4. Recommendations:

Expand the key management system to support hardware security modules (HSMs) or TPM-based storage for the private keys to further harden the local security posture.

5. Executive Highlights:

- **Quantum-Resistant Security**: Unlike traditional tools using RSA or ECC, Cryptify uses ML-KEM, protecting your organization's sensitive data against "harvest now, decrypt later" attacks by quantum computers.

- **End-to-End Privacy:** The tool supports both "Self-Encryption" (for private backups) and "Recipient Encryption" (for secure sharing), enabling a versatile secure communication channel.
- **Performance Optimization**: By integrating zstd compression, the tool often results in smaller encrypted files than the original source, optimizing both storage costs and network bandwidth.
- **User-Centric Interface:** A professional, dark-themed GUI built with customtkinter hides complex cryptographic operations behind a simple "Lock it Down" user experience, reducing the risk of operator error.
- **Enterprise Management**: Includes a robust KeyManager that handles key generation, secure storage, and public key imports, allowing for structured organizational deployment.

1. **Background and Context:**
   The project is situated in the evolving landscape of cybersecurity, where traditional cryptographic standards (like RSA and ECC) are becoming vulnerable to the looming threat of quantum computing. Cryptify is developed as a proactive security tool that integrates Post-Quantum Cryptography (PQC) into a practical, user-friendly application. It leverages the liboqs library, which provides implementations of algorithms evaluated by NIST for quantum resistance. The context of the project is to provide a "Lock it down" solution that is not only effective against current classical threats but is also "future-proofed" against quantum-scale decryption attempts.
2. **Problem Statement and Goals:**
   - **The Quantum Threat:** Current encryption methods rely on mathematical problems (like factoring large integers) that quantum computers can solve efficiently. Cryptify's goal is to mitigate this by implementing ML-KEM (Kyber768) for key encapsulation.
   - **Data Bloat:** Secure files can often be large and cumbersome to transfer. A core goal of this project is to integrate high-speed Zstandard (zstd) compression to reduce file size before the encryption process begins.
   - **Usability Gap:** Many advanced cryptographic tools are restricted to command-line interfaces (CLI). Cryptify aims to bridge this gap with a modern customtkinter GUI, making PQC accessible to non-technical users.
   - **Memory Security:** Standard software often leaves traces of keys in RAM. Cryptify aims to implement best-effort secure memory handling via a secure_erase utility that overwrites sensitive data buffers with zeros immediately after use.
3. **Scope and Limitations:**
   a. **Scope:**
      - **Hybrid Encryption:** The tool manages the full lifecycle of data protection: Keypair generation, public key export/import, file compression, PQC key exchange, and AES-GCM symmetric encryption.
      - **Large File Support:** The architecture is designed for scalability, using chunked processing (256 KB chunks) to handle files that exceed system memory capacity.
      - **Authenticated Metadata:** The tool doesn't just encrypt the file content; it also encrypts the metadata (package details) and appends it to the end of the file with a specific marker (META) for secure retrieval.

   b. **Limitations:**

- **Symmetric Key Strength:** While using PQC for key exchange, the symmetric encryption relies on AES-GCM, which, while highly secure, requires careful management of the 12-byte nonces generated for each chunk.
- **Local Storage:** The KeyManager stores keys in a local directory (~/.mlkem_keys), meaning security is still dependent on the physical or OS-level security of the host machine.

4. **Innovation Component:**

The primary innovation of Cryptify lies in its Hybrid Post-Quantum Pipeline:

- **PQC-Symmetric** Synergy: It combines the high security of ML-KEM (Kyber768) for the "handshake" or key exchange with the high performance of AES-GCM for the bulk data. This provides a quantum-resistant layer without the massive performance overhead of using PQC for the entire file.
- **Atomic Security** Operations: The tool uses HKDF (HMAC-based Extract-and-Expand Key Derivation Function) to derive the final AES keys from the PQC shared secret, adding an extra layer of mathematical separation between the exchange and the encryption.
- **Integrated Efficiency:** Unlike most security tools that require a separate step for compression, Cryptify builds the zstd compression engine directly into the encryption stream. It uses temporary file descriptors and copy_stream methods to ensure that data is compressed and encrypted in a single, efficient workflow.
- **Multi-Recipient Design:** The innovation extends to the KeyManager, which allows users to manage a "contact list" of public keys, enabling secure file sharing with others as easily as "Self-Encryption".

1. **Objectives and Goals:**
   The overarching mission of Cryptify is to transition data security from "classical" standards to "quantum-resistant" standards while maintaining high performance.
   - **Implement Post-Quantum Security (PQC):** The primary technical goal is to utilize ML-KEM (Kyber768) for key encapsulation. This ensures that even if an attacker captures encrypted traffic today, they cannot decrypt it in the future using a quantum computer.
   - **Performance Through Hybrid Cryptography:** Cryptify aims to solve the "slowness" often associated with advanced security. By using PQC only for the key exchange and AES-GCM (Advanced Encryption Standard in Galois/Counter Mode) for the actual file data, the tool achieves near-native file transfer speeds.
   - **Optimized Resource Management**: A major goal is to handle "Big Data" on standard consumer hardware. The project implements a chunked streaming architecture (256 KB processing chunks and 4 MB write buffers) to ensure that multi-gigabyte files do not exceed the system's available RAM.
   - **Integrated Storage Efficiency:** The project seeks to make encrypted files easier to manage by integrating Zstandard (zstd) compression. This goal ensures that the security overhead (encryption metadata) is often offset by the reduction in the original file size.
   - **Ephemeral Memory Security:** To mitigate the risk of "Cold Boot" attacks or memory forensics, a core objective is to ensure sensitive data (like shared secrets) exists in RAM for the shortest time possible. The tool uses a custom secure_erase function to overwrite these memory locations with zeros immediately after use.
2. **Outcomes and Deliverables:**
   The project is designed to produce a suite of functional modules and end-user products that constitute a complete security ecosystem.

   a. **Technical Deliverables (Code & Logic):**
      - **encryption.py & decryption.py:** A robust engine that manages the complex logic of PQC encapsulation, AES-GCM chunking, and metadata construction.
      - **key_manager.py:** A centralized management system that provides atomic writes for key storage and handles Argon2id or PBKDF2 password-based key derivation.
      - **CompressorDecompressor.py:** A standalone utility module leveraging the zstd library for streaming compression and decompression.

- **Authenticated File Format:** A custom encrypted file structure (.enc) that includes a 12-byte nonce prefix, chunk-size indicators, the ciphertext, and a protected JSON metadata block appended at the end with a META marker.

b. Product Outcomes (User Experience):

- **A Standalone GUI Application:** A fully functional desktop application built with customtkinter, featuring a sidebar-driven interface for generating keypairs, importing contacts, and performing batch file operations.
- **Portable Executable:** The final outcome is a packaged Linux binary (via PyInstaller) that allows users to run Cryptify on various distributions (like Kali or Ubuntu) without manually installing Python dependencies like liboqs.
- **Public Key Infrastructure (PKI) Simulation:** A workflow where users can "Export" a public key as a JSON file to give to others, and "Import" keys from colleagues to enable secure, multi-party communication.

c. Security Outcomes:

- **Confidentiality & Integrity**: Files are not only unreadable to unauthorized parties but also protected against tampering. The use of AES-GCM means that if a single byte of the file is altered, the decryption process will fail, alerting the user to a potential attack.
- **Forward Secrecy (Quantum-Scale):** Even if the long-term identity keys are eventually compromised by a quantum computer, previously encrypted files remain protected due to the ML-KEM encapsulation layer.

1. **Methods and Technology Used:**
   The project employs a sophisticated hybrid cryptographic pipeline designed for high performance and long-term security against quantum threats.
   - **Hybrid Post-Quantum Cryptography (PQC)**: Cryptify uses a hybrid model where ML-KEM (Kyber768), a NIST-standardized algorithm, handles key encapsulation to secure the exchange of secret data. This is paired with AES-GCM for the actual data encryption, providing a balance of quantum-resistant security and high-speed symmetric performance.
   - **Authenticated Encryption with Associated Data (AEAD):** The tool utilizes AES-GCM (Galois/Counter Mode), which ensures both the confidentiality and integrity of the data. If the encrypted file is tampered with, the decryption process will fail during tag verification.
   - **Streaming and Chunked Processing:** To handle large-scale files without exhausting system memory, the tool implements a chunked I/O architecture. Data is processed in 256 KB chunks and buffered into 4 MB blocks for writing to disk, ensuring high efficiency during disk operations.
   - **Streaming Compression:** The Zstandard (zstd) algorithm is integrated into the pipeline to compress files before encryption begins. It uses copy_stream methods to efficiently pass data from the input file to a temporary compressed file, reducing the overall storage footprint of the encrypted output.
   - **Key Derivation Protocols (KDF):**
     - **HKDF (HMAC-based Key Derivation Function):** Used with SHA256 to derive the final 32-byte AES keys from the shared secret generated by the ML-KEM exchange.
     - **Argon2id & PBKDF2**: The KeyManager uses these algorithms (prioritizing Argon2id if available) to derive encryption keys from user passwords, protecting the stored private keys against brute-force attacks.
   - **Secure Memory Erasure**: A critical security feature is the secure_erase function, which proactively overwrites sensitive data buffers (like shared secrets and keys) in RAM with zeros immediately after their use to prevent memory forensics.
   - **Atomic File Operations:** The project ensures data integrity during key storage by using **atomic writes**, where data is first written to a temporary file (.tmp) and then renamed to the final destination to prevent corruption during power loss or crashes.

2. **Tools and Software Used:**

Cryptify is built using a modern Python-based stack, leveraging several specialized libraries for security and interface design.

- **Python 3:** The core programming language used to orchestrate the entire application logic.
- **liboqs (oqs-python):** The primary library for Post-Quantum Cryptography, providing the implementation for the ML-KEM/Kyber algorithms.
- **Cryptography (hazmat):** A standard Python library used for low-level cryptographic primitives, including AESGCM, HKDF, and PBKDF2HMAC.
- **CustomTkinter:** A modern UI library based on Tkinter that provides the high-definition, dark-themed Graphical User Interface (GUI) for the tool.
- **Zstandard (zstd):** A high-performance compression library used for the integrated data reduction step.
- **Argon2-cffi:** Used for Argon2id password hashing, providing state-of-the-art resistance to GPU-based cracking.
- **AppDirs:** A utility library used to handle cross-platform directory paths for storing cache and configuration data securely in the user's home directory.
- **Secrets & Base64:** Standard Python modules used for generating cryptographically secure random tokens (salts/nonces) and encoding binary data for JSON storage.
- **Pillow (PIL):** Used for loading and rendering the Cryptify logo within the application sidebar.

3. **Core Architectural Layers:**

The architecture of Cryptify is designed as a modular, high-performance pipeline that integrates post-quantum security with efficient data handling. It follows a layered approach, separating the user interface, cryptographic logic, and storage management to ensure stability and security.

a. **Presentation Layer (User Interface):**
   - **Framework:** Built using customtkinter, providing a modern, asynchronous graphical interface.
   - **Concurrency:** The UI runs on the main thread, while all heavy operations (encryption, decryption, and compression) are dispatched to background threads using Python's threading module. This prevents the application from freezing during large file operations.
   - **Console Redirection:** A custom ConsoleRedirector captures standard output and errors, displaying real-time logs directly within the GUI for user feedback.

b. **Processing & Logic Layer (The Pipeline):**
   This layer acts as the engine, coordinating how data moves from a raw state to a secured state.

- **Compression Engine:** Before encryption, files are passed through the CompressorDecompressor module. It uses Zstandard (zstd) with a streaming copy_stream approach to minimize memory overhead while maximizing compression speed.
- **Cryptographic Engine (MLKEMCrypto):** This is the heart of the project. It orchestrates the Hybrid Encryption Scheme:
  - **Key Encapsulation:** It uses the oqs library to perform an ML-KEM (Kyber768) handshake, generating a shared secret.
  - **Key Derivation:** The shared secret is passed through an HKDF (SHA256) to derive a high-entropy 32-byte AES key.
  - **Data Encryption:** The compressed data is encrypted using AES-GCM (256-bit) in a chunked manner.

c. Data & Storage Layer:

- **Chunked I/O Architecture:** To support files of any size, the system reads and encrypts data in 256 KB chunks. These chunks are buffered into 4 MB blocks before being written to disk, which optimizes disk I/O performance.
- **Secure Metadata Management:** Cryptify uses a trailing metadata architecture.
  - The encrypted file starts with a 12-byte nonce prefix.
  - The main ciphertext follows.
  - A JSON metadata package (containing the KEM ciphertext, salt, and algorithm info) is encrypted separately and appended to the end of the file, marked by a META tag and an 8-byte pointer.
- **Key Management:** The KeyManager handles the local storage of keypairs in ~/.mlkem_keys. It uses atomic write operations (writing to a .tmp file before renaming) to prevent data loss or corruption.

4. Security & Memory Architecture:

- **Zero-Trust Memory Policy:** The architecture includes a memory-cleaning sub-module. The secure_erase function is explicitly called to overwrite sensitive bytearray objects (like shared secrets and private keys) with zeros as soon as they are no longer needed.
- **Authentication & Integrity**: By using AES-GCM, the architecture provides a built-in "tag" for every chunk. During decryption, the system verifies these tags; if any part of the file was tampered with, the architecture triggers a ValueError and halts the process to protect the user.

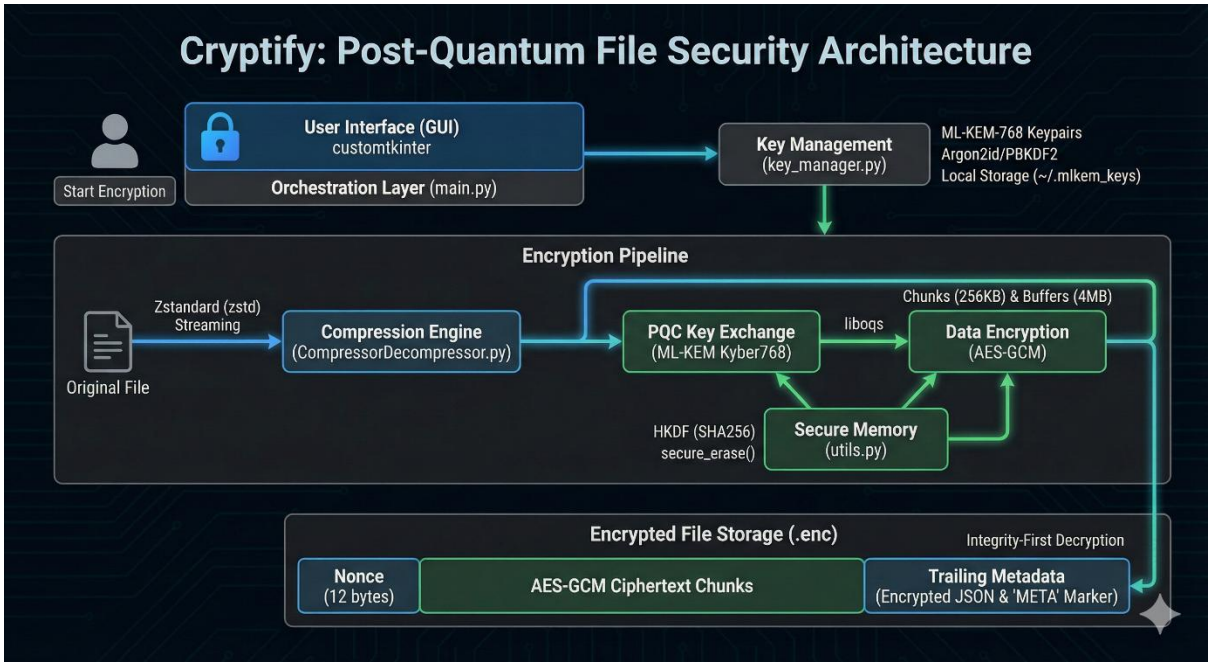
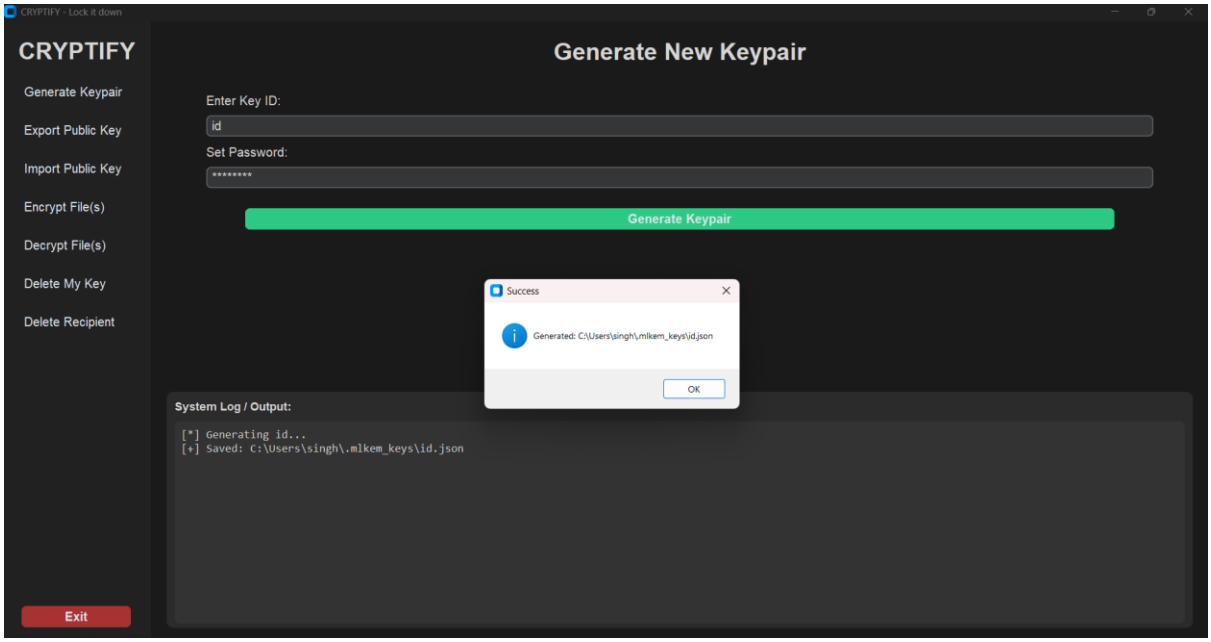


Image: Architecture of Cryptify

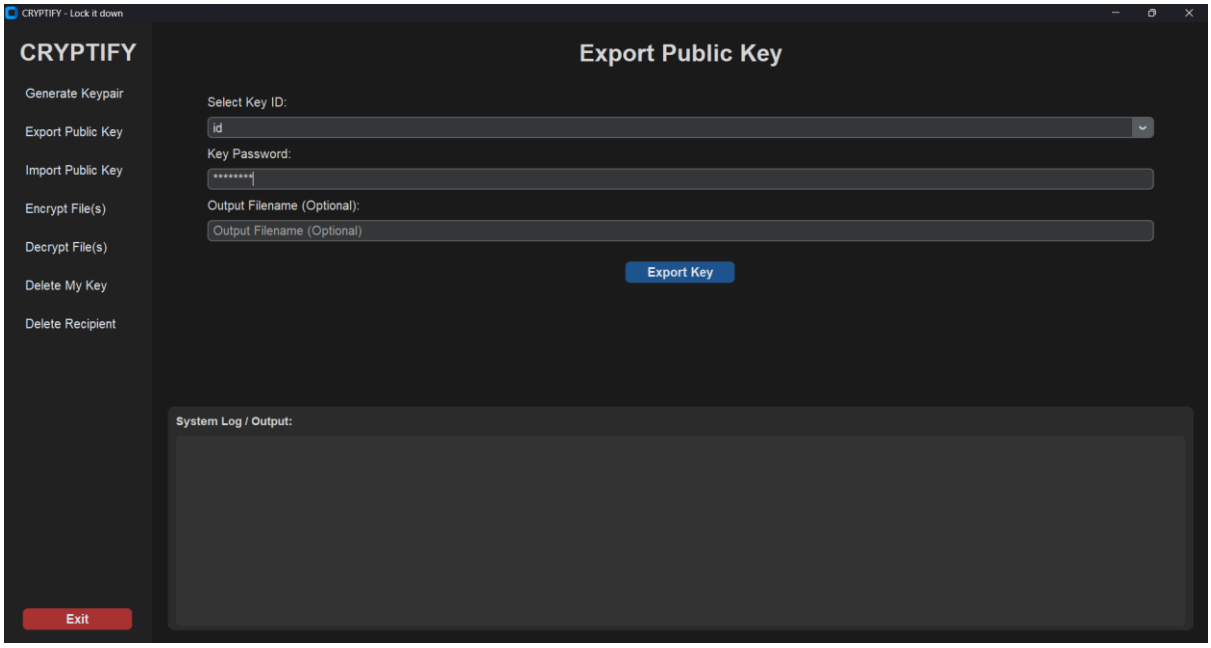
5. Project Working Screenshots & Explanation:

a. The Home Screen & Sidebar:



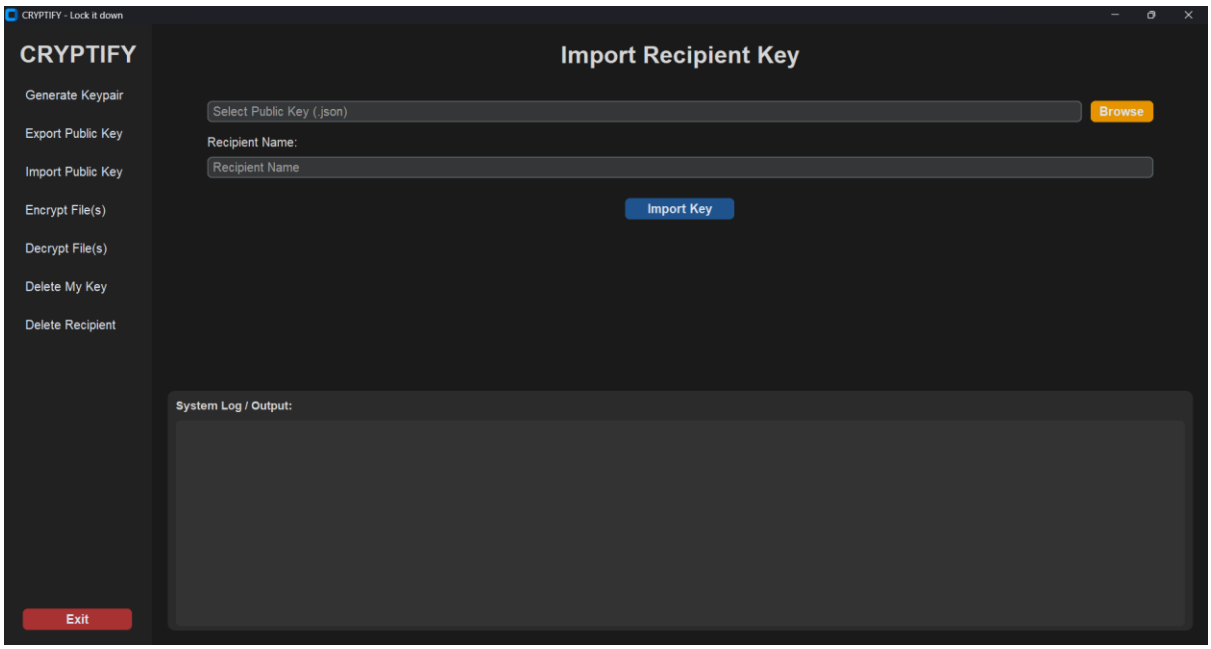
Explanation: Upon launching the application, the user is presented with a clean, modern interface built with customtkinter. The Sidebar on the left acts as the primary navigation hub, allowing users to easily switch between different modules of the application. The initial Home Screen provides a simple landing page, guiding the user to get started with securing their files.

b. Key Generation & Management:



Explanation: This screen demonstrates the Key Manager module (key_manager.py). Users can generate their own Post-Quantum (ML-KEM-768) keypairs, which are securely protected by a password using Argon2id hashing. The generated public keys are listed, making them easy to export and share with other users to establish secure communication channels.

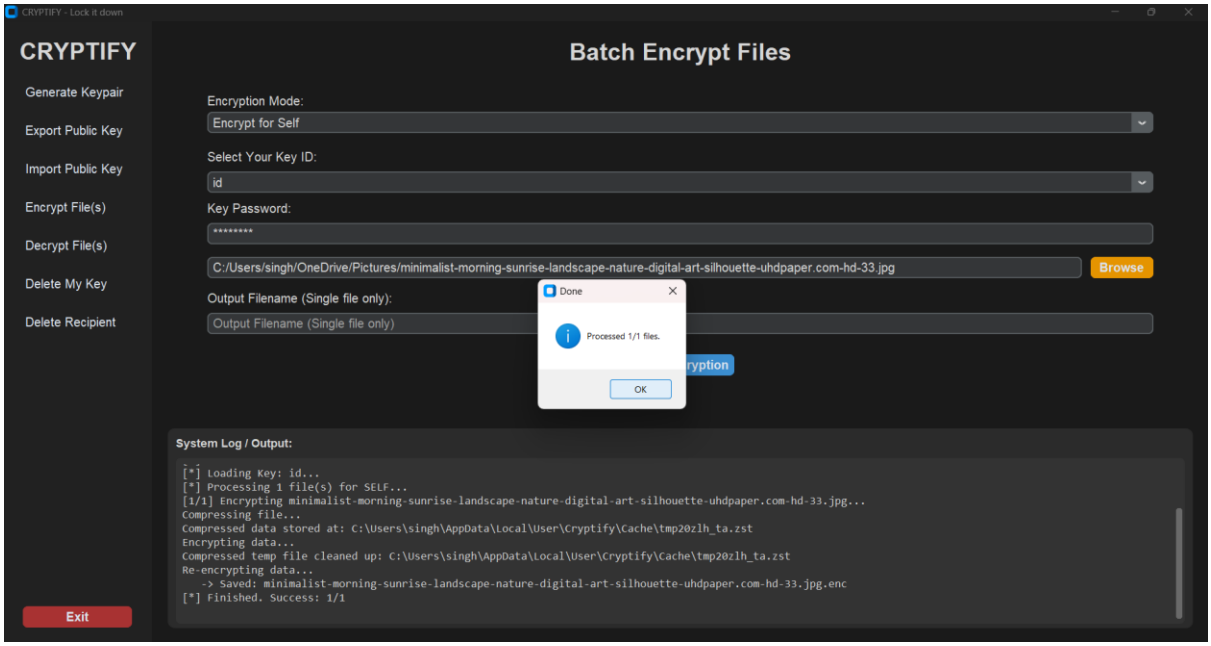
c. File Encryption Process:



Explanation: This screenshot captures the core encryption workflow. The user selects a target file and the intended recipient's public key. Upon initiating the

process, the application's background threads perform a series of actions: Zstandard compression to reduce file size, followed by a Post-Quantum key exchange to derive a secure session key. Finally, the data is encrypted using AES-GCM in chunks, with live progress reported in the integrated console window.

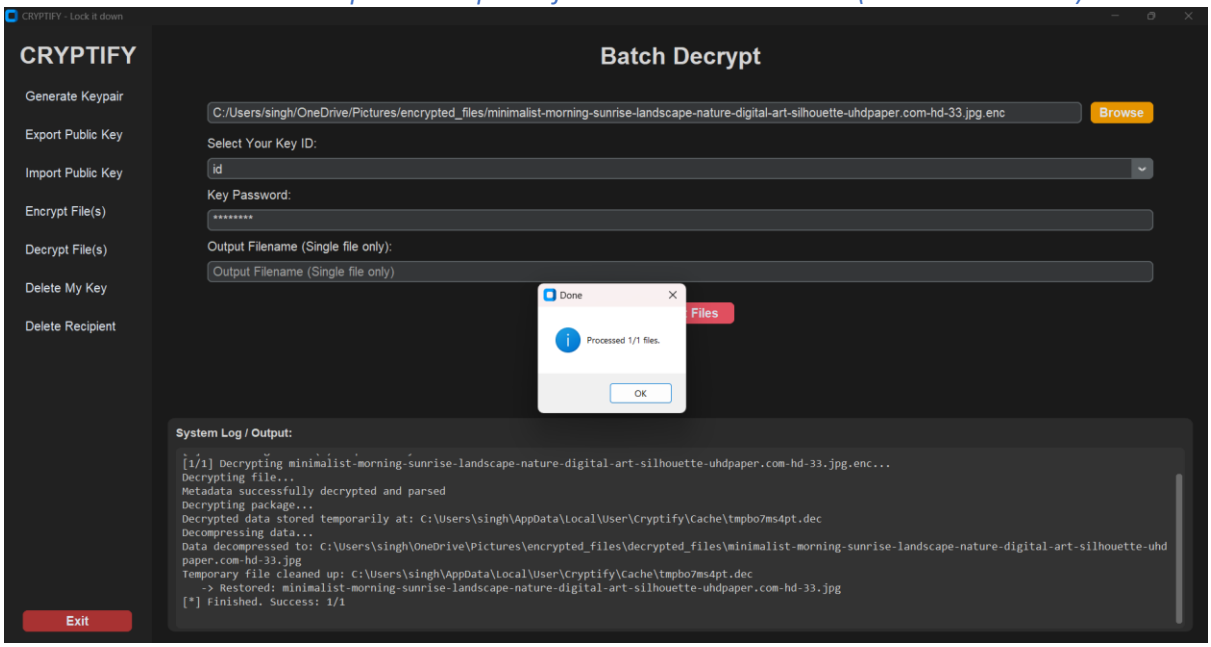
d. File Decryption & Verification;



Explanation: This screenshot captures the core encryption workflow. The user selects a target file and the intended recipient's public key. Upon initiating the process, the application's background threads perform a series of actions: Zstandard compression to reduce file size, followed by a Post-Quantum key exchange to derive a secure session key. Finally, the data is encrypted using AES-GCM in chunks, with live progress reported in the integrated console window.

e. File Decryption & Verification:

Explanation: The decryption screen demonstrates the reverse process. The user selects an encrypted (.enc) file and provides the password for their private key. The application reads the trailing metadata, performs the ML-KEM decapsulation to recover the shared secret, and then decrypts the AES-GCM chunks. The successful completion of this process also confirms the integrity of the file, as any tampering would have caused the AES-GCM verification to fail.



6. Project GitHub Link:

https://github.com/VeerSingh0001/Cryptify

This repository contains the full source code for the Cryptify project, including:

- The main GUI application (main.py).
- The core cryptographic modules (encryption.py, decryption.py, key_manager.py, utils.py).
- The compression utility (CompressorDecompressor.py).
- A requirements.txt file for easy installation of dependencies.
- Assets such as the application logo.

1. **Dalveer Singh:**

   Technical New Learnings:

   - **Post-Quantum Cryptography (PQC**): Moved from theoretical knowledge of RSA/ECC to practical implementation of lattice-based cryptography using ML-KEM-768 (Kyber). Gained deep experience working with the liboqs Python bindings and understanding key encapsulation workflows.
   - **Secure Memory Handling**: Discovered the limitations of Python's garbage collection for security tools. Learned techniques for mitigating data remanence attacks by implementing proactive memory wiping (secure_erase) using ctypes.
   - **Streaming Data Algorithms:** Learned to abandon "load-it-all-into-RAM" approaches for handling files. Mastered the implementation of chunked processing (reading in 256 KB blocks) to ensure the application remains memory-efficient regardless of input file size.

   Overall Experience:

   "Building Cryptify was the most challenging academic project I've undertaken because it bridged the vast gap between cryptographic theory and real-world software engineering. The most frustrating, yet valuable, experience was debugging AES-GCM decryption failures. Unlike standard bugs that give stack traces, a crypto bug just results in garbage data or a 'MacCheckFailed' error, forcing you to re-evaluate every byte of your pipeline. Successfully decrypting that first hybrid-encrypted file was an incredibly satisfying moment."

2. **Akashdeep Singh:**
   Technical New Learnings:
   - **Python Concurrency & Threading:** The biggest technical hurdle was preventing the GUI from freezing during large file encryption operations. I learned how to effectively use Python's threading module to offload heavy compute tasks to background threads while safely updating the customtkinter UI on the main thread.
   - **Modern GUI Development:** Transitioned from basic Tkinter to customtkinter. Learned how to design modern, dark-mode interfaces, manage complex layouts with frames, and create a responsive user experience.
   - **Cross-Platform Path Management**: Gained experience using libraries like appdirs and os.path to ensure secure key storage locations work reliably across different Linux distributions.

Overall Experience:

"My experience focused on making complex security accessible. While the backend team was fighting with quantum algorithms, my challenge was translating that into a simple 'Lock it down' button for the user. Balancing a responsive, modern-looking GUI with the heavy background processing required for encryption was difficult. It taught me that a powerful tool is useless if the user interface is confusing or unresponsive. I'm proud of the professional polish we achieved with the final application."

1. **Objectives and Achievements;**
   The Cryptify project was initiated to address the growing security gap between current classical encryption standards and the emerging threat of quantum computing. Below is a summary of the project's core objectives and the significant milestones achieved during development:
   - **Quantum-Resistant Foundation:** Objective: Implement a "future-proof" security layer.
   - **High-Performance Hybrid Pipeline:** Balance advanced security with practical processing speeds.
   - **Integrated Resource Optimization:** Minimize the storage footprint and manage system memory effectively.
   - **Advanced Key Management:** Create a secure and user-friendly system for managing cryptographic identities.
   - **Modern Accessible Interface:** Abstract complex cryptography behind an intuitive GUI.
2. **Future Scope of the Project:**
   While Cryptify provides a robust post-quantum security framework, the project is designed for continuous evolution. Future development will focus on the following areas:
   - **Hardware Security Integration**: We plan to extend the KeyManager to support hardware-based storage, such as TPM (Trusted Platform Module) or USB Security Keys, to ensure that private keys never exist in a vulnerable state on the hard drive.
   - **Mobile Ecosystem**: Development of a lightweight mobile application (Android/iOS) to allow for secure file decryption and viewing on the go, utilizing the same ML-KEM standards.
   - **Cloud Integration:** Building secure "Sync & Encrypt" plugins for major cloud providers (Google Drive, Dropbox), ensuring data is encrypted locally with PQC before ever being uploaded.
   - **Network Protocol (Cryptify-Transfer):** Expanding from file encryption to a secure peer-to-peer transfer protocol that uses the existing PQC handshake to secure data in transit across public networks.