Our proposal is based on the idea of making it extremely hard for the malicious server admin (who can have access to the hard drive which contains the uploaded data) to 'extract' the secret information.

Our implementation is best understood by a self-designed TILE GAME. The Game is as follows:

- In a floor having n tiles, each tile has a coupon hidden behind it.
- Not all coupons are valid. You as an individual do not know which ones are valid/invalid.
- You do not know the exact no. of valid coupons.
- All the coupons are identically different. That is, by looking at two coupons, you cannot know any correlation between them.
- Only if you select all the valid coupons (how many, you don't know) and send them in a 'particular' sequence (you don't know the sequence as well, only the coupon company knows the sequence) will you win a grand prize!

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36
37	38	39	40	41	42	43	44	45
46	47	48	49	50	51	52	53	54

- Now, since you neither know the no. of valid coupons, nor the sequence, the only way for you to win a grand prize is to try all possible combinations. That is, not only you need to decide upon the no. of coupons and positions to chose, you also need to decide upon the sequence in which you will send your coupons.
- As it happens, the total no. of trials needed is

$$\mathrm{N} = \sum_{i=1}^{n} \binom{n^1}{i} * (i)! >> 2^n$$

- Thus, the no. of trials needed is exponentially hard. If n = 32^6, the no. of trial will be $2^{2^{30}}$ which is extremely huge!

Why Sequence is important in RSA?

- The way RSA works, we know that the encrypted message (ciphertext) is nothing but a very long integer. Thus, an integer such as 1150903589645796522569854456456454684354866974306586032068145789654

230368805421546455667680409653089461253648125348621538462153584762135847621 358476213521 may represent some cipher text (the original message may be 'hello world').

- Now suppose some guy breaks this long integer into chunks of 5 digits and rearranges them somehow randomly in some permutation 430651150945645035896457984354652256985457896645468669786032068145423036880542154645504096530894612536481253486676884621621535358476213 21352158476 then while decryption, this value may be either some garbage or (rarely) some other message, but not the original data. Thus, we need this integer in proper sequence while decryption.

- Thus, if we first encrypt the secret and then split it in chunks of fixed size f and randomly permute those chunks, then for decryption, we will not only need to collect all those chunks, but also arrange them in correct sequence before applying decryption. This method works also for the symmetric algorithms like ChaCha20Poly1305.

Implementation:

We try to simulate the above-mentioned TILE GAME as a networked file system in a folder in the hard drive of untrusted server. More specifically, we implement our own custom file system and a set of services to access it. Any secret uploaded to our custom file system is guaranteed to be secure even if the malicious attacker has capability to remove the hard drive and physically access the contents of the hard drive.

We classify secrets in two types:

- Secrets private to a user. These are only accessible by the user himself/herself.
- Secrets shared to a set of users. These secrets are accessible to all the users which are entitled by the author/owner of the secret.

These two types of secrets are handled differently. For the secret of first type, we provide only the guarantee of Confidentiality. For the secret of second type, we provide the guarantee of confidentiality as well as integrity- only the owner can modify the secret, not the other users.

Layout of Implementation:

Our implementation consists of a client-server architecture. The server side consists of a custom file system mounted on an untrusted server and set of two different processes: UserService and GroupService which deal with the secrets of first and second kind. Both processes are child processes and are created by the file system process during mounting.

The client code is run by individual users to access their files on our custom filesystem.

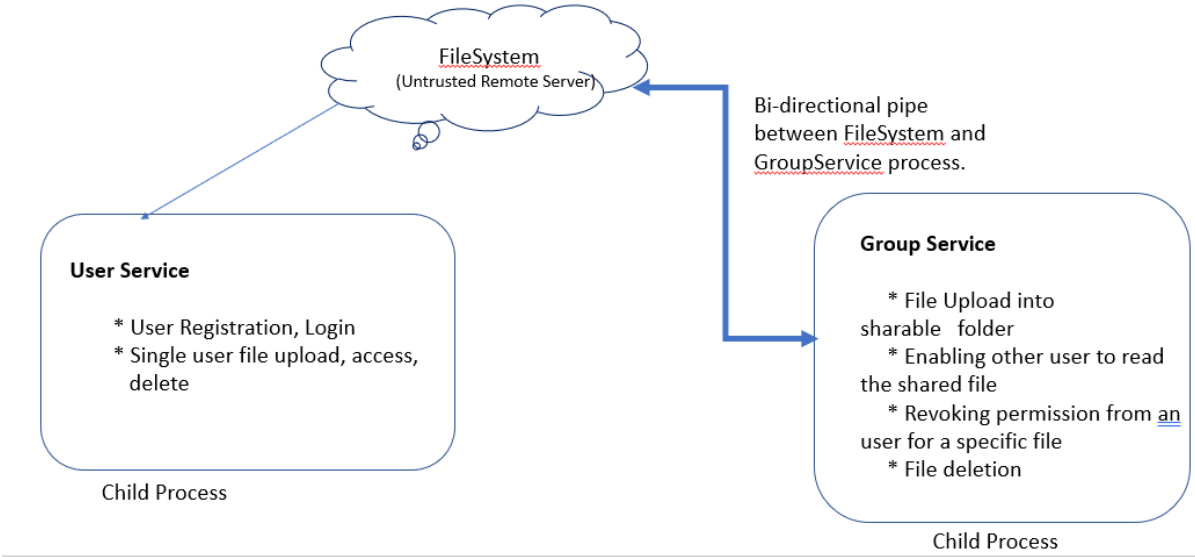

Fig: Layout of WrapCloud

- **File System:** Mounted as a folder in userspace. Corresponding to each registered user, a separate file directory structure. In the directory structure, there are 32^6 folders (locations or tiles) and corresponding to each folder, there is a file of size 128 bytes (coupons). All the files have encrypted content initialized.

  Shared folder serves as one-stop location for storing all the **secrets of second type** for all the users. Care is taken so that no user can access any location which he/she is not entitled to. Any access request to any location in the **shared** folder is trapped by our custom file system. The process making the request is then authenticated by the file system, in the same way authentication works for a client. Only the **GroupService** child process is allowed any access to the shared folder.

  The folders for individual users are named by their **userid,** which is a random number generated during that user's registration. This mapping prevents the filesystem (and by extension the untrusted server) to know someone's human-username. Please note that since all the folders contain only one binary file named 'file', the server cannot know the secret names as well.

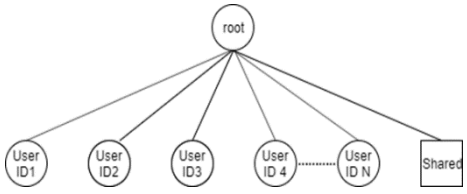

Fig: Per-user directory structure and a common shared folder inside the custom filesystem

- **UserService:** Started as an independent child process of the file system process during mount, it deals with all the folder and locations other than the **shared** folder. It deals with user registration, file upload, user login/logout, file access, etc.

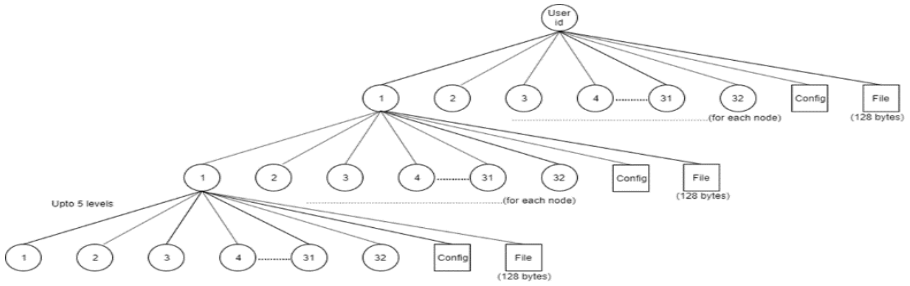

Fig3: Directory structure for one user. It contains 32^6 locations to store chunks of secret. Each 'file' named file is used to store the secret chunks.

- **GroupService:** Starts as an independent child process of the filesystem process during mount. It deals with only accesses related to **shared** folder. It is authenticated everytime an access to shared folder is made. It is the only process allowed to read/write to the shared folder. This way, the file system maintains the integrity of the data stored inside the **shared** folder.

  FileSystem process and **GroupService** process share a bidirectional pipe (potentially insecure) useful for communication between these two processes. The communication is important for authentication of GroupService process. Only by authentication, the FileSystem process ensures that only **GroupService** process can access the files within the shared folder.

WORKING

- Initialize:

The file system is mounted in the user space of the server machine. This can be done using certain framework libraries like FUSE library. The mountpoint can be any folder. During mount, the filesystem process is started with two child processes – UserService and GroupService using os.fork() method of python. The UserService process creates the directory structure inside the mount point of FileSystem process. A bidirectional pipe (insecure) is maintained between the FileSystem process and the GroupService process

Both UserService process and GroupService processes are RPC server processes, which listen for client connections.

Secrets of first type (Single User services):

- *User registration:*

:

Following steps are performed during user registration:

1. Human -> Client : Provides desired username, password

2. Client : Compute Hash(password)
3. Client -> UserService : Register(username, hash(password))
4. UserService: Generate random and unique userid u for this request.
5. UserService: Setup directory structure (fig 3) for this userid
6. UserService -> GroupService: Register(userid, hash(password))
7. GroupService: Store (userid, hash(password)) in login table.
8. UserService - > Client : return userid u
9. Client: Generate Epriv, Epub (RSA) for userid
10. Client: Store (userid, Epriv, Epub) in separate files locally. Assume that Epriv cannot be accessed by any attacker.

File Upload

The client-side maintains a local file which lists the (filename, fileid) pairs. "fileid" is a random and unique number specific to each filename of an uploaded file. It also maintains a list of nodes occupied by the current user in a file called "NodeList". This file is a 2D bit-matrix and is essentially encrypted using the user's public key. Each bit in this file's content represent a node/folder in the directory structure of the user. If the node is occupied, the corresponding bit is set.

The following steps take place:

1. Human -> Client : provide (path of secret file to be uploaded, username, password)
2. Client : f = read contents of file-to-be-uploaded
3. Client : Encrypt f. E = Epub(f). Load Epriv from local storage.
4. Client: Generate f_seq according to Algo 1.
5. Client - > US: Upload(E,f_seq,username, hash(password))
6. US: Authenticate the user. Find locations in directory structure of this user corresponding to the entries in the f_seq list
7. US: Divide E into chunks of size 127 bytes.
8. US: Store each chunk to the location corresponding to entries in f_seq
9. US -> Client: Return True if everyting done successfully. Otherwise, return False
10. Client : Save (filename, fileid, no.of chunks) locally in a separate file named 'fileinfo'. fileid returned by Algo 1
11. Client: Update "Nodelist"

File Access (Read)

The file access makes use of the (filename, fileid, no.of chunks) file. The steps are:

1. Human -> Client: Provides filename for the file to be accessed, username, password
2. Client: Read (filename, fileid, no. of chunks) file. Also read Epriv, userid files
3. Client: Generate f_seq list using Algo 1 (fileid, no. of chunks)
4. Client -> US: Read(username, hash(password), f_seq)
5. US: Authenticate the user. Read locations corresponding to each entry of f_seq
6. US: F = concatenated contents of each locations
7. US -> client : Return F. Note that F is still encrypted using Epub of the user.

File Delete

The steps are:

1. Human -> client: Provide filename to be deleted, username, password
2. Client: Read (filename, fileid, no. of chunks) file
3. Client: Read Epriv for this user.
4. Client: Read userid for this user from local storage.
5. Client: Generate f_seq using Algo 1
6. Client -> US: Delete(username, hash(password), f_seq)
7. US: Write encrypted garbage in locations corresponding to each entry of f_seq

Algo 1

Generate_file_sequence(filepath, priv_key):

```
    file_contents = read(filepath)
    N = len(file_contents)/127
    ifileid = RandomInt()
    File_seq = []
    A = ifileid

    For i in range(N):
V = Encrypt(priv_key, A) % pow(32,6) + 1
        < match if V already exists in Nodelist for this user>
        If no match found
        File_seq.append[V]
        A = V

        If exists (then conflict has occurred):
            ifileid = RandomInt()  # generate new ifileid
            i = 0
            Continue
    Return [ifileid, File_seq]
```

- # Secrets of second type (Shared Secrets)

The start of file operations for these type of secrets starts with getting ‘owner privilege’ for the client. After the client has obtained owner privileges, he/she can perform file upload, delete, share with other users. File access does not require owner privileges.

The owner privilege is just a way for the GS process to ensure the secure upload of file contents to file system over an untrusted medium between the GS and client. For every file upload request, the client must obtain owner privilege.

Obtain Privilege:

Used to obtain a secret key (shared) between GroupService (GS) and client. This secret key is used to encrypt the secret and send it for upload. Each file has its own secret key.

The secret key is shared using the Diffie Hellman protocol.

1. Client -> GS: Admin(userid, hash(password))
2. GS: Perform user authentication
3. GS <-> Client: Perform Diffie Hellman to share symmetric secret key
4. GS -> Send “nonce”. Nonce is necessary because the file-to-be-uploaded is encrypted using ChaCha20Poly1305 symmetric encryption, which needs a nonce.
5. GS: save (owner’s userid, nonce, shared secret key) in a separate file inside the **shared** folder.

File Upload

The GroupService(GS) maintains a no. of files for ensuring proper working. These files are (filename, list of users allowed) – stores info about which users can access any given file, (filename, owner’s userid) – stores info about the owner of a file, (filename, fileid, nonce, secret, no. of chunks) – stores info about the fileid, nonce, secret, no. of chunks of the file, “Nodelist” file – which contains a 2D bit matrix corresponding to all the nodes available inside the shared folder’s directory structure. A bit is set only if corresponding node is occupied. All these files are stored inside the shared folder.

Note that these values are important for decryption of the file, so these files must not be read by any other process running on the untrusted server. The FileSystem process ensures this by authenticating the GS process every time anything is accessed within the **shared** folder.

1. Human owner -> client: Provides the filepath, username, password
2. Client : (shared key, nonce) = ObtainPrivilege(userid, hash(password)). The userid is read by the client from local storage.
3. Client: Read the file as F. E = Eshared_key(F,nonce)
4. Client - > GS: Upload(E, owner’s userid, hash(password))
5. GS: Authenticate the owner.
6. GS: Break E into chunks of 127 B. Calculate (fileid, f_seq) for E by algo 1
7. GS: Store each chunk of E into locations corresponding to each entry of f_seq.
8. GS: Save (filename, owner’s userid) , (filename, fileid, nonce, secret, no. of chunks), (filename, list of users allowed) in separate files in **shared** folder. Initially, the list of users allowed will contain only one user – the owner himself.
9. GS: update the **Nodelist** of the **shared** folder corresponding to entries of f_seq list.

File Access (Read)

The medium between the client and the GroupService may be untrusted. The GroupService first checks whether the requesting user is allowed access or not on that file. Then, it uses DiffieHellman for sharing a secret key used for transmission of data between the user and GroupService. This secret key is different from the one used for encrypting the file by the owner.

1. Human -> client: Read(filename, username, password)
2. Client < - > GS: (shared key S, nonce n) = ObtainPrivilege(userid, password)
3. Client -> GS: Read (filename, userid, hash(password))
4. GS: Authenticate the user, read (filename, list of allowed userid) to check if this user is allowed to access this file.
5. GS: Only if this user is allowed, (id,n1,s) = read(filename, fileid, nonce, secret, no. of chunks). If the user is not allowed, return Null.
6. GS: Generate fil_seq according to Algo 1
7. GS: read the file contents of each “file” in the location corresponding to each entry in fil_seq list.
8. GS: E = Concatenate all the file contents. F = Decrypt(n1,s).
9. GS: E = Encrypt(n,S)
10. GS - > Client: return E

Share File

The admin provides his userid (owner user id) and the userid of the user with whom he wants to share the files with.

1. Human - > Client: filename, owner userid, password, other userid
2. Client - > GS: Share(filename, owner userid, hash(password), other userid)
3. GS: Authenticate owner
4. GS: Read(filename, owner userid) to verify that the owner is the one who uploaded this file.
5. GS: Update (filename: list of allowed users) to include other userid in the list of allowed users for this file

Revoke Permission:

Allows an owner to revoke the permission to access a file from other users.

1. Human -> Client: filename, username, password, other userid
2. Client -> GS: Revoke(filename, owner userid, hash(password), other userid)
3. GS: Authenticate the owner
4. GS: Read (filename, owner userid) to verify that the owner is the one who uploaded this file
5. GS: update(filename, list of allowed users) to remove the other userid from the list of allowed users.

Please note that every access to the shared folder is trapped by our custom filesystem. The filesystem then authenticates the GS process using the bidirectional pipe for communication. All the important files needed for maintaining confidentiality and integrity in case of the shared secrets and the login info is stored within the shared folder.

The authentication between the GS and the FileSystem process occurs as follows:

1. During mount, the FileSystem forks() the GS process and starts a bidirectional pipe with this child process.

2. After the GS process becomes active, a secret key is shared between the two using DiffieHellman protocol. The keys are sent across the bidirectional pipe.
3. Whenever there is an access to the shared folder, the filesystem sends a random nonce across the pipe.
4. If GS was the process which wanted to access the shared folder, it will be expecting a nonce on the pipe. It reads the nonce value and encrypts it using the shared key.
5. It then sends the encrypted value across the pipe. The FileSystem process will be expecting some content on the shared pipe. It decrypts the value and matches the nonce. If the nonce matches, it allows the access.
6. If it were some other process, there would be no shared pipe, it will not be able to read the nonce or use shared key and thus it will not get access.

In this way, the FileSystem process ensures the only the GroupService process accesses the Shared folder. It thus maintains the integrity of data.