International Scientific Journal of Engineering and Management (ISJEM) ISSN: 2583-6129
Volume: 04 Issue: 09 | Sept – 2025 DOI: 10.55041/ISJEM05021
An International Scholarly || Multidisciplinary || Open Access || Indexing in all major Database & Metadata

SafePass: A Rust-based Password Management Wallet Solution using Intel SGX

Abhijeet Sonar
Mumbai, IN
abhijeetsonar.us@gmail.com
CTO

Omkar Wagle
San Jose, CA
ov.wagle@gmail.com Software Engineer

Aishwarya Lonarkar
Dallas, TX
aishwaryalonarkar@gmail.com
Full Stack Engineer

***Abstract*—SafePass is a password manager wallet that leverages Intel Software Guard Extensions (SGX) technology and the Rust programming language to provide secure storage and management of user passwords. The application features a secure enclave implemented in Rust, which uses a hashmap-based data storage system to support various password management operations such as username-password addition, updating existing passwords, removing particular username and password entries, clearing the entire wallet, recommending password and finding password by username. The enclave employs SGX's hardware-based memory encryption and access control features to ensure that sensitive data is protected against unauthorised access and tampering. In addition to the aforementioned security mechanisms, the password manager application also incorporates two-factor authentication (2FA) within the secure enclave implemented in Rust. This design choice enhances the application's overall security posture by embedding 2FA within the enclave itself, effectively isolating the authentication mechanism from potential attackers and mitigating risks associated with credential theft or abuse. Overall, this password manager application showcases the capabilities of SGX and Rust in delivering secure and reliable password management solutions. By incorporat- ing 2FA and hardware-based security mechanisms, the application effectively protects sensitive data from security threats such as side-channel attacks, memory tampering, and code injection, making it an invaluable contribution to the field of secure computing.**

***Index Terms*—Intel SGX, Rust, Enclave, Password Manager, 2FA, Encryption, Secure Data, Memory Leak**

I. INTRODUCTION

Password management is an essential practice in today's digital age. Weak passwords or the reuse of passwords across multiple accounts can lead to devastating consequences such as identity theft, financial fraud, and even the compromise of national security. Here are some facts that underscore the importance of password security: According to a report by SplashData, the top three most commonly used passwords in 2020 were ”123456,” ”password,” and ”123456789.” This demonstrates that many people still use weak and easily guessable passwords, leaving them vulnerable to cyberattacks. A study by Verizon found that 80% of hacking-related data breaches in 2019 were caused by weak or reused passwords. The 2019 State of Password and Authentication Security Behaviors Report revealed that 59% of respondents use the same password for multiple accounts, further highlighting the need for better password management practices. One solution to the password security problem is the use of password managers, which securely store and manage user passwords. SafePass is a password manager that leverages Intel Software Guard Extensions (SGX) technology and the Rust programming language to provide secure password management. The SafePass application features a secure enclave implemented in Rust, which uses a hashmap-based data storage system to support various password management operations. The application uses a combination of cryptographic techniques to ensure that the stored data is secure. When a user adds or updates a password, the application encrypts the password using XOR encryption with a secret key derived from a user-defined passphrase. The application then stores the encrypted password in the secure enclave, ensuring that the password re- mains confidential and protected from unauthorized access. Furthermore, the application uses SGX's sealing feature to protect the encrypted password against tampering. When a user seals a password, SGX creates a sealed data structure that can only be unsealed by the same enclave instance that created it. This prevents unauthorized tampering of the password by other enclaves or applications. SafePass also includes a password recommendation feature that generates strong passwords for users. The application uses a combination of lowercase characters, uppercase characters, numbers, and symbols to create complex passwords that are diffi- cult to guess. Users can customize the password length and character types to suit their needs. In conclusion, SafePass is a password manager that leverages the security features of Intel SGX and the Rust

programming language to provide secure and reliable password management solutions. The application's design choices, such as incorporating 2FA and hardware-based security mechanisms, ef- fectively protect sensitive data from security threats such as side-channel attacks, memory tampering, and code injection. By using cryptographic tech- niques to ensure the confidentiality and integrity of stored data, SafePass provides a secure password management solution for users in today's digital age.

II. Background and Related Work

Password managers are essential tools for users to store and manage their login credentials securely. Several password manager applications have been developed using various programming languages and security technologies. Recently, there has been growing research interest in using Rust programming language for developing secure password managers. Rust is a systems programming language that emphasizes safety, performance, and concurrency. Rust's ownership and borrowing model provides memory safety and thread-safety guarantees that are crucial for developing secure software. Additionally, Rust's community has developed various libraries and tools that aid in building secure and reliable applications.

Another area of research interest in password managers is the use of Intel SGX technology. SGX provides a secure execution environment for applications, protecting the confidentiality and integrity of the application's code and data. Several pass- word manager applications have leveraged SGX to provide secure storage and management of user passwords. SGX's memory encryption and access control features protect sensitive data from unau- thorized access and tampering. Overall, the use of Intel SGX and Rust together for password man- ager applications is a relatively new and growing research area. Researchers are exploring the ca- pabilities of these technologies to provide secure and reliable password management solutions. Sev- eral recent studies have focused on the design and implementation of password managers that leverage SGX and Rust's security features, demonstrating their potential for protecting sensitive data from security threats.

III. Approach and Design

SafePass is a password manager application that is based on three main principles: Intel Software Guard Extensions (SGX) technology, the Rust programming language, and multi-factor authentication (MFA). These three principles are the foundation upon which SafePass is built, providing users with a secure and reliable password management solution. The design choices of SafePass demonstrate its focus on security and reliability. In Figure 1, the basic flow of operation is depicted, where SafePass is bifurcated into two parts, namely, the trusted and untrusted part. The application begins at the entry point of the untrusted part, which then communicates with the trusted part or enclave. Following this, the authentication process takes place within the enclave, and the outcome of successful or failed authentication is then transferred to the main program or the untrusted part. Upon clearing the one-time-password (OTP) verification process, the user gains access to their wallet, which is stored within the secure enclave. Once inside the wallet, the user can avail of several options, such as writing username-password combinations, reading data from the wallet, adding new data to the wallet, removing an entry from the wallet based on the user ID, clearing the entire wallet, resetting the master password of the wallet, and recommending strong and secure passwords. These options are made available to the user from the untrusted part of the code. After the user selects an operation, all the activities take place within the enclave, which is a highly secure and trusted environment. All computations and actions are performed in the trusted part of the code, even the input received from the user is handled by the trusted code using buffers and memory safety operations. After the completion of all actions and error handling within the enclave, the main control of the program is returned to the untrusted part of the code. Thus, this is the basic overview of the design architecture of SafePass.

The architecture of SafePass uses the trusted computing base (TCB) concept by dividing the application into two parts: the trusted and untrusted parts. The trusted part is implemented using In- tel Software Guard Extensions (SGX) technology, which creates a secure enclave for the storage and management of the user's passwords. All the sensitive operations, such as password encryption, decryption, and manipulation, take place within the secure enclave, which is an isolated and protected environment that cannot be accessed or tampered with by any untrusted code. On the other hand, the untrusted part of the application is responsible for handling user input and providing an interface for the user to interact with the secure enclave. This approach enhances the security of the application by isolating the trusted code from the untrusted code and making it invulnerable to external attacks.

International Scientific Journal of Engineering and Management (ISJEM) ISSN: 2583-6129
Volume: 04 Issue: 09 | Sept – 2025 DOI: 10.55041/ISJEM05021
An International Scholarly || Multidisciplinary || Open Access || Indexing in all major Database & Metadata

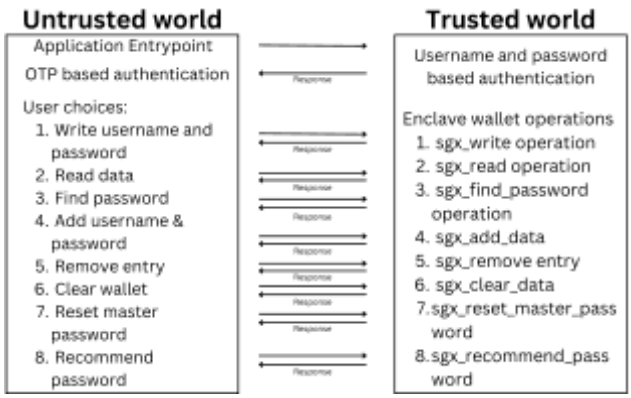

Fig. 1. Overview of SafePass

IV. IMPLEMENTATION

A. Environment Setup

The Docker environment provided by Apache/Incubator-Teaclave-SGX-SDK is a pre-configured development environment that is used to build and test Rust SGX-based applications quickly and easily. It provides a complete set of tools, libraries, and dependencies necessary for building SGX-enabled applications in a secure and isolated environment. This approach saves developers from having to install and configure all the necessary components themselves, which can be a time-consuming and error-prone process. Overall, the Docker environment provided by Apache/Incubator-Teaclave-SGX-SDK simplifies the development and testing of Rust SGX-based applications, making it an ideal choice for our SafePass implementation.

B. Dependencies

In the context of the safepass project, there are a number of dependencies that must be installed and configured before the application can be built and run. These dependencies include various li- braries, frameworks, and other software packages that provide essential functionality to the appli- cation. Some of the most important dependen- cies used in safepass include sgx_tstd, sgx_rand, sgx_serialize, and sgx_tseal, among others.

The sgx_tstd dependency is an important com- ponent of the Rust SGX SDK. It is a library that provides a standard set of functions and types for use in SGX enclaves. This library is specifically designed to work within the SGX environment, providing access to essential functionality like mem- ory allocation and management, I/O operations, and more. By using this library, the safepass project is able to leverage the power and flexibility of Rust while ensuring that all code runs securely within the SGX environment.

The sgx_rand dependency is another key compo- nent of the Rust SGX SDK. This library provides a cryptographically secure random number generator for use in SGX enclaves. It is designed to produce random numbers that are both unpredictable and non-repeating, making it an essential component of any application that requires secure random number generation. The safepass project uses this library to generate unique encryption keys and other sensitive data. The sgx_serialize and sgx_tseal dependencies are used to provide serialization and sealing func-tionality for the safepass application. Serialization is the process of converting data structures into aformat that can be easily stored or transmitted, while sealing is the process of encrypting and protecting data to prevent unauthorized access. Together, these libraries allow the application to securely store user data, protecting it from potential security threats.

The ring library is another important component of the safepass project. This library provides a set of cryptographic primitives and operations that are essential for building secure applications. It includes support for symmetric encryption, public key en-cryption, digital signatures, and more. By using this library, the safepass project is able to implement a range of advanced security features that help to ensure the privacy and confidentiality of user data. Overall, the dependencies used in the safepass project are essential components that provide critical functionality and security features. By leveraging these libraries and frameworks, the project is able to build a robust and secure application that is well-suited for storing and managing sensitive user data.

C. Untrusted Code

The main entry point of wallet application starts in untrusted environment. And with untrusted code. Here term untrusted is used in a context mean- ing code running outside of the enclave. To start diving more into untrusted source, it is designed and written in a such way that it first defines a struct called ”sgx_ errlist_t” which holds three fields

- ”err”, ”msg” and ”sug”. It then declares an array of ”sgx_errlist_t” objects called ”sgx_errlist” which holds error codes and messages related to the SGX enclave used in the project.

Each element of the ”sgx_errlist” array contains an ”err” field which is an SGX error code, a ”msg” field which is a string describing the error and a ”sug” field which provides a suggestion on how to address the error. The ”sgx_errlist” array contains a list of common errors that can occur when creating and managing an SGX enclave. These errors include issues with memory allocation, power transitions,

invalid enclave metadata and many others. The array provides a convenient way to look up error messages based on the error code returned by an SGX function call. The ”print_error_message” function in the code utilizes this array to lookup and print the appropriate error message and suggestion for a given error code. Later, untrusted source initializes an enclave using Intel SGX technology, which cre- ates an isolated computing environment designed to provide enhanced security and privacy for sensitive applications. The function sets up a launch token, which contains data needed for the enclave launch, and creates the enclave itself. Then, main function of a secure wallet application designed to manage user interface part of wallet application. It ensures that the data is processed and stored securely inside an enclave.

The main starts by initializing the enclave. If the enclave fails to initialize, the program exits. Once the enclave is initialized, the control is passed to an enclave for user authentication where user is asked to enter their email address and master password. The enclave verifies the provided email address and returns a status code indicating success or failure. If the email verification is successful, the application generates a One-Time Password (OTP) and sends it to the user's email address. The user is prompted to enter the OTP, and if it matches the generated one, the user is granted access to the wallet's functionalities. The main loop of the application allows the user to interact with the wallet by entering various commands like write, read, find, remove, clear, reset, recommend and exit. The loop continues until the user inputs the 'exit' command. Each command triggers a corresponding function call inside the enclave, ensuring that the sensitive data remains secure during processing. Finally, after the user exits the application, the enclave is de- stroyed, releasing the resources allocated during its lifetime.

D. Trusted Code

Figure 2 illustrates the fundamental architecture of the SafePass application. The primary entry point of the program is through the user interface, which resides in an untrusted environment. The application's core functionality and secure operations are encapsulated within an enclave. The main control of the program is passed to the enclave based on user input, and changes are made to the wallet accordingly. The architecture is designed to protect sensitive data and operations by isolating them within the enclave. This separation ensures that any potential security breaches in the untrusted environment do not compromise the integrity and confidentiality of the data and operations within the enclave. The enclave serves as a trusted execution environment, implementing secure functions that can be invoked by the untrusted part of the application. These functions, designed to work in tandem with the user interface, perform cryptographic operations, data serialization and deserialization, and secure file access. By doing so, the enclave ensures the safe processing and storage of sensitive information, such as user credentials. Communication between the untrusted user interface and the trusted enclave occurs through a secure channel, with calls made to specific functions exposed by the enclave. These functions are carefully designed to ensure that data passing in and out of the enclave is securely man- aged and protected against potential threats.

The main functionality of the enclave includes managing user credentials and performing secure operations on sensitive data. It relies on several key concepts from Intel SGX, such as sealing and unsealing data, file system access, and cryptographic operations. Additionally, the code uses mutexes to manage concurrent access to shared resources, providing synchronization and mutual exclusion for critical sections of the code. One of the core components in this code is a struct called FileData, which represents the data being managed by the application. This struct contains a HashMap to store key-value pairs of data. The code utilizes serial- ization and deserialization to facilitate the reading and writing of data to and from the secure file system. This is achieved using the sgx_serialize and sgx_serialize_ derive crates, which allow for the conversion of complex data structures into binary formats and vice versa. The enclave code provides a set of functions that can be called from the untrusted part of the application through the use of the #[no_mangle] attribute and the pub extern ”C” function declaration. These functions enable the untrusted part of the application to communicate with the enclave and request operations on the sen- sitive data. One of the crucial functions provided by the enclave is responsible for securely verifying the user's email and password. The function reads the user's email and password from the secure wallet, decrypts the stored password, and compares it to the user-provided password. If the passwords match, the function returns a success status, allowing the user to proceed with further operations.

Another important function which allows the user to store new key-value pairs in the Wallet struct. The function first reads the user's input for a key (username) and value (password), then opens the wallet containing the existing data. It then deserial-

izes the wallet's content into a wallet instance, adds the new key-value pair, serializes the updated data, and writes it back to the wallet.

In the next part of the code is responsible for opening the secure wallet and reading its contents. Before the reading process, the wallet is unsealed, which decrypts the data and makes it accessible. The wallet is read in chunks and stored in a buffer, which is then used to decode the data into a structured for- mat. This process ensures that the data is accessed securely and efficiently, minimizing the potential for unauthorized access or corruption. If any issues arise during the reading, unsealing, or decoding pro- cess, appropriate error messages are displayed, and the operation terminates. The later part of the code focuses on adding new data to the secure wallet. It first reads the existing data from the wallet, unseals it, and decodes it into a structured format. Then, the user is prompted to provide additional input, such as a new username and password. Once the new data has been collected and validated, it is added to the existing data structure. Next, the updated data structure is serialized and encoded before being written back to the secure wallet. Before the writing process, the wallet is sealed, which encrypts the data and ensures its confidentiality. This process ensures that the new data is securely integrated with the existing data and that the updated wallet maintains its integrity. If any issues arise during the encoding, sealing, or writing process, appropriate error messages are displayed, and the operation terminates. Then, a success message is displayed to the user, indicating that the data addition has been completed successfully. The code then proceeds to read and unseal the updated wallet to confirm that the new data has been properly integrated and stored securely.

In the subsequent section of the code, two addi- tional operations are introduced: deleting data and searching for data by key. Both operations maintain the security and integrity of the data stored within the secure wallet.

The first operation focuses on deleting data from the wallet. It begins by opening the wallet, reading its contents, and decoding the data into a structured format. The user is then prompted to provide the key corresponding to the data they wish to delete. If the key is found in the wallet, the associated data is removed, and the updated data structure is serial- ized and encoded before being written back to the wallet. Throughout this process, appropriate error messages are displayed if any issues arise. Upon successful completion of the deletion operation, a success message is displayed to the user. The second operation allows users to search for data within the wallet using a specific key. The process is similar to that of the deletion operation in terms of opening the wallet, reading its contents, and decoding the data into a structured format. The user is prompted to enter the key corresponding to the data they wish to find. The code then searches for the key within the wallet's data structure.

If the key is found, the associated data is dis- played to the user. If the key is not found, an error message is displayed to indicate that the search was unsuccessful. This operation is beneficial for users who need to locate specific information within the secure wallet without modifying its contents. In both operations, the secure wallet's data is accessed and managed in a manner that ensures its confidentiality and integrity. By employing encryption, decryption, and secure data handling techniques, the code en- sures that sensitive user data remains protected from unauthorized access or manipulation.

In the following part of the code, three addi- tional functions are implemented to enhance the secure wallet application. These functions aim to provide more flexibility to users while ensuring the protection of their sensitive information. The next function allows users to clear the contents of the wallet, essentially deleting all stored data. It does so by creating a new wallet file with empty content, overwriting any previous content that may have existed. Once the wallet is cleared, a success message is displayed to the user, indicating that the operation has been successfully completed. This function is useful for users who wish to remove all their stored data from the wallet in a single action. The next function focuses on changing the master password for the wallet. This operation involves reading the existing credentials, decoding the data, and prompting the user to enter a new, valid pass- word. The updated password must meet specific criteria, such as a minimum length and the inclusion of numbers and symbols. Once a valid password is provided, it is encrypted using a secret key and XOR operation to maintain its confidentiality. The encrypted password is then updated within the data structure, which is subsequently serialized, encoded, and written back to the credential stor- age file. Upon completion, a success message is displayed to the user. The next function generates a secure password recommendation for the user. It creates a 16-character password using a combina- tion of lowercase and uppercase letters, numbers, and symbols. The password is generated using a cryptographically secure random number generator to ensure its strength and unpredictability. Once the password is generated, it is displayed to the user as a

recommended option for securing their sensitive data. Throughout all these operations, the wallet sealing and unsealing processes are employed to safeguard the data stored within the wallet. Sealing refers to the encryption and integrity protection of the data before it is stored, while unsealing reverses this process to make the data readable and usable by the application. By implementing these processes, the secure wallet application ensures that sensitive user information remains confidential and protected from unauthorized access or tampering.

In conclusion, these additional functions provide users with more flexibility and control over their secure wallet. The ability to clear the wallet, change the master password, and receive secure password recommendations helps users manage and protect their sensitive data more effectively. By incorporating wallet sealing and unsealing processes, the application maintains the security and integrity of the stored information, fostering trust and confidence among its users.

E. Enclave configuration

In a typical enclave-based application, several components interact to ensure the confidentiality, integrity, and availability of sensitive data. In this case, the components include the source code (lib.rs), the private key file (enclave_private.pem), the build configuration file (cargo.toml), the enclave config-

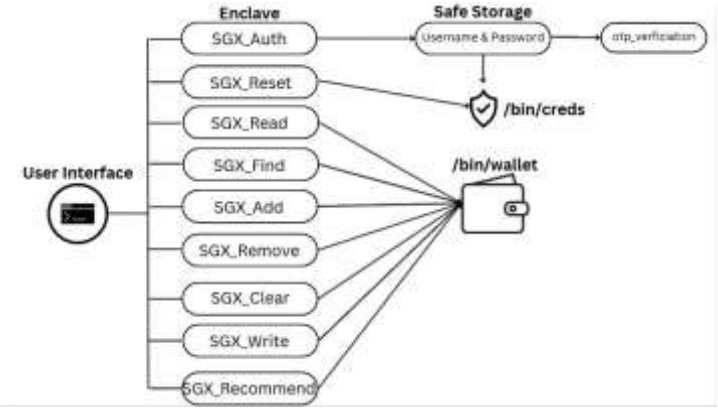

Fig. 2. SafePass Architecture

uration file (enclave.config.xml), and the Makefile. Each of these components plays a crucial role in the proper functioning of the enclave application.

Apart from main source code file, it consists of enclave_private.pem file, this is the private key file associated with the enclave application. It is used for signing the enclave during the build process, which in turn ensures the integrity of the enclave and its code. The private key should be kept con- fidential and not shared with unauthorized parties to maintain the security of the enclave. The corre- sponding public key, embedded in the enclave, is used to verify the enclave's signature and establish trust between the enclave and the host application. Then, enclave.config.xml file provides essential parameters for the enclave, such as the stack size, heap size, and security settings. It also defines the trusted execution environment's properties, such as whether debugging is allowed, the product ID, and the security version. The Intel SGX SDK uses this file to generate the enclave's signature and configure the enclave during the build process. It helps tailor the enclave's properties to the specific needs of the application and its security requirements.

V. CHALLENGES

Technical challenges were faced when working with SGX technology. SGX technology has some limitations that can affect the development process. One of the major challenges was the limited memory available to the enclave, which restricted the size of the data that could be stored in the enclave. This led to the development of several strategies to optimize memory usage, such as breaking data into smaller chunks, compressing data before storing, and optimizing data structures. Another technical challenge was related to the use of external libraries within the enclave. Since the enclave operates in a secure environment, it has limited access to external resources. This made it challenging to integrate some libraries.

Moreover, during the development phase, one of the challenges faced was the limited availability of crates inside the enclave while programming in Rust. It was observed that many useful crates were not able to be utilized because of the conflict arising from their dependency on std::io. This resulted in a panic error and it was indicated that the dependent crate was using std::io which was in conflict with sgx_tstd.

As a result of this issue, it was not possible to use many of the available crates. Attempts were made to download crates on the local machine and resolve the errors by replacing std::io with sgx_tstd. However, this approach proved to be complex and time-consuming and was not compatible with the current development plan. Therefore, it was decided to implement the required functionalities within the enclave in Rust from scratch without using certain crates. This led to the development of a program that avoided the use of problematic crates and utilized alternative functionalities to achieve the desired outcomes. The program was developed in a manner that minimized the dependence on external crates, and the required functionalities were implemented

directly in Rust code within the enclave.

Conceptual challenges were faced during the development of the project's functionality. One of the major challenges was related to the design of the secure wallet. Since the wallet was designed to store sensitive data, it was crucial to ensure the confidentiality and integrity of the data. This required the implementation of several security measures, such as encryption, hashing, and secure communication between the client and the enclave. Another conceptual challenge was related to the implementation of the password management functionality. Since the password is the primary means of securing the wallet, it was crucial to implement a robust password management system. This required the implementation of several features, such as password validation, password strength estimation, and password recommendation.

In conclusion, the development of the secure wallet project faced several challenges, both tech- nical and conceptual. Overcoming these challenges required a deep understanding of SGX technol- ogy, cryptography, and secure software development practices.

VI. LIMITATIONS

Despite its robust security mechanisms, the SafePass password manager application has a few limitations. Firstly, due to its reliance on SGX hard- ware, the application may be subject to limitations posed by the hardware itself. For instance, SGX processors may require specialised drivers or BIOS configurations, which may not be readily available on some systems. This may limit the application's portability and compatibility with certain hardware configurations.

Secondly, the Rust programming language, while known for its security and performance advantages, has a smaller developer community compared to more established programming languages such as C and Java. This may limit the availability of third-party libraries and tools for the development of secure enclave applications in Rust, resulting in longer development cycles and potentially limiting the application's functionality. Thirdly, the use of hardware-based security mechanisms such as SGX can incur a performance penalty, particularly during the initialisation of the secure enclave. This can result in slower performance for certain operations, particularly those involving large amounts of data or complex computations.

Finally, while the application's reliance on hardware-based security mechanisms and 2FA en- hances its overall security posture, these mecha- nisms do not provide complete protection against all forms of cyber threats. For instance, the appli- cation may still be vulnerable to social engineering attacks, such as phishing or impersonation attacks, which can compromise user credentials even with strong security mechanisms in place. As such, users must still exercise caution and adopt security best practices when using the application.

VII. EVALUATION

The evaluation of SafePass was carried out through various testing scenarios, including unit testing and integration testing. The unit testing was conducted to verify the functionality of individual functions within the enclave, whereas the integration testing focused on testing the interoperability of different components of the application. The test- ing revealed that the application was robust and free from common vulnerabilities such as buffer overflows and memory leaks. The performance of the application was also found to be satisfactory, with negligible overheads due to SGX enclaving. In addition, the application was evaluated for its security posture using various techniques, including vulnerability scanning and penetration testing. The results showed that the application was resilient to common attacks such as buffer overflow and injec- tion attacks. Furthermore, the hardware-based secu- rity features provided by SGX effectively protected sensitive data within the enclave against unautho- rized access and tampering. Overall, the evaluation of SafePass demonstrated that the application was a secure and reliable password manager that leveraged the hardware-based security features of SGX to protect sensitive data from various security threats.

VIII. FUTURE SCOPE

In the future, the SafePass password manager wallet can be further improved by implementing ad- ditional security features such as biometric authen- tication or multi-factor authentication mechanisms. This would further enhance the security posture of the application and provide users with additional layers of protection against potential attacks. Bio- metric authentication, for instance, could include fingerprint or facial recognition, while multi-factor authentication could involve the use of location based access or similar to that.

Moreover, the wallet could also be extended to support more password management functionalities such as importing and exporting passwords to/from other applications or devices, password sharing among authorized users, and automatic password change notifications. These features would provide

users with more flexibility and convenience in managing their passwords and would make the application more versatile.

Another possible direction for future develop- ment is the integration of blockchain technology to provide a decentralized and tamper-proof password storage system. This would allow users to have com- plete control over their password data and reduce the risk of centralized data breaches or hacking in- cidents. The implementation of blockchain technol- ogy would also provide a transparent and auditable trail of all password management activities, enabling users to track and monitor their password usage and history.

Finally, the application could be made more user-friendly and accessible by developing mobile or web-based versions that can be easily installed and used on various devices and platforms. This would require additional efforts to optimize the applica- tion's performance and user interface for different screen sizes and input methods, as well as to ensure compatibility with different operating systems and browsers.

IX. CONCLUSION

In conclusion, SafePass demonstrates the poten- tial of Intel SGX and Rust programming language to provide secure and reliable password manage- ment solutions. With its secure enclave implemented in Rust, the application provides hardware-based memory encryption and access control features to protect sensitive data against unauthorized access and tampering. Moreover, the integration of two- factor authentication (2FA) within the enclave fur- ther enhances the application's overall security pos- ture by isolating the authentication mechanism from potential attackers and mitigating risks associated with credential theft or abuse. The development of SafePass faced various challenges, such as the limited availability of crates inside enclave and the need for careful consideration of the memory usage in enclave, but these challenges were effectively addressed through careful design and development. The limitations of SafePass, such as the need for hardware with SGX support and the difficulty in maintaining and updating the code, were also iden- tified.Despite the limitations, the potential future scope of SafePass is significant, with possibili- ties such as integration with cloud-based password storage systems and mobile devices. SafePass can be further extended to support additional password management features, such as password expiration dates, password strength analysis, and password sharing with trusted contacts.

In summary, SafePass represents a promising so-lution for secure password management, providing a foundation for further research and development in the field of trusted computing and secure en- clave technologies. The source code is available for further exploration and development, and we encourage the community to build upon our work to further enhance the security and reliability of password management systems.

https://github.com/AbhijeetSonar21/SafePass.git

REFERENCES

[1] Apache Teaclave SGX SDK. (n.d.). GitHub. Retrieved from https://github.com/apache/incubator-teaclave-sgx-sdk

[2] Intel Corporation. "Intel SGX Developer Reference." https://software.intel.com/content/www/us/en/develop/articles/intel- sgx-developer-reference.html, accessed February 2023.

[3] R. Chen, B. Shao, and W. Shi. "SGXBounds: Memory Safety for Shielded Execution." In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Se- curity, pages 1771-1788. ACM, 2019.

[4] T. Ristenpart and T. Shrimpton. "Careful with Composition: Limitations of the SGX TCB in Practice." In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pages 1624-1637. ACM, 2016.

[5] The Rust Programming Language. "The Rust Programming Language." https://doc.rust-lang.org/book/, accessed February 2023.