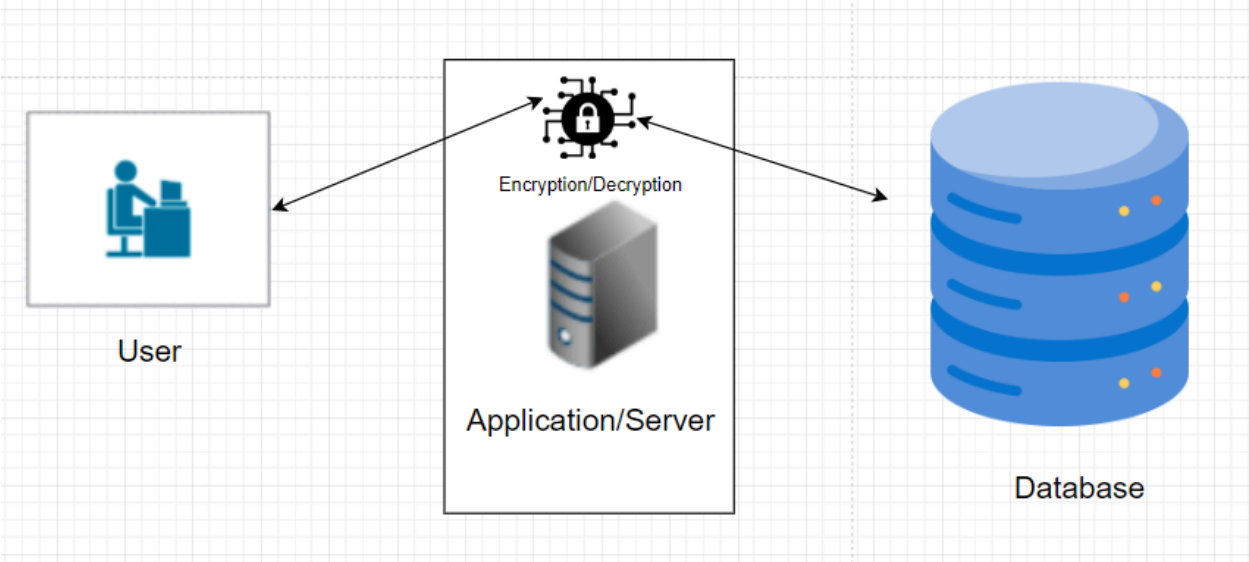
# ECE 422 Project 2 Final Report

## April 8, 2024


| Student Name | ID |
| --- | --- |
| Francis Sepnio Jr. | 1617559 |
| Defrim Binakaj | 1621841 |
| Reynel Guerra-Pavez | 1616041 |

# High-level Architectural Overview




We aimed to maintain a high-level architecture that prioritizes simplicity and minimizes individual components while ensuring robust encryption and decryption processes to reduce potential errors and program complexity. Reviewing the diagram, our architecture consists of three primary components: the user running our program, the program/application itself, and the database.

Regarding the application, our focus centered on implementing a strictly server-side approach, where encryption and decryption keys are stored. This approach helps secure user information and maintain confidentiality. Our interaction with user data is confined to handling encrypted inputs, while encrypting and securely storing any necessary information in the database. The server retrieves required data from the database and compares the encrypted input with stored data to verify matches.

# Detailed Design

## Design of Authentication, Encryption, Metadata, and Permissions

### Authenticated and Secure Communication

To ensure authenticated and secure communication within our application, we've integrated a component responsible for encrypting and decrypting data within the server

environment. All user information undergoes encryption or hashing before storage in the database or during handling within our system. For instance, sensitive data such as a user's username and password are never stored as plaintext. Instead, we encrypt any input before processing it within our system and securely store the encrypted data. When a user initiates a login attempt, we retrieve their input, encrypt the username, and hash the password. Subsequently, we compare the hashed password with the encrypted username in our database. A successful login occurs when the hashed password matches the password stored in the database for the encrypted username, ensuring a robust authentication process.

### Data Encryption

We use AES symmetric encryption for encrypting our data. This means we only need one key to encrypt/decrypt our data. When a user runs the program, the key is stored in memory within our application. We chose to implement it this way over storing it in a file or in the database so that no external user would be able to access it. This ensures no direct exposure of the key to any external user. This is assuming that the user has no direct access to the source code, which, considering our file system as a separate entity and pretending like our project repository doesn't exist, felt like a reasonable abstraction to make. Additionally, we've enhanced the authenticity of our encryption by utilizing HMAC in our encryption. By combining AES encryption with HMAC, we can further verify the integrity of our encrypted data, mitigating the risk of data tampering and unauthorized access. For the packages themselves, we chose to use the Python Cryptography library for its AES encryption algorithm and HMAC. We selected this library as it is very popular and well-documented, which helped us troubleshoot during development. We also used hashlib's SHA-256 hashing algorithm to hash our users' passwords in the database, which was also selected for its popularity and ease of use. Finally, we used the binascii library to convert our encrypted data into hexadecimal format, making it easy to store in the database without worrying about conflict due to characters like slashes messing with things like our file and directory paths.

### Metadata Design

In this section, we will discuss the user's metadata, the file's metadata, and the directory metadata. In terms of the user metadata, the important data that we want to keep track of is a user's unique user ID, username, password, and the groups that they belong to. The user ID is an automatically incrementing integer which SQLite creates for us every time a new entry is added to the "users" table. The username is what the user inputs to the SFS when registering a

new account. This is used for uniquely identifying the user when running SQL queries. We also keep track of the user's groups in the database. This is for the purpose of permission checking, for cases where the user wishes to access files or directories that their group has permissions to, for example. The username and group metadata is encrypted using hexadecimal encryption as described in the section above and stored in the database. The password is also what the user inputs to the SFS during registration, and this password is hashed using SHA-256 hashing and stored in the database. During user authentication, the password input is hashed and then checked with the existing hashed password in the database for a match. However, like all hashing algorithms, this is prone to hash collision, though unlikely with SHA-256.

In terms of file metadata, we keep track of the unique file ID, file name, owner ID, file path, file permissions, and file content. Like the user ID, the file ID is an automatically incrementing integer which SQLite creates for us every time a new entry is added to the "files" table. The file's owner ID references the unique user ID from the "users" table, and this is kept track of for the purpose of ensuring that a user owns the file before being able to change the permissions of it. The file name and file content are encrypted on database and on disk, ensuring that external users not part of the SFS cannot view them. These are kept track of for the important reason of maintaining file integrity, and checking if a user's files have been tampered with by a user outside of the SFS. We also keep track of the file permissions, which are encrypted on the database, and are used for checking if a user has access to the file. The file path is the most important piece of metadata for a file, and this is because this is the unique identifier we use for querying our "files" table. All files on the SFS will have a unique file path. This path is also encrypted on the database to ensure security, though it is important to note that things like

The directory metadata is very similar to the file metadata, except we do not have to keep track of a directory's contents. We keep the directory metadata for the same reasons as those for a file, so the justification is the same as above.

The design decisions we have made for the metadata that we keep track of as well as the methods we use to store them securely on the database maintain a very secure file system.

### Multi-user Access Management

Our group implemented owner/group/other permissions of files and directories by having permissions be one of 3 different states: "owner," "all," and group. When a directory or file is created, it is given the "owner" permission by default, meaning that the owner of the file is the only person who is able to view the decrypted file name, see the decrypted contents, or

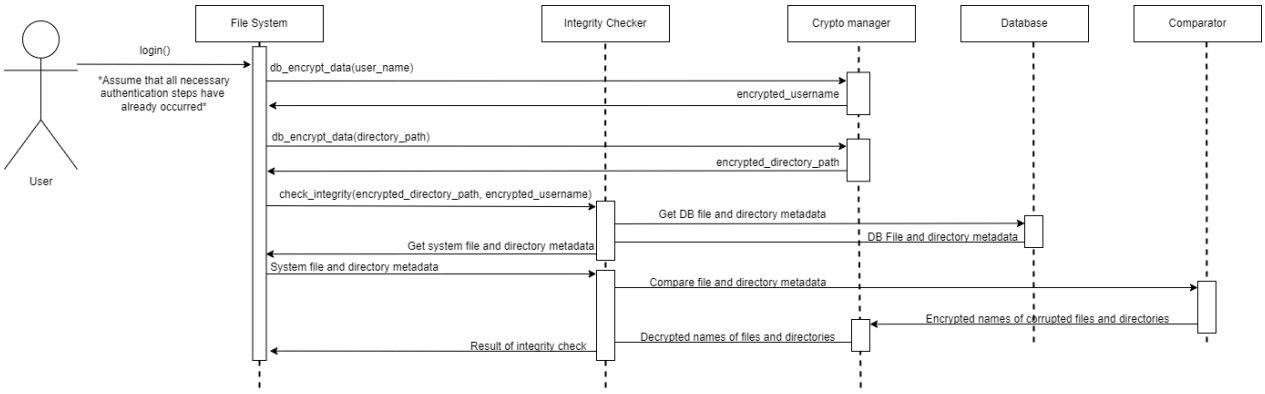
read/write the file in any way. When the owner of the file runs the `chmod` command, they are able to change the permission of the file or directory. Upon running the `chmod` command, the user is presented a menu and can select which of the permission settings to enable. If the owner selects “all,” this means that everybody who is a registered user of the SFS is able to read/write and access any file or directory with this permission. The last permission that owners are able to set on their files is the group permission. For this, the user inputs a comma-separated list of groups that they wish to grant access to. It’s important to note that the owner can only grant permissions to groups that the owner themselves are a part of.

## Technologies / Methodologies / Tools

For this project, we used **Python** as the programming language. This is because it has a great library, called **os**, for facilitating operating system commands and processes which are the backbone of the project’s functionality. We connected the code to a **SQLite** database in order to store everything performed in our program, since it is easily stored as a local file in the project directory, has simple python integration with the **sqlite3** library, and it has great relational data management to help us dictate file sharing permissions among users. **sha256** (with **hashlib**) was used for password hashing to ensure secure storage. The **Cryptography** library was used for encrypting and decrypting user, file, and directory metadata using its AES asymmetric encryption algorithms. We also used its HMAC algorithm for authenticating our encryption. The **binascii** library was used for converting our encrypted data to a hexadecimal representation and vice-versa so that our data was easy to store in the database. **getpass** was used to hide password input, ensuring that the user’s credentials are not exposed during login and registration. **GitHub Projects** was used for managing the project's progress using its ticket system. **Git** was used for version control because of simultaneous development, allowing us to fix merge conflicts and finish the program. We also used pair programming very often throughout the development of this project, which helped us find errors in code before running it, which greatly improved the productivity of our group.

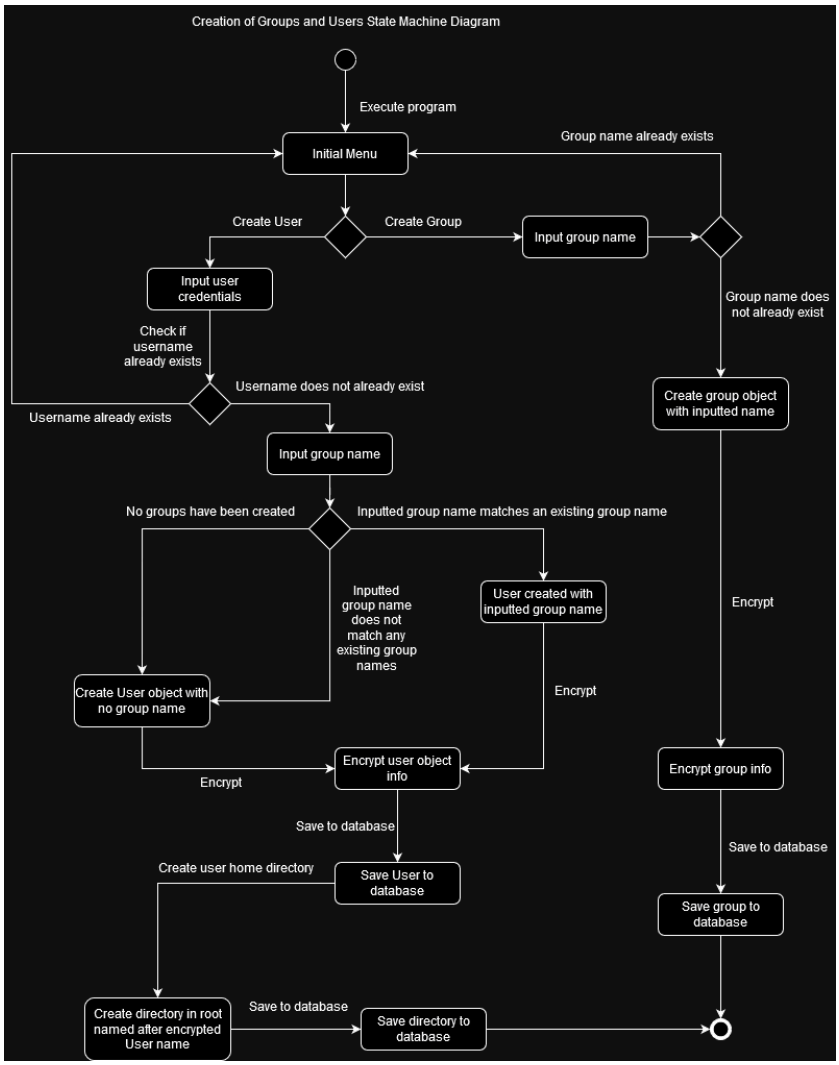
## UML Sequence Diagram

The UML Sequence Diagram below models the requirement “SFS should detect corruption of file/directory names and contents.” This shows the general sequence of events and they are not exactly 1:1 to the function calls of our program as our program didn’t exactly follow an object-oriented implementation.




For the purposes of conciseness, it is assumed that in this diagram, the necessary function calls to authenticate the user’s login have already occurred, and this is the sequence of events that occur after the user successfully logs into the SFS. First, upon login, the user’s username and directory path to their home directory is encrypted. Then, these encrypted data values are sent to the integrity checker which queries the database for the file and directory metadata, using the username and path as unique identifiers. File metadata would consist of the file names and file contents. Directory metadata would consist of the directory names. Upon retrieval of this database metadata, the integrity checker then goes to the file system itself which performs the necessary commands to retrieve the file and directory metadata from the disk. The integrity checker then sends it to a comparator to compare the disk data and the database data to find any discrepancies. Since it was working with encrypted data this entire time, the crypto manager decrypts the corrupted file/directory names, if any, and sends the resulting information to the file system to be printed to the user.

## State Machine Diagram




The state machine diagram illustrates the different states that a user will experience during requirement 1 of the SFS specification - creation of groups and users. Firstly, when the program starts, the user is taken to the “Initial Menu” state where they have a number of options to pick from. Depending on which option is selected, it will take them to that corresponding state. The user can choose to register an account, in other words, create a user, which will take them to the “Create User” state, or create a group, taking them to the “Create Group” state. When creating a user, the input name of the user will be checked to verify if a user already exists with that name; if so, the user is not created.

Assuming the inputted user name is unique, the user then selects a group for the user. If there are no groups or the user inputs a non-existing group name, the user is created without a group. If a valid group name is inputted, the user is assigned to the inputted group name. Afterwards, the user that was created has its information encrypted and then stored in the database. Finally, a home directory of the user is created with the user’s name, while also being stored in the database (since the directory is created with the encrypted user’s name, no need to re-encrypt). On the other hand, the group path is much simpler - the user inputs a group name; if the group already exists, it is not created, but if the inputted group name is unique, then a group is created, encrypted, and stored in the database. Both states, once successfully completed, will return to the “Initial Menu” state.

# Deployment Instructions

In order to run the SFS, you want to ensure that Python is installed on your system and that you are using a Unix system.To install all of the dependencies:

1. Run `git clone https://github.com/PerrieFanClub-ECE422/Project-2`
2. `cd` into the Project-2 folder
3. Run `pip3 install -r requirements.txt`.

Now, you should be ready to run the SFS by running `python3 main.py`. This will create a "root" directory in the same directory as the project files. A database file called "sfs.db" is also created.

# User Guide

When a user enters the SFS, they are given four options:

1. Login: A user logs in with their credentials if they already registered an account with the SFS.
2. Register: A user can register a new account by entering a unique name, a password, and joining a pre-existing group (optional). Their credentials are encrypted in the database.
3. Create Group: A user can create a new group if it doesn't already exist in the database.
4. Exit: Logs the user out of the SFS and exits the program.

Upon authentication to the SFS, a user will have a directory created for them. A user is only able to access directories and files for which they have permissions to.

Our SFS supports the following commands:

| Command | Description |
|---|---|
| `cd [filename]` | Moves into the directory specified by filename. |
| `pwd` | Prints the current directory. |
| `ls` | Displays file and directory names in current directory. Names are encrypted if the user does not have access to them. |

| | |
|---|---|
| `touch [filename]` | Creates a new file in the current directory with name specified by filename. |
| `rm [filename]` | Deletes a file specified by filename. |
| `mkdir [directoryname]` | Makes a new directory specified by directoryname. |
| `cat [filename]` | Prints the contents of the file specified by filename. |
| `echo [filename] [content]` | Writes new content to the file |
| `mv [filename] [newname]` | Renames the file specified by filename to newname. |
| `chmod [-f,-d] [target]` | Changes the permission settings of a target directory or file, denoted by -f or -d. Only the owner of the file or directory may change the permission settings. |
| `cmds` | Prints a list of commands to the terminal. |
| `exit` | Exits the SFS. |

## Conclusion

In conclusion, our group was successfully able to implement a functional and secure file system using authentication and encryption methods learned in class. We familiarized ourselves with various tools and technologies such as SQLite for databases and the Cryptography Python library for AES encryption. Like the Unix file system, our SFS allowed creation of groups and users, directory support and a per-user home directory, as well as file operations like creation, deletion, reading, writing, and renaming of files. Our SFS authenticated users before accessing the system, and encrypted all communication between the users and the SFS. By encrypting file names and content on disk, and notifying a user if their files were tampered with by an external user, we were able to maintain file integrity. Through the implementation of permissions and user groups, we were able to introduce file sharing and maintain confidentiality of information. Our group gained valuable knowledge and hands-on experience with authentication and encryption while working on this project

# References:

- Project 2 PDF on eClass
- Demo Marking Guide on eClass
- Project 2 FAQ sheet on eClass

For encryption and decryption with the Cryptography library:

- https://cryptography.io/en/latest/hazmat/primitives/mac/hmac/
- https://cryptography.io/en/latest/hazmat/primitives/symmetric-encryption/
- https://cryptography.io/en/latest/hazmat/primitives/padding/
- https://github.com/ly4k/Certipy/issues/31
- https://docs.python.org/3/library/binascii.html
- https://www.educative.io/answers/what-is-binasciihexlify-in-python

SQLite3:

- https://docs.python.org/3/library/sqlite3.html#how-to-use-placeholders-to-bind-values-in-sql-queries
- https://stackoverflow.com/questions/25371636/how-to-get-sqlite-result-error-codes-in-python
- https://stackoverflow.com/questions/36816640/how-to-handled-sqlite-errors-such-as-has-no-column-named-that-are-not-raised-a

Other Libraries:

- https://docs.python.org/3/library/os.html
- https://docs.python.org/3/library/hashlib.html
- https://docs.python.org/3/library/getpass.html