Team 22
Brian VerVaet, Max Molnar, Sam Spencer, Wyatt Dahlenburg, Zach Kent

INDEX

PURPOSE

Many types of devices use Flash for persistent storage. In situations where power fluctuates (batteries running out, power outages, pulling the plug), the integrity of these Flash- based file systems can be compromised, rendering the device degraded or useless. YFFS is a filesystem designed to avoid these hazards and be used as a secure, dependable storage medium on any flash device. YFFS provides stable storage despite power fluctuations by implementing a transaction-based file system. This type of file system is designed to leave the structure of directory, files, and metadata in a sound state. Additionally, this design is able to increase the chances of successful recovery in case of a hard failure. While there are transaction-based file systems that accomplish the task of flash stability, no such file system exists today that also provides an interface for user-defined encryption.

FUNCTIONAL REQUIREMENTS

1. YFFS should handle files.

- As a user...
  - I would like to be able to add files to the filesystem
  - I would like to be able to remove files from the filesystem
  - I would like to be able to read files in the filesystem
  - I would like to be able to list all of the files in the filesystem
- As a developer...
  - I would like a simple set of commands for the filesystem
- As an admin…
  - I would like to maintain permissions of files

2. YFFS should handle encryption and obfuscation.

- As a user…
  - I would like my files to be secure
- As a developer…
  - I would like an API to assist with the inclusion of encryption plug-ins
- As an admin…
  - I would like to define encryption for all write and read operations to the filesystem
  - I would like to define a cipher for all filenames to be obfuscated

3. YFFS should be fault tolerant.
   - As a user…
     - I would like dependable data in the event of power failure
   - As an admin…
     - I would like an atomic file system

4. YFFS should work on an embedded machine.
   - As a user...
     - I would like to be able to create a filesystem
     - I would like to be able to mount a filesystem
     - I would like to be able to unmount a filesystem
     - I would like to use this file system on embedded devices
     - I would like complete documentation of the filesystem tools

NON-FUNCTIONAL REQUIREMENTS

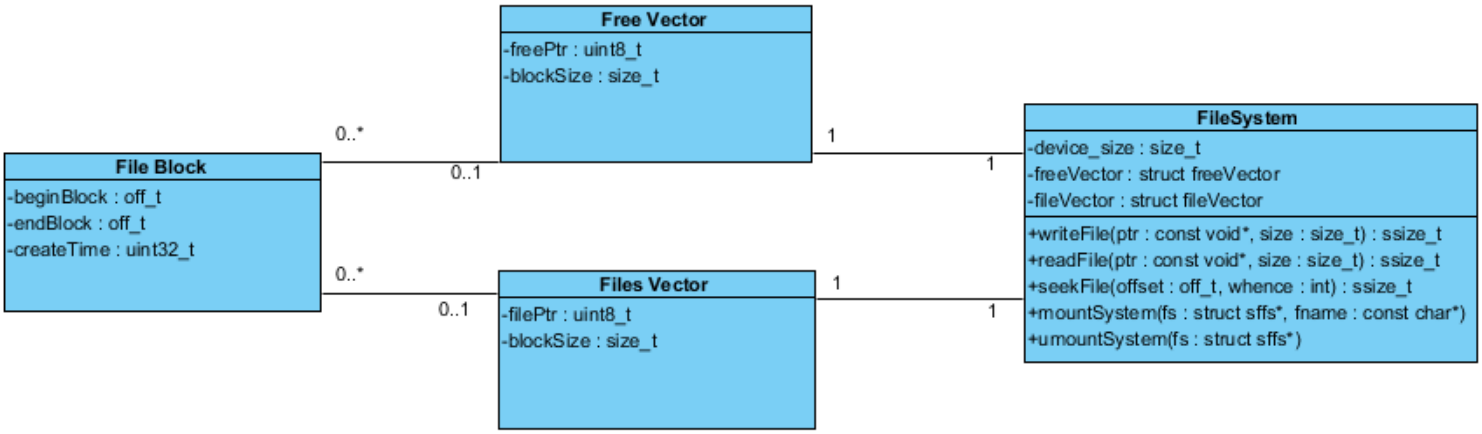
1. YFFS should be easy to use.
   - We must be able to implement it on many different devices
     - Should work on NAND and NOR hardware devices
   - The encryption interface should work quickly
     - Users do not want to wait a long time for every interaction
   - Interface should be user friendly
     - Commands should be easy to understand
   - Documentation will be efficient and concise

2. YFFS should work directly with Linux.
   - The system will mount directly to Linux
   - The system will be functional with Linux utilities
     - Mkfs, mount, unmount, etc.
   - The system will mount to embedded devices

DESIGN OUTLINE

YFFS provides a filesystem to users that ensures data is not corrupted. Our project follows the transaction processing model, where information processing is divided into individual, indivisible operations called transactions. YFFS will distinctively succeed or fail at data storage. Users will be able define an encryption algorithm for all write operations, and a cipher for all filenames to be obfuscated.

- The Filesystem
  - The filesystem will be fault-tolerant; i.e. Data will be transferred fully or it will result in an error
  - The filesystem will be atomic
- The Encryption Interface
  - Users will be able to encrypt files through standard encryption libraries or custom implementations
  - The user will be able to switch encryption interfaces
  - Decryption will be presented in a user friendly manner to show progress
- The Filename Obfuscation Interface
  - File names will be hidden from everyone except the designated user
  - File names will be stored through a cipher function



DESIGN ISSUES

What Open source file system should we use?

- XtreamFS
- SFFS

We chose to go with SFFS because we felt it was a more lightweight file system. It has less code to understand and it better fit our needs and requirements.

How should our filesystem handle encryption?

- Provide APIs for users to work with
- Provide our own collection of encryption libraries

We chose to provide APIs that users could use to integrate their own encryption needs. Adding encryption libraries would make our file system too bulky, and we also feel like adding encryption libraries is not feasible in the given time frame. Having APIs available also gives our file system a lot more flexibility.

Should YFFS work on an embedded system

- Work on an embedded system
- Will not work on an embedded system

We want our file system to be able to work on an embedded system. Since a major part of this project is creating a lightweight system, it makes sense that system should work in an embedded environment.

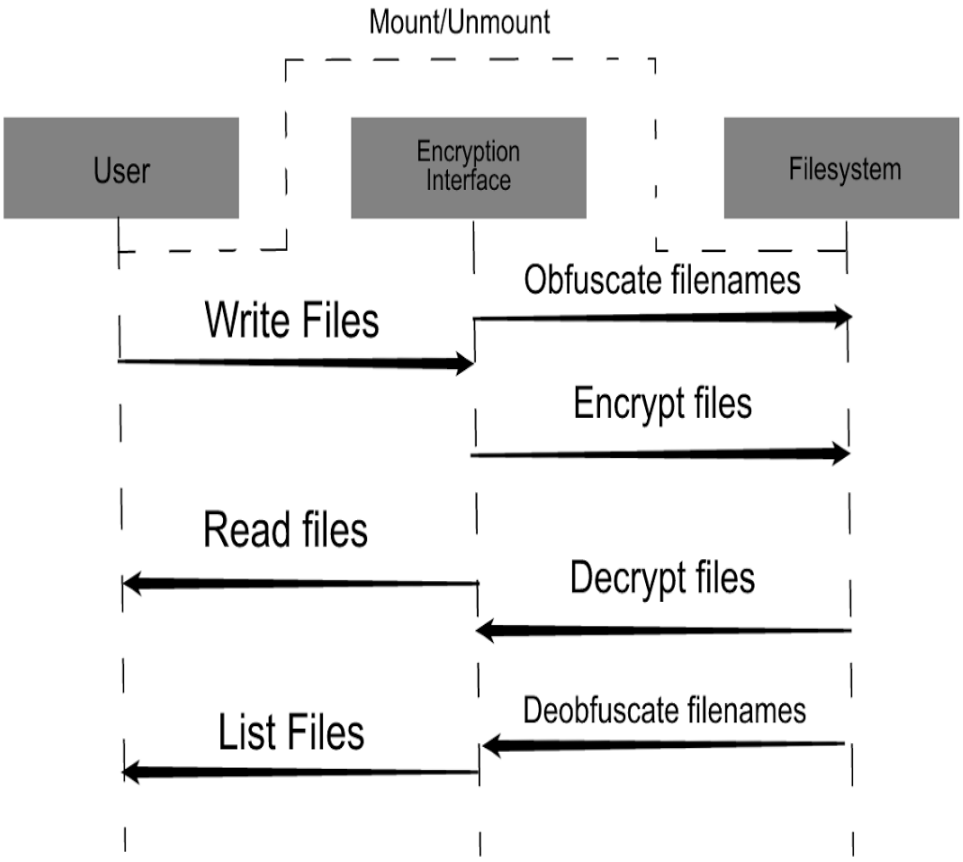
How should YFFS handle fault tolerance

- Distributed file system
- Atomic read/write functions

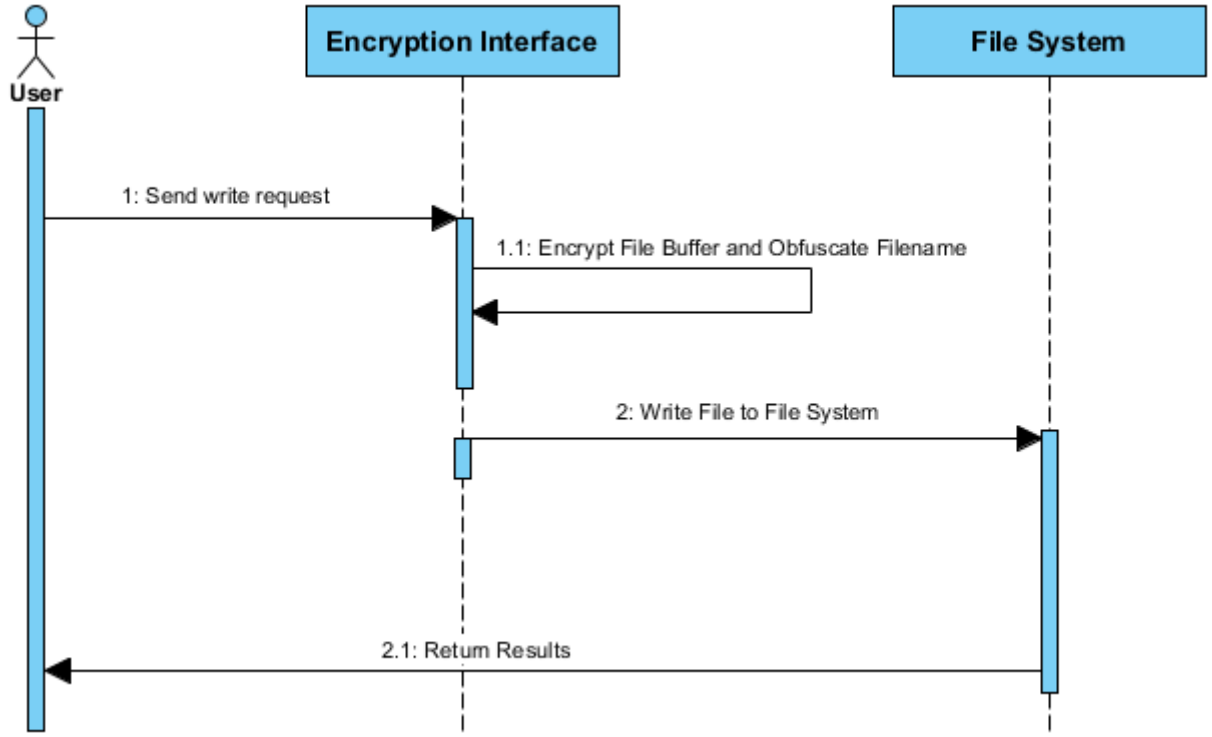
Atomic read/write functions are a more relevant choice for our file system. They allow it to be much more lightweight than a distributed file system. Also, having a distributed file system reduces security since those files are just sitting on a server. By choosing atomic read/write functions we guarantee we maintain fault tolerance without losing a lot of security.

Sequence Diagram

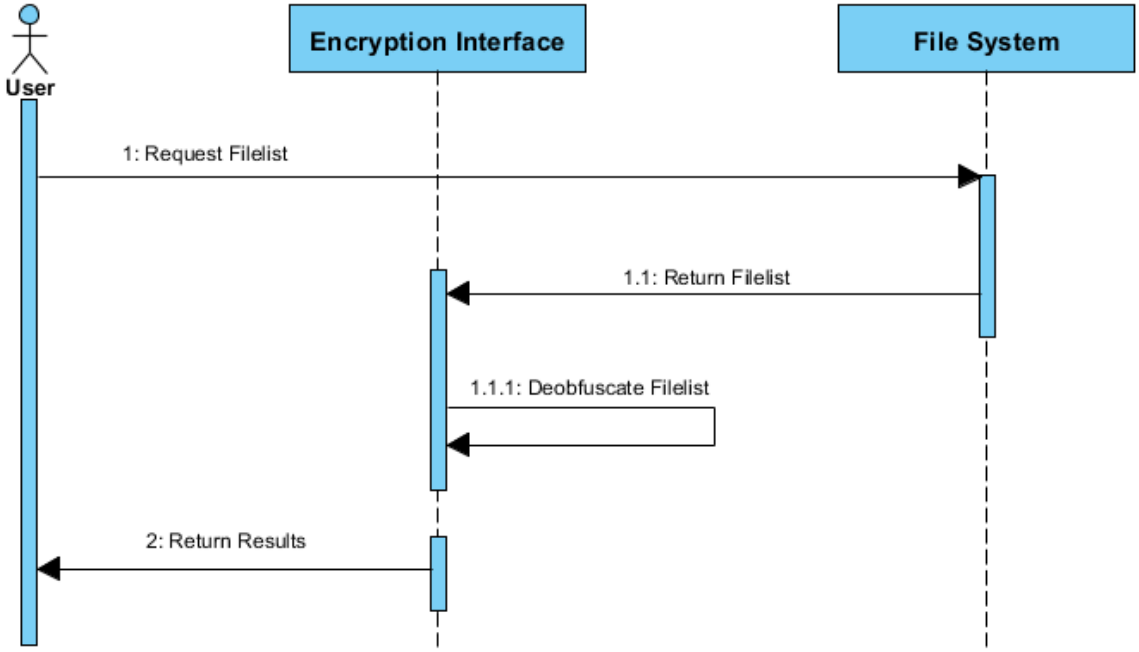
The sequence diagram shows the layout for what a user has access to. The user is allowed to make three transactions: read, write, and list files. The filesystem will obfuscate the names of the files and encrypt them when they are added. When the files are queried they will be decrypted. If a user wants to list, the filenames must be deobfuscated. The user will have full control of the filesystem and will not need to know the back-end.



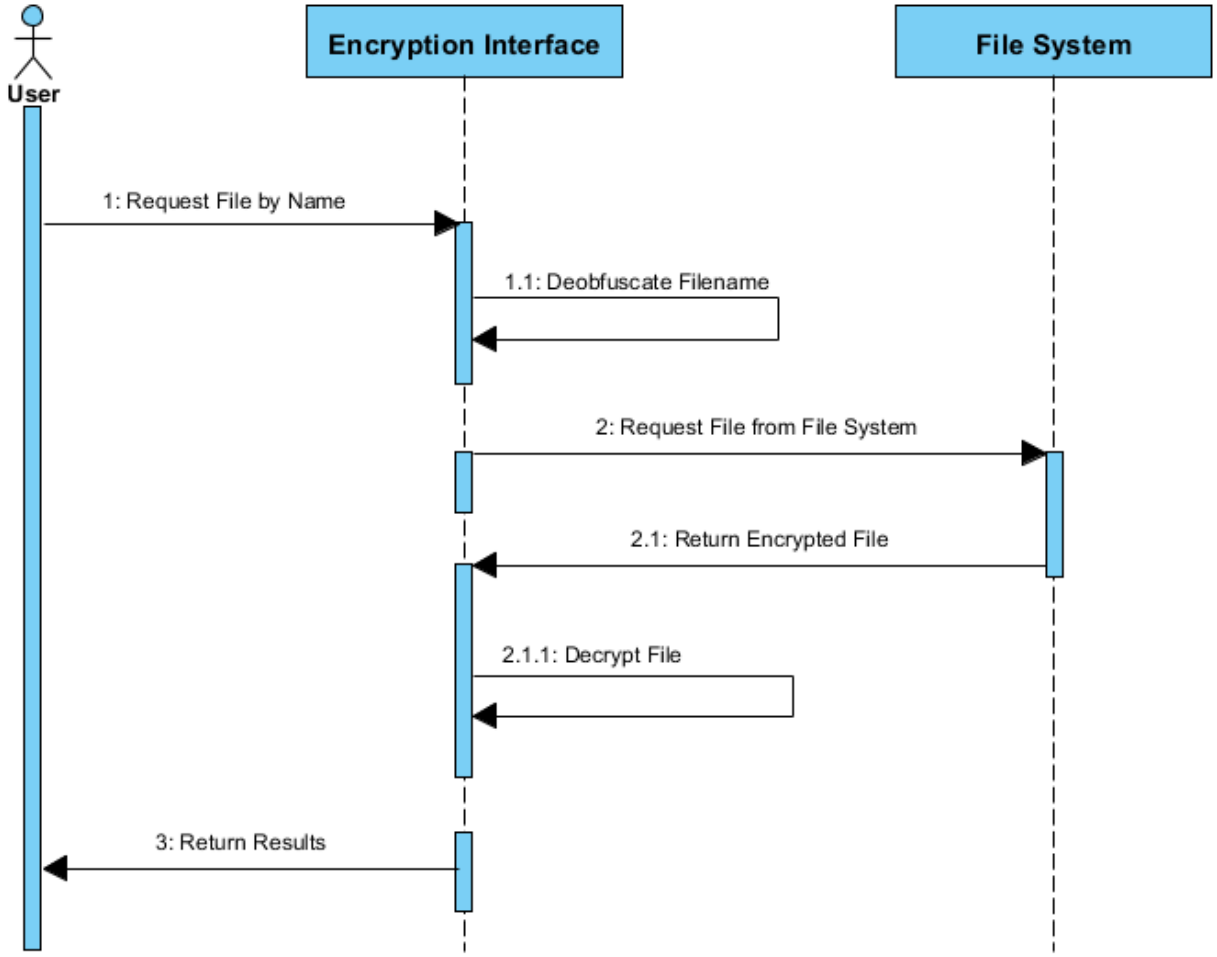
Write Sequence



List Sequence

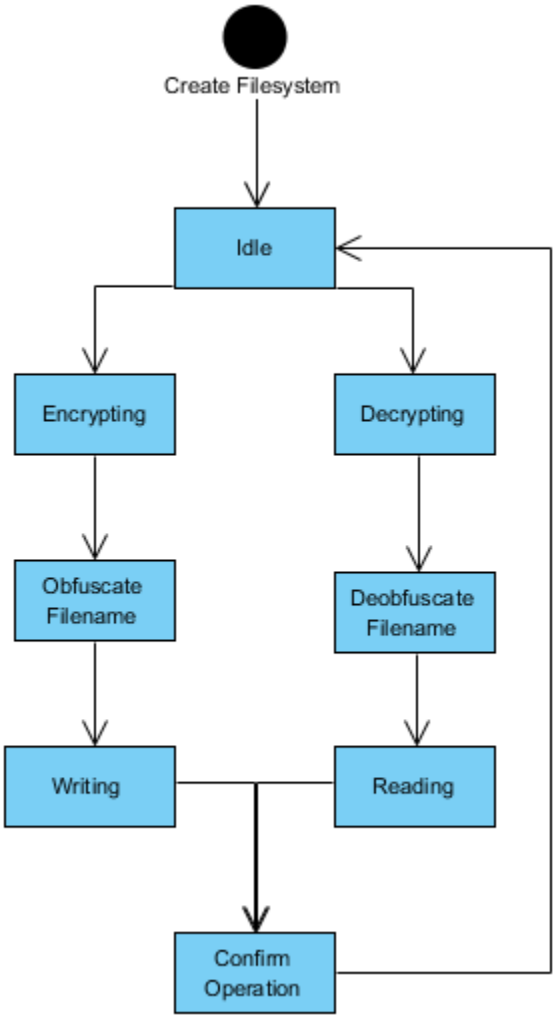


Read Sequence



State Diagram

Our file system will have two main groups of states; reading and writing. In order to write files, the data being store must first be encrypted. To be able to read these encrypted files, the filesystem must first decrypt them. File names must also be obfuscated when they are written, and deobfuscated if the user wishes to view them as a list. The confirmation step is added to maintain fault tolerance throughout the entire process. If it is not reached, the attempted change will not be completed. Once the operation is confirmed to have completed, the system goes back to an idle state waiting for more commands.



API Design

void* encryptBlock	void* orgBlock, int blockSize
void* decryptBlock	void* orgBlock, int blockSize
char* obFilename	char* filename
char* deobFilename	char* filename