Tamper Proof Digital Vault (TPDV)

Tiago Silvestre - tiago.silvestre@ua.pt, Diogo Matos - dftm@ua.pt, David Araújo - davidaraujo@ua.pt

May 12, 2024

Table of Contents

Introduction

The objective of this project is to develop a "tamper-proof digital vault" (TPDV) leveraging Intel's SGX enclaves. The TPDV is designed to securely store digital assets in a write-once manner, preventing any alteration or deletion of stored information. Although a malicious operator could potentially destroy the entire TPDV, they cannot modify any existing data without detection.

This report offers a concise overview of our TPDV implementation, along with the results from various conducted tests. We commence by outlining the overall structure of the system, followed by a detailed examination of its individual feature by feature. Finally, we consolidate our findings in the conclusions.

Overall structure

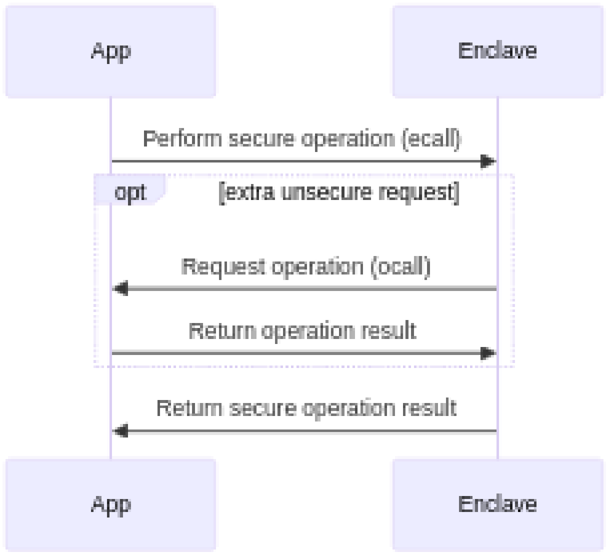

App
Enclave
Perform secure operation (ecall)
opt
[extra unsecure request]
Request operation (ocall)
Return operation result
Return secure operation result
App
Enclave

Figure 1: Overall flow of a vault operation

There are two modules in this project: the application `app.cpp` and the enclave `Enclave1.cpp`. The code within the application module runs **without any additional isolation**, while the code within `Enclave1.cpp` executes within the secure confines of an SGX enclave.

Upon initialization, the application creates the enclave, and **whenever a critical operation is required**, such as accessing sensitive data, it invokes an `ecall` to execute a method within the enclave. Although the enclave operates within a restricted environment, there are instances where it may **need to perform non-sensitive actions, such as printing or reading a file, which can be safely executed outside the enclave**. In such cases, the **enclave utilizes an `ocall`** to invoke methods external to the enclave.

In our specific scenario, the portion of the program running outside the enclave primarily serves as an interface for user interaction with the vault. All operations directly involving the vault's data are securely conducted within the enclave.

Features and tests

Create a vault

Initially, users should have the capability to create a vault from within the application. This vault includes several discretionary attributes, namely a **name**, a **password** (for user access), the **author's name**, and a counter for the **number of files** it contains. To create the vault, users must provide the first three attributes.

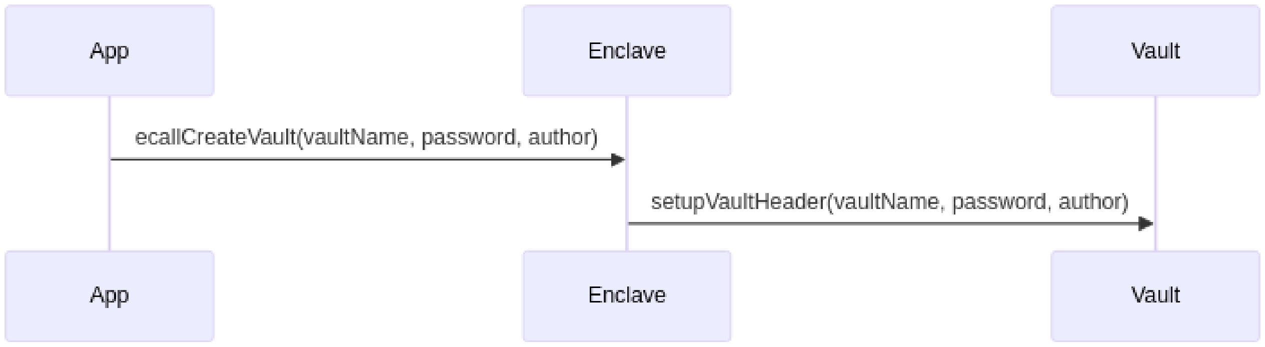
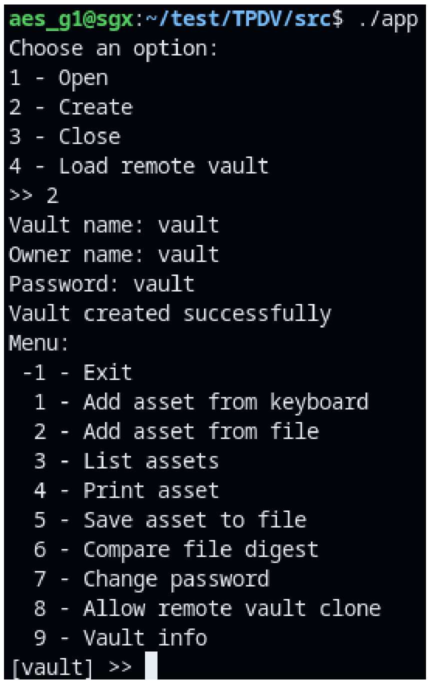

App
Enclave
Vault
ecallCreateVault(vaultName, password, author)
setupVaultHeader(vaultName, password, author)
App
Enclave
Vault

Figure 2: Create vault flow

The `setupVaultHeader` function will also be utilized whenever a vault is loaded, ensuring that a new random nonce is stored in the vault's header for future utilization.



```
aes_g1@sgx:~/test/TPDV/src$ ./app
Choose an option:
1 - Open
2 - Create
3 - Close
4 - Load remote vault
>> 2
Vault name: vault
Owner name: vault
Password: vault
Vault created successfully
Menu:
 -1 - Exit
  1 - Add asset from keyboard
  2 - Add asset from file
  3 - List assets
  4 - Print asset
  5 - Save asset to file
  6 - Compare file digest
  7 - Change password
  8 - Allow remote vault clone
  9 - Vault info
[vault] >>
```

Figure 3: Creating a vault

Insert assets

To make the vault functional, users need the ability to deposit assets into it. This can be achieved in two ways: either by entering content directly from `stdin` or by uploading a file. Regardless of the chosen method, the data is initially acquired from various sources and then transferred to the enclave via an `ecall`.

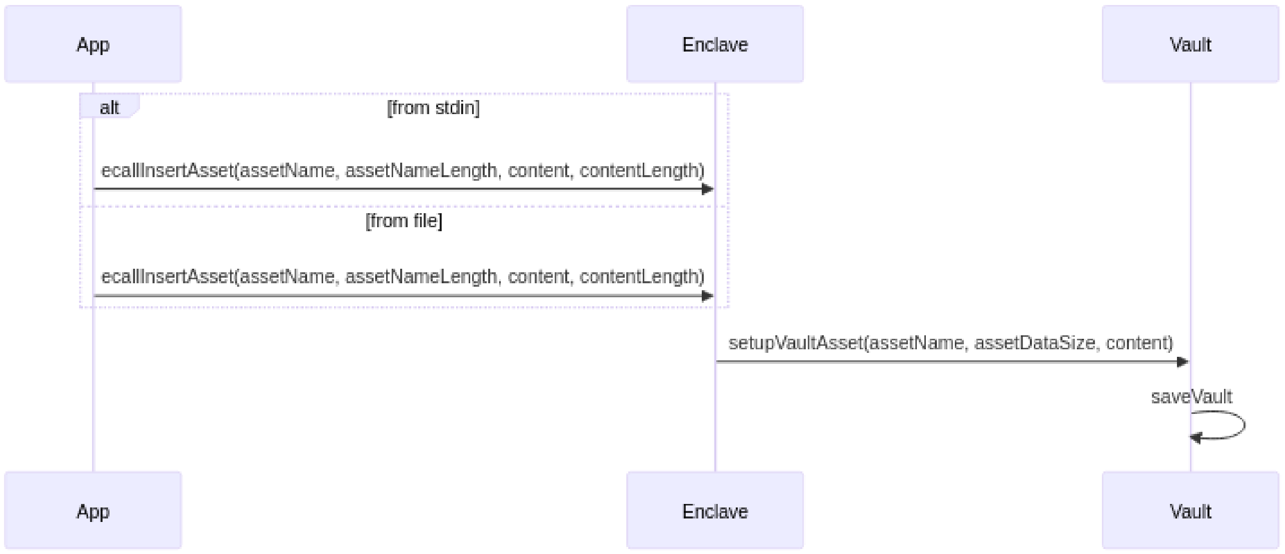

App
Enclave
Vault
alt
[from stdin]
ecallInsertAsset(assetName, assetNameLength, content, contentLength)
[from file]
ecallInsertAsset(assetName, assetNameLength, content, contentLength)
setupVaultAsset(assetName, assetDataSize, content)
saveVault
App
Enclave
Vault

Figure 4: Insert asset vault flow

A bidirectional linked list of assets is established to manage the stored assets within the vault. Each new asset is appended to this list. Each vault asset includes pointers to the previous and next assets, along with the raw content, content size, name, and SHA256 hash of the asset.

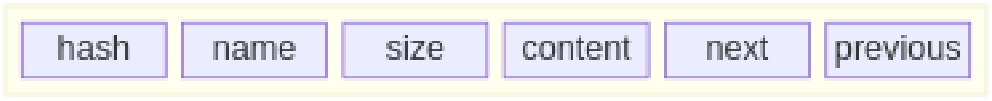
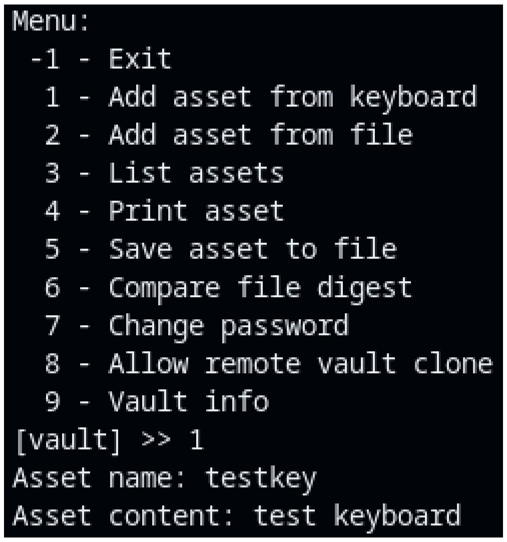
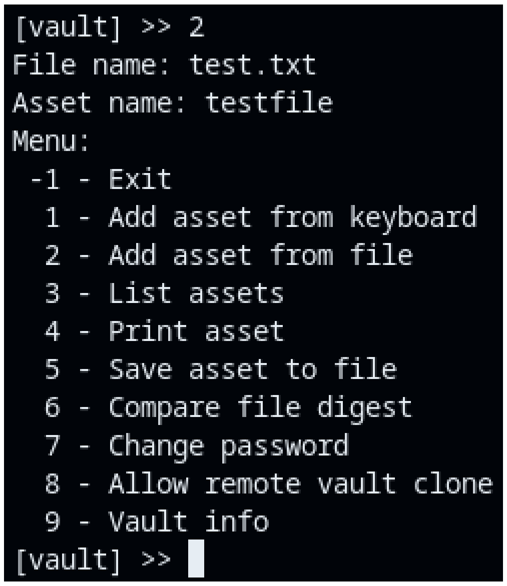


Figure 5: Asset structure



```
Menu:
 -1 - Exit
  1 - Add asset from keyboard
  2 - Add asset from file
  3 - List assets
  4 - Print asset
  5 - Save asset to file
  6 - Compare file digest
  7 - Change password
  8 - Allow remote vault clone
  9 - Vault info
[vault] >> 1
Asset name: testkey
Asset content: test keyboard
```

Figure 6: Insert asset via stdin



```
[vault] >> 2
File name: test.txt
Asset name: testfile
Menu:
 -1 - Exit
  1 - Add asset from keyboard
  2 - Add asset from file
  3 - List assets
  4 - Print asset
  5 - Save asset to file
  6 - Compare file digest
  7 - Change password
  8 - Allow remote vault clone
  9 - Vault info
[vault] >>
```

Figure 7: Insert asset via stdin

Saving the vault

After pushing the new assets into the vault, the vault is saved. This phase is particularly critical as it's where the application ensures the data is correctly organized for future use and interpretation.

Initially, we allocate sufficient memory to store the vault headers and the content from the assets that was pushed into the vault. **The vault header size is fixed, being immutable, while the asset list is dynamic and can accommodate varying numbers of assets.** The assets are stored as a continuous data block, with each asset forming a segment of this block structured as follows.

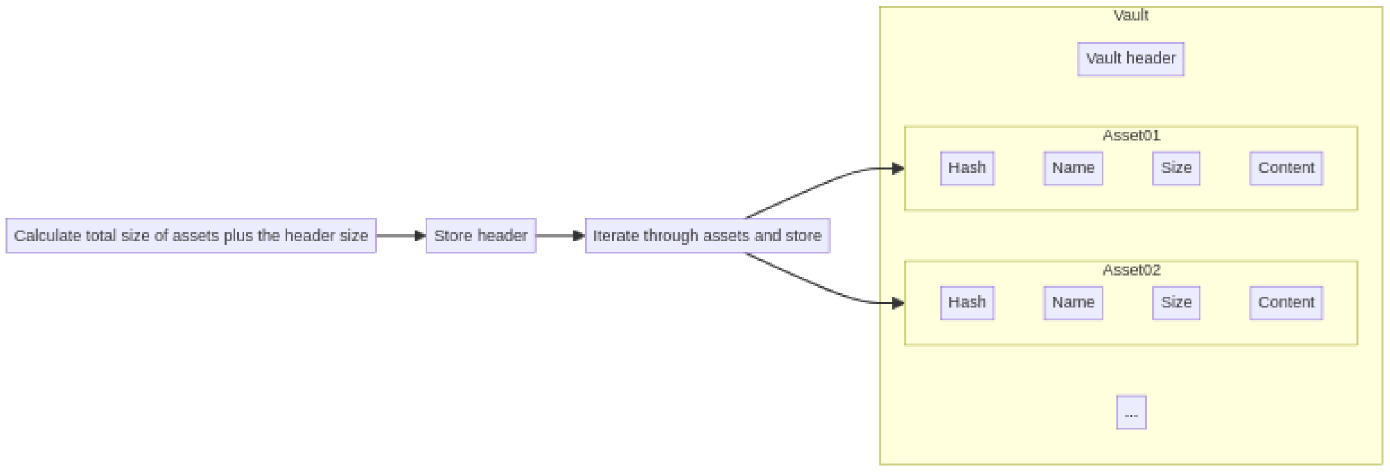


Figure 8: Saving vault structure

Consequently, the application iterates through the list of assets, copying the content of each node. This content encapsulates all the necessary information to reconstruct the file and extract it from the vault later. Additionally, it stores the hash of each asset for subsequent use.

Loading the vault

When loading the vault from a file, the user is required to provide the vault name and password. This information is then passed via an `ocall` to be processed within the enclave. Inside the enclave, the provided password is matched against the one stored within the vault. If the comparison is successful, the vault is loaded and made accessible.

The initial step involves unsealing the data (detailed explanation found here), followed by mounting the vault header and loading the assets into the bidirectional asset list.

```
...
ret = sgx_unseal_data(sealed_data, NULL, NULL, (uint8_t *)unsealed_data, (uint32_t *)&sealed_size);
...
```

As the application cannot blindly trust that the vault has not been tampered with, it employs a verification process: upon loading each asset, it calculates its hash and compares it against the hash stored alongside with the asset itself. While this solution isn't foolproof, as an attacker capable of modifying the content could also alter the hash, it does provide a degree of protection against unauthorized data tampering or data corruption.

```
...
sgx_sha256_hash_t hash;
memcpy(&hash, &unsealed_data[i + 68], 32);

setupVaultAsset(newAsset, &unsealed_data[i + 32], unsealed_data[i + 64], (unsigned char *)&unsealed_data[i +

if (!ecallCheckDigest(newAsset->name, (const char *)hash))
{
    return 2;
}
...
```

List assets

Listing files is also managed through an `ecall`, which triggers the enclave code to iterate through the vault assets and print relevant information such as name and size.

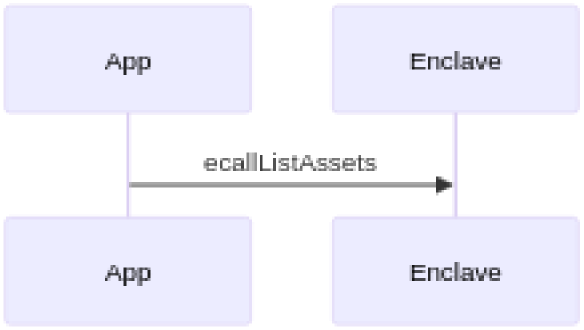
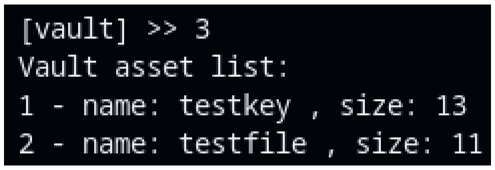


Figure 9: List assets flow

```
...
while (node != NULL)
{
    enclavePrintf("%d - name: %s , size: %d \n", i, node->name, node->size);
    node = node->next;
    i++;
}
...
```



```
[vault] >> 3
Vault asset list:
1 - name: testkey , size: 13
2 - name: testfile , size: 11
```

Figure 10: List assets

Print assets

Printing assets operates similarly to the previous functionality. However, in this case, the `ecall` prompts the enclave to print the content of the files to `stdout`. While this is highly useful for text-based content, it may not be as practical for binary files, as their raw binary data will be printed as output.

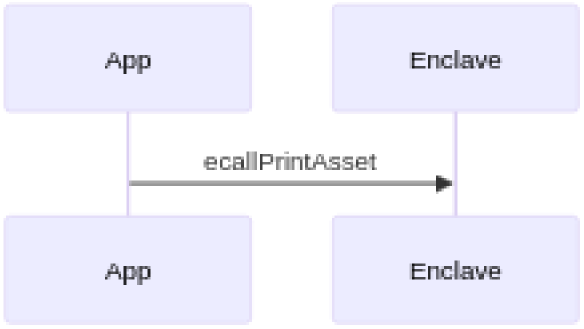
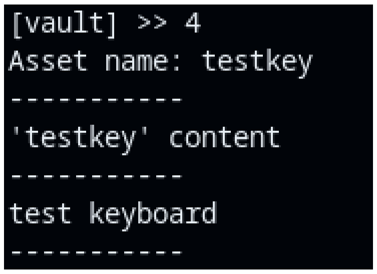


Figure 11: Print asset flow

```
...
while (node != NULL && strcmp(node->name, name) != 0)
node = node->next;

if (node != NULL)
{
    enclavePrintf("----------\n'%s' content \n----------\n%s\n----------\n", name, node->content);
    return 0;
}
...
```



```
[vault] >> 4
Asset name: testkey
-----------
'testkey' content
-----------
test keyboard
-----------
```

Figure 12: Print assets

Export assets

If a user chooses to retrieve a file it has stored as an asset in the vault, it must be able to export it to a new file. Calling `ecallSaveAssetToFile` and providing the asset name and a new for the new file, the enclave will be able to handle this request.

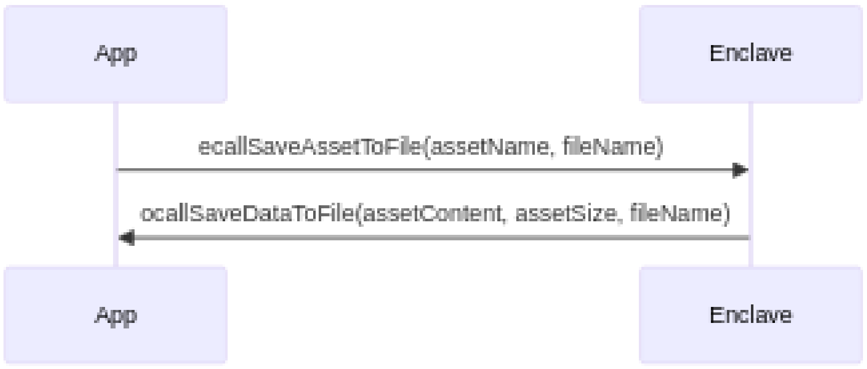
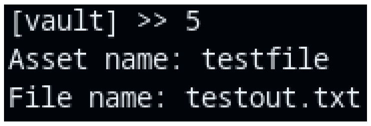

App
Enclave
ecallSaveAssetToFile(assetName, fileName)
ocallSaveDataToFile(assetContent, assetSize, fileName)
App
Enclave

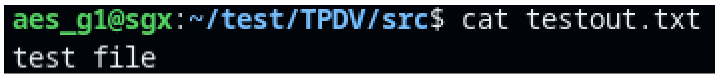
Figure 13: Export asset flow

When a user opts to retrieve a file stored as an asset in the vault, they must be able to export it to a new file. By invoking `ecallSaveAssetToFile` and specifying the asset name along with a new file name, the enclave can process this request.



```
[vault] >> 5
Asset name: testfile
File name: testout.txt
```

Figure 14: Export asset to file



```
aes_g1@sgx:~/test/TPDV/src$ cat testout.txt
test file
```

Figure 15: Content of exported file

Compare assets hash

A user may also wish to verify if an asset stored in the vault is the correct one. For this purpose, they should be able to compare its hash with one they possess. By invoking `ecallCheckDigest`, the user can provide the text-based hash, typically obtained from tools like `sha256sum`, to match it against the one calculated by the enclave. This comparison enables users to confirm the integrity and authenticity of the stored asset.

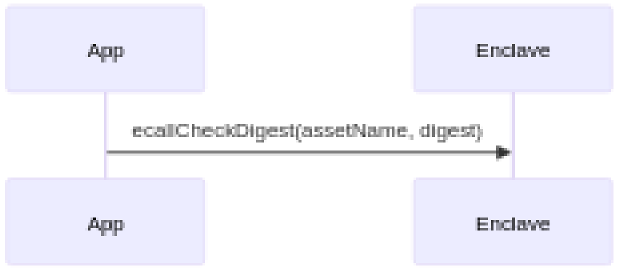
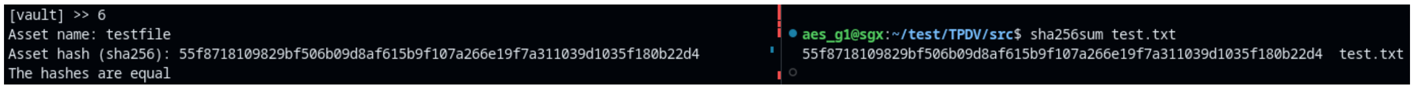


Figure 16: Compare asset hash flow



```
[vault] >> 6
Asset name: testfile
Asset hash (sha256): 55f8718109829bf506b09d8af615b9f107a266e19f7a311039d1035f180b22d4
The hashes are equal
```

```
aes_g1@sgx:~/test/TPDV/src$ sha256sum test.txt
55f8718109829bf506b09d8af615b9f107a266e19f7a311039d1035f180b22d4  test.txt
```

Figure 17: Comparing hashs

Change passwords

A user may wish to change the password for the vault, a process that needs to be conducted securely within the enclave. This can be accomplished by calling `ecallChangePassword`.

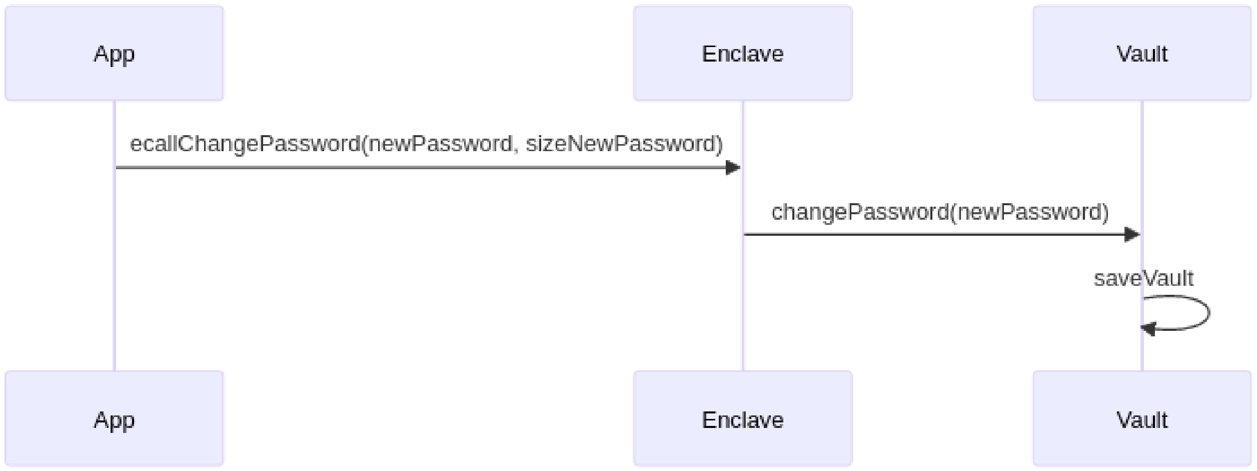
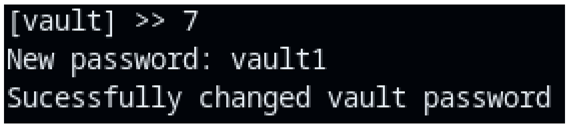

App
Enclave
Vault
ecallChangePassword(newPassword, sizeNewPassword)
changePassword(newPassword)
saveVault
App
Enclave
Vault

Figure 18: Change password flow

This operation will replace the password stored in the header with the new one provided. Subsequently, saving the vault will seal the data, ensuring its security.



```
[vault] >> 7
New password: vault1
Sucessfully changed vault password
```

Figure 19: Changing password of the vault

Seal and unseal

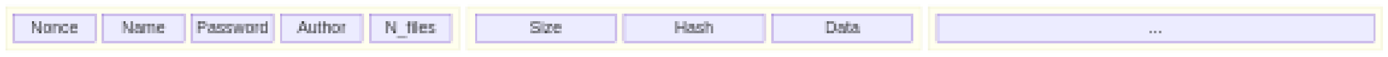


Figure 20: Sealed data structure

To ensure persistence across multiple executions of the enclave, the vault data needs to be exported into a file that can later be imported. To achieve this, whenever the vault is created or modified, all of its data is sealed and stored in a file with the same name as the vault. The sealing process utilizes keys generated by the enclave, derived from a CPU-specific key and a key extracted from the logs generated during enclave creation.

With the sealed data stored within the vault file, the program is capable of loading the vault while maintaining its previous state. Additionally, leveraging the stored hash values for each file, it can detect instances where files have been corrupted.

Clone vault

Implementing the cloning of a vault from a remote host was achieved using TLS communication. This involved setting up a TLS server that waits for client connections, and a client that initiates the request for the vault.

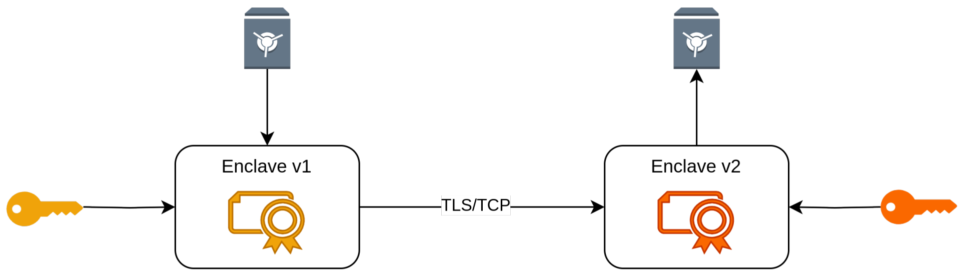


Figure 21: Clone structure

TLS operation necessitates trusted certificates for proper functioning. To fulfill this requirement, a script (obtained from (here)[https://github.com/diogoftm/simulated-kms/blob/main/certs/makefile]) was utilized to generate certificates signed by a self-signed CA. These certificates are then loaded by both the server and the client.

In the `src/App/AppSocket.cpp` file, you can find the implementation of both the client and server components. We've developed a simple message exchange protocol, termed the `BaseMessageLayer`, to support basic communication.

The cloning process builds upon this protocol. Both server and client communication rely on callbacks passed as arguments to setup functions, as demonstrated in `TlsClient::connect(...)` and `TlsServer::run_server(...)`. These callbacks, namely `serveClientCallback(SSL* ssl)` and `clientConnectionWithServerCallback(SSL* ssl)`, are defined in `src/App.cpp`.

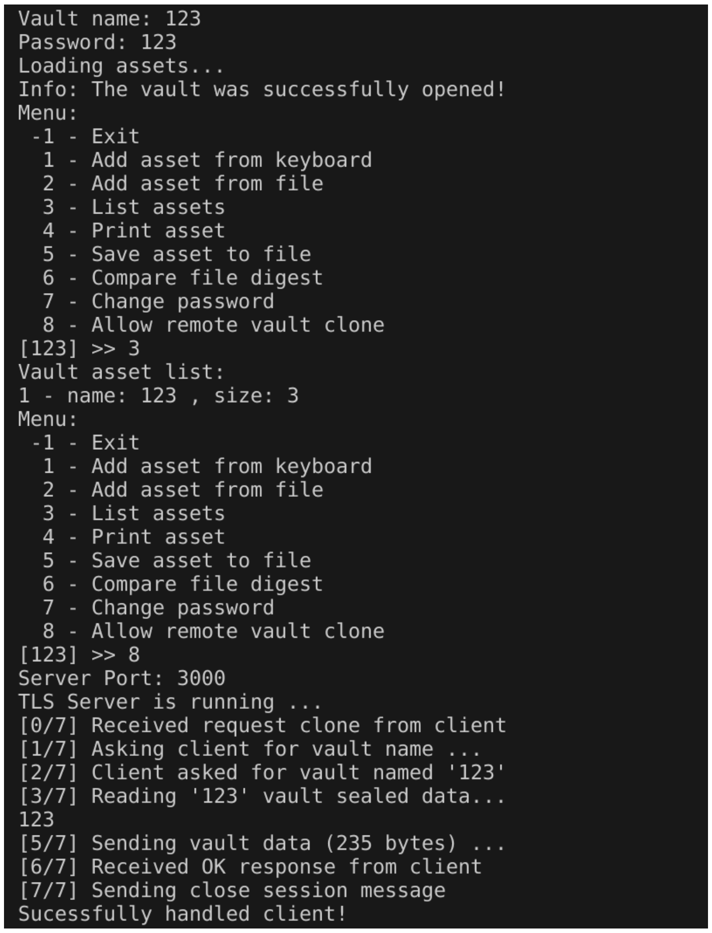
The cloning operation is divided into several phases. After initializing communication, the following steps occur (assuming success).

1. Client sends a request clone message.
2. Server asks for vault name.
3. Client asks user for vault name, (if it's not present in server, server responses with a invalid vault message response and the communication terminates).
4. Server transmits unsealed vault data (encrypted with hash of the vault password using AESGCM algorithm), data is obtained from `ecallGetUnsealedCipheredData`.
5. Client sends an ok message after clone is completed.
6. Server sends a close session message.

After the cloning process is completed, the client prompts the user for the vault password and decrypts it within the enclave using the same process as the server (AESGCM with the password hash).

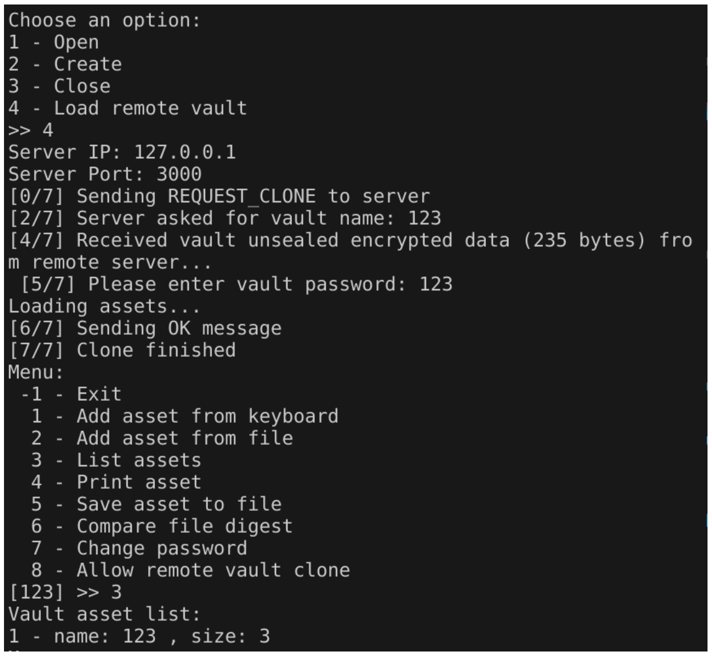
Currently, the server does not validate the client certificate. This functionality could be implemented by introducing additional steps, such as requesting the client certificate, verifying if it matches the expected certificate on the server side, and then requesting a challenge to be signed by the client as proof that it possesses the private certificate key.

Server



```
Vault name: 123
Password: 123
Loading assets...
Info: The vault was successfully opened!
Menu:
 -1 - Exit
  1 - Add asset from keyboard
  2 - Add asset from file
  3 - List assets
  4 - Print asset
  5 - Save asset to file
  6 - Compare file digest
  7 - Change password
  8 - Allow remote vault clone
[123] >> 3
Vault asset list:
1 - name: 123 , size: 3
Menu:
 -1 - Exit
  1 - Add asset from keyboard
  2 - Add asset from file
  3 - List assets
  4 - Print asset
  5 - Save asset to file
  6 - Compare file digest
  7 - Change password
  8 - Allow remote vault clone
[123] >> 8
Server Port: 3000
TLS Server is running ...
[0/7] Received request clone from client
[1/7] Asking client for vault name ...
[2/7] Client asked for vault named '123'
[3/7] Reading '123' vault sealed data...
123
[5/7] Sending vault data (235 bytes) ...
[6/7] Received OK response from client
[7/7] Sending close session message
Sucessfully handled client!
```

Client



```
Choose an option:
1 - Open
2 - Create
3 - Close
4 - Load remote vault
>> 4
Server IP: 127.0.0.1
Server Port: 3000
[0/7] Sending REQUEST_CLONE to server
[2/7] Server asked for vault name: 123
[4/7] Received vault unsealed encrypted data (235 bytes) fro
m remote server...
 [5/7] Please enter vault password: 123
Loading assets...
[6/7] Sending OK message
[7/7] Clone finished
Menu:
 -1 - Exit
  1 - Add asset from keyboard
  2 - Add asset from file
  3 - List assets
  4 - Print asset
  5 - Save asset to file
  6 - Compare file digest
  7 - Change password
  8 - Allow remote vault clone
[123] >> 3
Vault asset list:
1 - name: 123 , size: 3
```

Conclusions

This project showcases the robust capabilities of Intel SGX as a secure mechanism for safeguarding sensitive data during execution. By encapsulating critical processes within SGX enclaves, it creates a fortified environment where data handling occurs with heightened security.

This vault empowers users to confidently manage, store, and transport sensitive data that would otherwise be vulnerable to exposure. Leveraging Intel SGX enclaves ensures that even in potentially compromised environments, data remains protected, preserving its confidentiality and integrity throughout its lifecycle.