Password Manager Report

Mohammad Wehbe
Computer Science Department
American University of Beirut
Beirut, Lebanon
mmw08@mail.aub.edu

***Abstract*—Password managers play an important role in digital security by allowing users to securely handle their passwords for multiple accounts and platforms. This report presents the design, implementation, and security analysis of our command-line password manager that is built in Go. Our approach uses Scrypt which is key derivation function to derive a key from user password to encrypt the data using AES-256-GCM in order to protect the user credentials. Also it implements multiple memory protection techniques such as memory locking and multi-pass data wiping. Moreover, we provide user with great features such as password strength validation, duplicate password detection, and optional cloud backup. Our implementation demonstrates how implementing modern cryptography can protect the user from multiple threats and attacks, but it shows that no system has perfect security.**

***Index Terms*—Scrypt, AES-GSM, Memory protection, Key Derivation function, Cryptography, Password Manager.**

I. Introduction

Nowadays, individuals use multiple range of platforms that facilitate their life like social media, banking accounts, delivery application, taxi applications, etc. Each of these requires its own credentials, including a password and username. However, it is difficult for an individual to remember multiple passwords, so people reuse the same password across multiple sites, choose weak passwords, or store them insecurely in plaintext files. Given the ongoing risks of credential stuffing, it is essential to have a strong and secure method for password management. Password managers are essential security tools that help users generate, store, and manage their credentials for various services [3]. However, many popular password managers that are commonly used are not open source, which creates a critical lack of transparency, making it impossible to independently confirm the security approach they use to protect user credentials.

As a result, in this project, we create an open source, command-line (CLI) password manager in Go, emphasizing transparency and the use of strong cryptographic techniques. To protect users credential, the resulting password manager tool must be resilient against multiple attacks such as:

- Brute force attacks: Where an attacker obtains the encrypted database file.
- Data manipulation: Where an attacker attempts to modify the data.
- Key exposure: Where an attacker compromises system memory to extract the encryption key.

To surpass these security issues and established attack vectors, we designed our approach around a multi-layered security model that ensures the confidentiality, integrity, and authenticity of user credential. This model use strong key derivation library which generates a robust key from the user's master password. Furthermore, we employ authenticated encryption via AES-256-GCM to ensure data confidentiality and integrity. Finally, memory protection is achieved by implementing OS specific memory locking to remove sensitive information from memory.

II. Related Work

A. Relevant Cryptographic Protocols

Most modern password managers rely on a Key derived from the user's password and use AES encryption to protect the user credential database. In most approaches, the user password is the access point to the database, and without it, the encrypted data is useless. From this given password, the system used what is called Key Derivation Functions (KDFs) to derive a key. The use of KDFs is critical for resisting offline dictionary and brute-force attacks. The most common standard KDFs that are being used are the one that uses memory hard algorithm.

- Argon2id: This hybrid function is resistant against both GPU cracking attacks and side channel attacks which achieves a balance secure for modern application [5]. To clarify, It uses memory hard algorithm which forces attackers to allocate significant amounts of expensive RAM for each parallel guessing attempt, creating a very high computational cost.
- Scrypt: Similar to Argon2id, Its is built to resist brute force attacks by forcing attackers to use large, adjustable amounts of high-speed memory (RAM) [1]. This time memory trade off makes it much more expensive and difficult to parallelize attacks using specialized hardware like GPUs.
- PBKDF2 (Password-Based Key Derivation Function 2): This algorithm uses HMAC with SHA-256 as a pseudo-random function and repeatedly applies it to the password and salt thousands of times [6]. PBKDF2 is not memory-hard, meaning it requires very little memory to compute. This makes it more vulnerable to brute force attack that uses specialized hardware such as GPUs.

KDF processes the master key with a randomly unique cryptographic salt. This ensures that two identical master passwords held by different users result in two distinct derived keys which can prevent rainbow table attacks.

AES-GCM (Galois/Counter Mode) is authenticated symmetric encryption. It combines AES encryption in Counter mode with the GHASH function to provide both confidentiality and authenticity through an authentication tag [8]. In the password manager application, it encrypt the database entry with the key derived from KDFs using AES, then the entry is automatically authenticated with GHASH function. Thus, if the data is modified, the user can know since authentication tag verification will fail, so the system will not decrypt.

There are numerous password managers available today, each with its own strengths and weaknesses. Several research papers have analyzed their vulnerabilities by performing audits and simulated attacks. For example, in [4], the authors conducted a systematic security assessment of five popular web-based password managers and found that four were vulnerable to attacks that could allow attackers to steal user credentials. The study highlighted that, despite the theoretical security benefits of password managers, real world implementations especially those synchronized with the browser introduce significant challenges.

Among open-source password managers, two examples stand out. The first one is KeePass which is local password manager using AES-256 encryption with Argon2, ideal for users who prefer offline, file based storage [10]. The second one is Pass which is a simple UNIX-based password manager that encrypts each password using OpenPGP and works easily with Git for version control [9].

Each of these systems employs unique cryptographic primitives and architectures to secure user data. However, our implementation focuses on low level memory protection using Go's secure string and OS level calls such as memory lock. Moreover, our approach emphasizes security transparency and resilience against memory-swapping attacks in a terminal based environment.

III. System Design

A. Architecture Overview

In this project, the password manager is built in one code file, but the code itself can be separated into four main layers. At the top, we have user interface layer which handles all interaction with the user like displaying menus, accepting input, managing the screen, and providing feedback. This layer presents the request of the user. Then there is application layer that handles managing the database, tracking session state and timeouts, and running security audits. Then there is security Layer which handles all cryptographic operations such as encryption, decryption, password validation and key derivation. Then, there is a layer that uses function to clean memory in order to prevent swapping.

1) Password entry: In our design, we store each password as a PasswordEntry, which contains four fields: the site name (like "Facebook" or "gmail.com"), the username or email address, the password itself, and optional notes for additional context. Each entry includes a Wipe method that securely erases all these fields from memory using multiple overwrite passes when the entry is no longer needed.

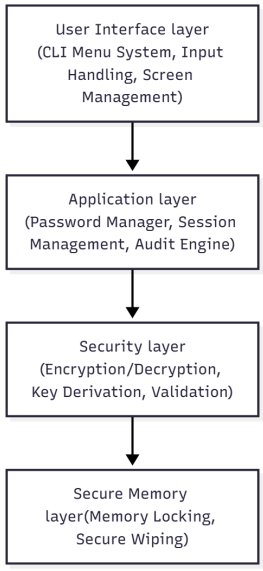

Fig. 1. System Architecture Layers of the Command-Line Password Manager

2) Database structure and protection: The database contents (which hold all password entries) are first converted into a JSON format. This data is then encrypted using AES-256 in GCM (Galois/Counter Mode), a robust and authenticated symmetric encryption standard. The encryption key that is used to encrypt the data is not the master password itself. However, we use Scrypt to generate a strong 256-bit key from the Master Password. Also, a unique 16-byte random Salt is generated and used in the encryption to prevent rainbow table attacks. The final result is a ciphertext for the database that are stored on a file called "passwords.db".

3) User Interface: The user interacts with the application primarily via CLI through text input and menu selections. First, the user should start executing the file. Then, the application immediately prompts the user for the Master Password. When the system is verify Master Password and database, the system will present a main menu listing the available actions (like: Add Entry, Get Password, Update Entry, Delete Entry, Configure Sync, Exit) to the user. As a result, the user can select the option that he/she wants to use by typing it correspond number, and the system will execute it.

B. Cryptographic Primitives & Design Decision

1) Key Derivation: The journey from the user's password to the encryption key follows a carefully designed path (see Fig. 2). To begin with, the user enters their master password during registration, which must be at least 12 characters long and include at least one uppercase letter, one lowercase letter, one digit, and one special character. Next, a 16-byte random salt is generated to ensure uniqueness which can also prevent

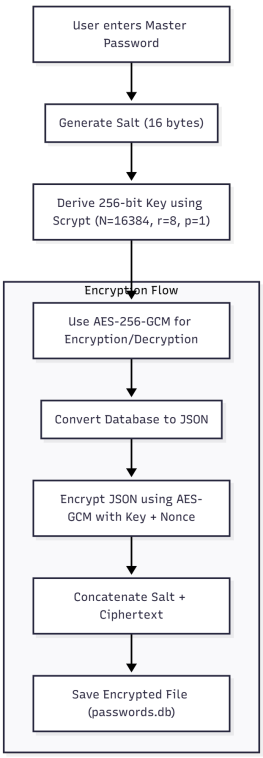

Fig. 2. Password Encryption Workflow

rainbow table attacks, and it is combined with the master password to derive a 256-bit key using the Scrypt library in Go.

The Scrypt function is configured with the following parameters: N = 16384, r = 8, and p = 1. These values are chosen to enhance security by increasing memory hardness [1]. This lead to make attack more difficult and computationally expensive. To clarify, when N is 16384, each key derivation requires approximately 16 MB of memory. The parameter r = 8 determines the block size and influences memory access patterns, while p = 1 indicates that no parallelization is used, which preserves memory hardness.

These chosen parameter values make a balance between security and usability. Higher values would provide stronger protection but at the cost of slower performance for legitimate users, while lower values would reduce security. This configuration computes in less than one second which is considered fast enough for a good user experience, but it is slow to make brute-force attacks prohibitively expensive. As a result, it would take decades to successfully guess the password manager, so the stronger the password is chosen by user, the better security he/she will have.

2) Encryption Scheme: After the key is derived from master password using Scrypt, the encryption process of the database and entries starts to protect the user data from being leaked. It starts with the conversion the plaintext database to JSON format. Then we take the data, and encrypt it using the 256-bit key we generated. To make the encryption more secure, we use GCM mode (Galois/Counter Mode), which checks that it hasn't been changed or tampered with. To clarify, the GCM produces ciphertext along with a 128-bit authentication tag, so if anyone modifies even a single bit of the ciphertext, decryption will fail because the tag will not match. We also create a random 12-byte nonce that will only be used once so if the same data is encrypted again, it will produce completely different ciphertext each time.

After that we save the ciphertext in the file. Note that, the first 16 bytes of the file are the salt, the next 12 bytes represent nonce, and the rest are the ciphertext with its authentication tag. This format ensures that the file contains all necessary to decrypt the database given the master password.

This design ensures that cryptographic properties are satisfied. To clarify, confidentiality is ensured by AES-256 since recovering the plaintext without the key is computationally infeasible with current technology. Integrity ensured by GCM authentication tag which detect any modification to the ciphertext. Finally, authentication is satisfied by the key derivation since only someone who knows the master password can derive the correct encryption key.

3) Session Management: What happens if user unlock his/her password manager and then step away from his/her computer? Anyone who are near away could access his/her passwords. Our solution implements automatic locking based on inactivity. To clarify, when user successfully sign in, a session begins with a two-minute timeout. Whenever user interacts with the application like typing a menu choice or enter data, the timer resets. However, if two minutes pass with no activity, the system automatically locks itself. The locking process starts with displaying timeout message, securely wiping all passwords from memory, and forcing a garbage collection to free memory pages. Then user returns to the startup screen where he/she must re-authenticate.

This design prevents unauthorized access while the user is away from his/her laptop. Note that we choose time out to be 2 minutes and not less since user may take too much time looking for the list of his password or reading a long password.

4) Memory Protection Strategy: Operating systems can move data from RAM to disk to save space. This is not secure approach for a password manager application because sensitive information (passwords) could end up stored on disk. This can allow attacker to extract passwords if he/she can access disk. To prevent this, we use memory locking which prevents the storing of the data in the disk via keeping critical data only in physical RAM. In our implementation, since we are using Window 11 to test the code, memory locking is done with virtualLock. To clarify, memory is locked automatically as soon as it's used for sensitive data which ensures that data never leaves secure memory.

When user are done using the sensitive data, we automatically erase it from memory. We use a multi-pass wiping method in which data is overwritten several times with random values to ensure nothing can be recovered. We use this approach as an extra safety measure since the performance cost is minimal and it ensures complete data removal.

5) Password Security Features:

- Strength Validation: Weak passwords is the main issue in our application, so we enforce user to choose a password with minimum requirements. Every master password must be at least 12 characters long and include uppercase letters, lowercase letters, digits, and special characters. Therefore, password must satisfy all the above requirement to allow user to sign up. Also, we calculate a strength of the password based on the length, so password are rated as WEAK, MEDIUM, STRONG, or VERY STRONG which provides user with clear feedback about the security. As a result, a password matching the minimum requirements might be MEDIUM strength, while a 20-character password with good diversity rates VERY STRONG. When users add new password entry, we show them this assessment immediately. If they're about to save a weak password, we warn them explicitly and require explicit confirmation. This educates users about password security while giving them the freedom to make informed decisions.
- Password Generation: In our design, user has a cryptographic password generator feature, which allows him/her to create truly random passwords using Go's crypto/rand package. Users can choose the length of the password or keep the default value which is 16, and the function builds the random values including uppercase letters, lowercase letters, digits, and special characters. By using cryptographic randomness and careful selection, we ensure that a 16-character password can resist brute-force attacks for decades.
- Password Auditing: Moreover, in our design, the user has the security audit feature which provides two automated analyses: the first checks for duplicate passwords, and the second checks for weak passwords . To clarify, the system checks over all password entries and identifies when user used the same password across multiple sites. Then the system will show the user a list of cites/services share same passwords. Also, it applies strength validation to every entry to identify password that doesn't meet minimum requirement. Then, we show the site and username, so user can know exactly which password website to update.

6) Cloud Backup: Many users may lost their file that is store in their device due to damage on device. To handle this issue, we give the user an option to synchronize a backup file and store it on his/her cloud. To clarify, When you enable cloud sync, you point to a folder (like your OneDrive folder). When you lock or exit the application, it copies the encrypted database file to that folder. Therefore, if the file is lost on the user device, the user can download it from the cloud and use it.

What does the cloud service see? The cloud service can only see an encrypted file. This means that even if the cloud service is breached, attackers only get encrypted files that they can't decrypt without user master password.

IV. Implementation

A. Language used

The project is tested on windows 11 Laptop, and it is built in Go using "1.25.1 windows/amd64" version because it offers strong security features via cryptographic libraries. Moreover, Go's standard library provides reliable implementations for AES, GCM, Scrypt, and secure random generation, all maintained by experts.

Go's concurrency features made session management easy by allowing background processes to run safely alongside user interactions. It also ensures memory safety and lets developers manage sensitive data securely [2]. In this design, we only use three external libraries which are:

- golang.org/x/crypto/scrypt: for secure key derivation [2].
- golang.org/x/term: for safe password input in the terminal [2].
- github.com/atotto/clipboard: for copying passwords securely to the clipboard [7].

B. Core Cryptographic Implementation

1) Key Derivation: For key derivation, the function takes a master password from user and generates a unique random salt, and then it calls Scrypt with carefully chosen parameters to derive 256-bit keys, all of which are explained in Section III-B1.

2) Encryption and Decryption: The encryption process, already described in the previous section, converts the plaintext database into ciphertext and store it in the database file. In this section, we focus on the decryption stage, which reverses the decryption operation to retrieve the original data. To begin with, the system first extracts the 16 bytes of salt and 12 byte nonce from the beginning of the encrypted data. Then the system will derive 256-bit key from the user's master password, it reconstructs the AES cipher in GCM mode. Then the system will start with verification of authentication tag to ensure that the file is not modified. If the verification failed, the decryption will stop, but if it passed, then the ciphertext will be decrypted, so user can access the password list. Note, authentication prevents attackers from tampering with the encrypted database which maintains both confidentiality and integrity.

C. Secure Memory Management

1) Secure String and Memory locking: In our approach, all sensitive data is stored in secure String data type. When it is created, it locks the memory, so data can't be swapped to disk. It automatically wipes data using multiple overwrite passes before unlocking the memory.

2) Garbage Collection: Go's memory management can move or free memory unpredictably. To handle this, memory is locked right after allocation, and sensitive data is wiped before freeing. The program also uses runtime.GC() to help clear memory sooner which ensures sensitive memory is always cleaned up, even if an error occurs.

D. User Interface and User Experience

1) Secure Password Input: When user enters a password, if we keep the input as it is, nearby individuals can know his/her password by locking to screen. To handle this issue, passwords are entered in raw terminal mode to prevent them from being exposed on the screen, and each typed character is represented by an asterisk. This can allow the user to visualize the enter without exposing the input. Moreover, if the user types a character and cancel it, the system can handle this approach, and it always wipe all temporary data from memory.

2) Clipboard Security: When the user decides to retrieve the password for a given website, he/she has a clipboard feature which allows users to copy passwords for quick pasting. It only allows a copy of password in clipboard for a limited 10-second window. After the time expires, the clipboard will automatically clear the password from it list to reduce exposure risk. .

E. Constant-Time Operations

When user enter his/her password, there is a comparison occur. This introduces timing attacks which are a vulnerability where attackers can learn secrets by measuring how long operations take. To handle this issue, we perform constant time comparison and we use "crypto subtle" Library that compares passwords byte by byte taking the same amount of time whether the passwords match or differ in the first character.

F. Error Handling

In every application, error must exist, so whenever an error occurs such as file access or memory locking, the system first ensures that all sensitive data is securely wiped before returning the error. This approach guarantees cleanup happens even if errors occur unexpectedly.

Moreover, the system uses generic error messages for authentication failures to prevent the attacker to know the type of error that the user has since if an attacker can distinguished between authentication failure, then we give him/her an oracle to test whether file tempering is detected. However, we still providing informative but safe messages for user facing issues to maintain usability without exposing security details

V. Security Analysis

A. Threat Model

In order to analyze the security of our approach, we should first define the possible threats that we aim to defend against. Our threat model makes several assumptions about the capabilities and goals of potential attackers. We categorize potential threats and attacks into two main groups: attacks with possible protection and attacks beyond our control.

1) Attacks with possible protection:

- Local Adversaries with Limited Access: An attacker (can be friend, family member, colleague) who can gain a temporary access to the user's computer while the password manager is locked. They might try to access the encrypted database file, but without the master password, they cannot access the password manager application or data.
- Memory Dump Attacks: This type of attack happens when an attacker gains access to a snapshot of the program's memory, using malware or debugging tools. Thus, he/she can get the master password from memory.
- Brute-Force Password Attacks: in this attack, an attacker obtains the encrypted database file and attempts to guess the master password by trying millions of possibilities.
- Weak Password Vulnerabilities: Users who undermine their own security through poor password practices.
- Data Tampering: An attacker maliciously alters the encrypted file's content.
- Key Guessing: in this attack, an attacker measures password validation time to guess the master key.

2) Attacks beyond our control:

- Root Access: If an attacker has administrative control on the system, he/she can install keyloggers that capture the master password as it's typed. Defense against this type of attacks requires operating system security controls and hardware-based protections which are beyond our application's scope.
- Physical Access with specialized tools: in this attack, an attacker use special hardware tools that can retrieve information from RAM which require physical access and expertise. This is also considered out of our control to handle it in our application.
- Quantum Computer Attacks: quantum computers could theoretically break all symmetric encryption including AES-256 using Grover's algorithm.
- Social Engineering: in this attack, an attacker trick the users to give him/her their password.

B. Security Properties and Guarantees

In our approach, we implement multiple security guarantees based on the cryptographic primitives to ensure that our application is secure against multiple attack.

- Confidentiality: In our approach, the encryption key is created from the user's master password. The password database is then protected using AES-256 encryption, which is extremely secure, and trying every possible key would take a lot of time and money. This means no one can realistically break it, so only user with master password can access the data.
- Integrity and Authentication: GCM provides authenticated encryption using a 128-bit authentication tag. If an attacker tries to modify the encrypted data, the tag verification will fail during decryption. As a result, the system will refuse to decrypt the data, preventing the attacker from learning anything about the contents or behavior of our system.
- Resistance to Brute-Force Attacks: in our approach, we use Scrypt key derivation function to derive the key from master password with N=16384, r=8, and p=1. These values are chosen to enhance security by increasing

memory hardness. This lead to make attack more difficult and computationally expensive. To clarify, each password require more than 0.2 seconds of CPU time without any penalization, so trying billions of guesses requires decades. As a result, it seems impossible to get the user master password.
- Defense Against Dictionary Attacks: In this approach, we derive the key using user password and a random salt stored with each database ensures that two users with the same master password will have completely different encrypted databases, so each user requires individualized attack effort. This prevents precomputation attacks.
- Memory Protection: our implementation protected sensitive data from being swap to disk by locking it in physical RAM. Also we perform wiping using multiple overwrite after finishing using the data. As a result if attacker tries to dump the memory, he/she cannot retrieve any data from the disk.
- Session Security: In our approach, we implement an automatic timeout mechanism that locks the password manager after two minutes of inactivity. This prevents situations where a user forgets to manually lock the application, reducing the risk of unauthorized access by someone else (for example, a family member). When the timeout occurs, the system also performs secure memory wiping, ensuring that no sensitive data remains in memory.
- Back up option for database file: In our implementation, users have the option to back up their encrypted password file to a cloud service such as OneDrive or Google Drive. This ensures that if the local file is lost, the user can easily download it from the cloud and restore their data. The backup is completely secure because the file is encrypted locally before being uploaded, and only the user with the master key can decrypt it.

C. Limitations and Vulnerabilities

Every security system has its limitation, so no security system is perfect. In this section, we will present limitations and potential vulnerabilities for our system.

- No Recovery Mechanism: In our approach, there is no way user can recover the master password, if user forget his/her master password, there is no way to recover the encrypted passwords. Thus, all his/her data are lost.
- Memory Protection Limitations: in our approach, we implement memory locking and wiping to protect attackers from implementing Memory Dump. However, CPU caches may contain copies of sensitive data, so an attacker can perform an attack to retrieve data from the cache.
- Clipboard Exposure: in our approach, when user copy passwords to the clipboard, passwords stay in clipboard for 10 seconds. This can open a window for attack. To illustrate, a malware that is designed to monitor clipboard contents could capture passwords during this window.
- Quantum computer: in our approach, we did not consider the potential threat generated by future quantum computers, which could potentially break symmetric encryption algorithms such as AES-GCM.
- Single Master Password: in our approach, the security of the entire database depends on a single master password. If this password is compromised, all stored credentials are exposed. Therefore, adding multi-factor authentication can provide more security, but it would complicate the design and deployment.

VI. Conclusion

A. Summary of Achievements

This project successfully developed a secure password manager that combines strong cryptographic protection with practical usability. We used AES-256-GCM for encryption and Scrypt key derivation. To ensure high security level for stored credentials against potential attacks, we implement memory protection, secure wiping, and session management. The command-line interface is easy to use, but a more user-friendly graphical interface (GUI) would improve accessibility. Our system provides multiple features, including password strength checking, detection of duplicate or weak passwords, and random password generation for both security and convenience. Additionally, our approach is open-source, allowing anyone to verify or audit the implementation.

B. Lessons Learned

Throughout designing and implementing this project, several important lessons were learned that extended my theoretical understanding and practical experience. The lessons can be summarized in the below list:

- Gained practical experience with fundamental cryptographic primitives. Implementing encryption, key derivation, and authentication mechanisms provided hands-on understanding of how these concepts work in real-world systems.
- Learned that there is a trade-off between security and performance. For example, implementing session timeouts enhances security but introduces latency.
- Understood the importance of clear documentation, well-defined threat models, and structured system design. Proper planning improves the overall security of the system.
- Discovered that while using cryptographic functions from well-established libraries is straightforward, implementing them correctly is challenging. Before this project, AES-GCM and similar algorithms seemed complex to apply in practice, but applying them was easy.
- Learned to implement new memory protection techniques not covered in class, such as secure memory wiping and locking.

C. Future Work

The current implementation can be considered good, but it should be enhanced by applying the below enhancement.

- Apply Post-Quantum Cryptography: In our approach, we use AES-GCM encryption which can be broken via quantum computer, so we should implement post quantum encryption in order to protect the user credentials from being exposed with future quantum computers.
- Apply Two-Factor Authentication for Master Password: Our system should require a secondary authentication factor (TOTP code, biometric) in addition to the master password. This would protect the user in case his/her master password exposed, so attackers need user's authenticator factor to access data.
- Apply GUI: Our approach is using CLI interface, but users prefer graphical interfaces. However, a new challenge will be introduced which is ensuring that GUI doesn't create new vulnerabilities.
- Apply Searching in the database: our current approach only retrieve all the password stored, but user usually search for a specific website.
- Create Mobile Application: Our current implementation work on CLI desktop, but many users create their accounts in their mobile, so they need password manager on their mobile devices.

VII. Source Code

The complete source code for this project, including implementation files, documentation, and the user manual (README), is available on GitHub at the following $link$.

References

[1] I. Percival, "Stronger Key Derivation via Sequential Memory-Hard Function," in Proceedings of the 21st USENIX Security Symposium (USENIX Security 12), Bellevue, WA, 2012, pp. 297–312.

[2] The Go Authors, "Security – The Go Programming Language," Go.dev Documentation, Available: https://go.dev/doc/security/

[3] N. Kobiessi, "Password Manager – Applied Cryptography," Applied-Cryptography.page.

[4] Z. Li, W. He, D. Akhawe, and D. Song, "The Emperor's New Password Manager: Security Analysis of Web-based Password Managers," in Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14), San Diego, CA, USA, Aug. 2014, pp. 299–314

[5] Biryukov, A., Dinu, D., Khovratovich, D. (2015). Argon2: The password hashing competition (PHC) winner. Password Hashing Competition

[6] Kaliski, B. (2000). PKCS 5: Password-Based Cryptography Specification Version 2.0. IETF RFC 2898.

[7] M. Atotto, "atotto/clipboard," GitHub repository, Available: https://github.com/atotto/clipboard

[8] McGrew, D. A., Viega, J. (2004). The Galois/Counter Mode of Operation (GCM). In Proceedings of the 2nd Workshop on Organization and Self-Organization of Electrophilic Reagents and their Application in Organic Synthesis

[9] J. Donenfeld, "Pass: The Standard Unix Password Manager"

[10] KeePass, "KeePass Password Safe,"