1 System Design

1.1 General Design

Every user has a unique User structure and a unique FileInfo structure. The User structure stores the username, the userpassword, the address of the FileInfo structure, the key to decrypt FileInfo structure, and the user's private key. The FileInfo structure stores five []byte arrays, including the filename, key to decrypt the file data, key for checking HMAC of the file data, the address for storing the file data, the salt used for appending a file, and the address for storing the number of appends conducted to the file. One index and the corresponding element of the five arrays uniquely identify a file.

Table 1: Design Form

Purpose	Key String	Value
get FileInfo structure	FileInfoAddress in userdata	V1=Nonce+E(key=FileInfoKey in userdata,FileInfo)
get userdata structure	Argon2Key(username+password,salt="name_hash")	V2=Nonce+E(key=Argon2Key(name+password,salt="saltforuserkey"),Userdata)
check HMAC of FileInfo	Argon2Key(username+password,salt="fileinfoHMAC")	HMAC(key=password,V1)
check HMAC of userdata	Argon2Key(username+password,salt="userHMAC")	HMAC(key=password,V2)
get appended times	NumberAddress in FileInfo	Nonce+E(key=DecryptedKey in FileInfo,number)
get file data	StoreAddress in FileInfo	Nonce+HMAC(key=HMACKey in FileInfo,ciphertext),E(key=Key in FileInfo,data)
get the appended data	hash(salt+StoreAddress(from FileInfo)) for n times	Ibid, HMACKey=hash(salt+HASHKey) for n times,key=hash(salt,Key) for n times

1.2 InitUser

The function first stores the username and the password, then generates the RSAKeyPair, stores the public key in key store at location hash(username), stores the private key in the structure. Then it initializes a new FileInfo without data. Prepare the random bytes Key and address, and the Nonce for encrypting the file info. Store the $Nonce + E(Key, file\_info)$ at the generated random address. Then we generate an IV to encrypt the User structure, the key for encryption is $Argon2Key(password)$, with a constant salt. Store the encrypted at address $Argon2Key(salt = "User", username + password)$. In the end, we stored the HMAC of structure FileInfo and User at an address generated by $Argon2Key(salt = fixed\_string, username + password)$, using $password$ as the HMAC key.

1.3 GetUser

This function first checks the HMAC, then checks if the password matches, finally returns the decrypted userdata.

1.4 StoreFile

This function first gets the user's FileInfo structure and checks the HMAC, then generates the a new random decryption key and nonce, key for HMAC checking, address for storing, a random address storing the number of appending actions, the random salt, and appends the five arrays with the corresponding content. (if there is a collision with the existing filename and the filename argument, it will change the nonce and restore the data with the previous key and address) Then it will calculate the HMAC of the encrypted data and store $Nonce + HMAC + encrypted\_content$ at the generated random address.

1.5 AppendFile

First get the FileInfo and check HMAC, then check if the file exists. The function works like this: for n'th file, we calculate the new file's address and key by calculating $hash(salt + origin)$ for n times. We store the number n at a random address. So for appending the file, we simply calculate the $hash(salt + origin\_address)$ and $hash(salt + origin\_key)$ for n times, encrypt the data and save that data and the nonce at the new address, then we store $n + 1$ at the address storing the number of appends conducted to the file. And for a file, we have a random address store encrypted n, using a random nonce and the same key for encrypting the file data, we store $Nonce + E(key, number)$ at that address, and the address is in the fileinfo structure.

1.6 LoadFile

This function first gets the FileInfo and checks HMAC, then finds the corresponding file by filename, if it exists, decrypt the file data, including the appended file.

1.7 ShareFile

First load the FileInfo and find the corresponding file by filename. Then load the recipient's public key, generate a random key $K1$ and encrypt $K1$ by recipient's public key. We use this key to encrypt the corresponding content of FileInfo, including the key for decrypting the file data, key for checking the HMAC, the address where the file is stored, the salt and the address of the number of appends conducted to the file (The elements in FileInfo structure except for the filename). We will generate and nonce and $AESCFB(K1, data)$. In the end, we generate a RSA sign on the plaintext data and return $RSA(Rec_{Pub}, K1)+AESCFB(K1, data)+Nonce+RSASign(Self_{Pri}, data)$.

1.8 RecieveFile

First split the data into four parts, use self private key to decrypt $K1$, use $K1$ and $Nonce$ to decrypt the data, add the data and the filename to the user's FileInfo so that the user can have access to that file. In the end, restore the FileInfo and the new generated HMAC for the structure.

1.9 RevokeFile

Remove the previous FileInfo at the structure, generate new address, key for decrypt, key for HMAC and address to store number. appent the new record, re-encrypt the data (decrypt the previous one first) and save it to the new address.

1.10 TestCase

We tested the init, storage, appending, sharing and revoking. We tested after Alice shared a file with Bob, if Alice updated the file, if Bob will know that. Besides, if Alice revoke the file, if Bob can load or append the file.

We also used the function *DatastoreGetMap* provided by userlib, and change data in datastore, see if the integrity error will occur. Also, if Bob's access has been revoked, and if he attemps to overwrite the data in the previous address, will Alice be affected.

2 Security Analysis

2.1 MITM attack - CFB oriented

If the attacker tried to change the data on the server, and tried to let us decrypt different answer, we can use HMAC to avoid that happens. For User, FileInfo structure, and also the file data they all have HMAC to guarantee the integrity, so that without knowing the user's password, the attacker can't change any bit on the server. Besides, we change the $Nonce$ every time, we have tested this by setting a hook in function *CFBEncryption* to confirm that the IV all different, and it works, so there is no way that the attacker can attack our CFB encryption.

2.2 MITM attack - against sharing

We signed the plain text by RSA signature, so if the attacker changes any bits in the message, the recipient can notice that. And since we signed the plain text instead of the cipher text, only recipient can decrypt the message, and only recipient can verify if the message came from Alice, Mallory is SOL.

2.3 Side-channel attack

We used the method encrypt then hmac to aviod Side-channel attack, also, we used *Equal* function provided by userlib which doesn't leak timing.