Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

Task:

The purpose of this project is to implement code that will allow the encryption and decryption of files through the command line (such as when files are read from or written to). The encryption / decryption is done with AES.

Basic Terms and Definitions:

File System Terminology:

Q: What is a file system?
A: A file system is a storage directory structure maintained by the kernel.

Q: What is a virtual file system (VFS)?
A: A virtual file system is an abstraction of a file system used by an operating system's kernel. With the VFS, the OS can grant different user applications access to files.

Q: What is FUSE?
A: FUSE stands for "Filesystem in Userspace" and it allows users to create their own version of a filesystem without having to touch kernel code. It is an easy way to bridge users to the kernel interface.

Q: What is the nullfs driver?
A: The nullfs driver is an option that can be enabled in the kernel and that allows the kernel to mount a file system subtree.

Q: What does EPERM stand for?
A: EPERM is an error code that should be returned when the operating system receives any calls to read or write an encrypted file without a key set. According to the man page for the command chmod(2), EPERM means "The effective UID does not match the owner of the file, and the process is not privileged" or "The file is marked immutable or append-only."

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

Advanced Encryption Standard (AES):

Q: What is the AES algorithm and how does it work?
A: The Advanced Encryption Standard algorithm can be used to temporarily hide and later decipher confidential information. Used by defense agencies in the United States, the AES algorithm allows and denies access to information with its requirement of a key, a string of characters that only an appropriate entity should know.

Q: Why is the CBC version of the AES algorithm a better choice in this case than the EBC version of the algorithm?
A: The EBC version of the algorithm results in patterns when there is repetition in the data to be encrypted, making the encrypted data relatively easy to decode. The CBC algorithm does not have this problem, so it is a better choice in this case.

Data Structures Decisions:

Q: Why is an array an appropriate choice for storing the user ids?
A: We thought that using an array, which stores the user and their key, would be a better approach because we are only accommodating up to 16 users. The performance of retrieving keys for a particular user in the array would not matter because 16 is a relatively small size. If the number of users was larger, then we would have selected a hash table for storing keys. The implemented array contains elements that are of the type ***struct entry***, a structure that holds the userId and key_info. The ***struct key_info*** holds two unsigned integer variables, k0 and k1, that hold halves of the user key.

Q: How and when are the user keys stored?
A: The key is stored manually; we have a seperate program called **setkey** which will accomplish that for the user. The user will run **setkey** and input a 16-char key as an argument. The program will take in the key and check if the characters in the key are in hex as well. Then, the key is split into two halves. The first half of the key is for k0, and other half is for k1. The program converts each half into an unsigned int. Afterwards, it invokes our syscall: ***syscall(564, k0, k1)*** where 564 is to do setkey and k0, k1 is the halves of the key as an unsigned int.

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

Requirements:

1. Implement a system call that adds an encryption/decryption key for a particular user ID.
    a. Given setkey (unsigned int k0, unsigned int k1)
2. Implement kernel code that applies the key to a file if the key is set for the current user and the file's sticky bit is set. A separate program should only be used to set the key for a user.
3. Implement a program that allows a user to encrypt/decrypt their own files through the command line.
4. Implement functionality for the virtual filesystem (FUSE, in our case) which will modify create, read, and write operations.
    a. Create: The create operation is added so that the vfs has ability to create files.
    b. Read: The read operation is modified so that reading files will perform decryption of the file and display the decrypted content to the user if the user has the correct key set for that encrypted file.
    c. Write: The write operation is modified so that writing to an encrypted file will write the contents to that file, but stay encrypted afterwards.

Flowcharts:

The following are original functions that need to be written in protectfile.c.

Prototype: int processArguments(int argc, char *argv[])
Parameters: int argc (number of arguments parsed), char *argv[] (arguments parsed)
Output: int or error message
Return values: int
Description: This function accepts and processes user input from the command line. The function expects to find a crypt mode (encrypt or decrypt indication), a user key, and the name of a file to manipulate.
Flowchart: See Figure 1.

Prototype: int cryptFile(char *fileName, char *aesKey)
Parameters: char *fileName (name of file being processed), char *aesKey (user key)
Output: int or error message

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

Return values: int
Description: This function encrypts/decrypts the file passed into it using the given user key.
Flowchart: See Figure 2.

Prototype: int main(int argc, char *argv[])
Parameters: int argc (number of arguments parsed), char *argv[] (arguments parsed)
Output: int or error message
Return values: int
Description: This function handles the arguments passed into the program by calling *processArguments* then will do encryption/decryption based on the mode specified by the user by doing *cryptFile*. Also, the sticky bit will be turned on/off accordingly depending on the specified mode.
Flowchart: See Figure 3.

The following are original functions that need to be written in setkey.c.

Prototype: int main(int argc, char *argv[])
Parameters: int argc (number of arguments parsed), char *argv[] (arguments parsed)
Output: int or error message
Return values: int
Description: This function processes the argument passed in which is the key. It will check if they key is correct such as length, and if they are hex characters. Then, it will split the key into two halves and each half will be converted to an unsigned int k0 and k1 respectively. The k0, and k1 will be inserted into an array which holds a user and their keys by calling the setkey system call: ***syscall(564, k0, k1)***.

Programming Decisions:

Kernel Modifications:

This table describes the changes that need to be made to specific, existing files/functions in the FreeBSD kernel and the reasonings behind those modifications.

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

File	Changes	Reasoning
vfs_syscalls.c	a. (Line 4617) Implementation of the setkey syscall. b. (Line 4664) Implementation of the getuserkey syscall.	a. Needed to implement system calls to set and retrieve user keys.
syscalls.master	a. (Line 1344) Implementation of prototype of the setkey syscall. b. (Line 1346) Implementation of prototype of the getuser key syscall.	a. We needed to include the prototype for the two new system calls. This file holds prototype for all the system calls.
Makefile		a. This file copies the *vfs_syscalls.c syscalls.master* files into /usr/src/sys/kern. Then, it will run a command which will make the syscalls. Finally, it will build and install the kernel, then reboot.

File Modifications:

This segment describes the included files for this assignment, their implementations, and their purposes. The table below goes more in depth, explaining the reasoning behind these decisions.

Makefile (*created file)*
Description: Copies relevant files into the kernel that were changed. Then, it builds and installs the kernels. Afterwards, it will reboot the system.

vfs_syscalls.c (*modified file)*
Description: Added setkey syscall and getuserkey syscall to the file. The setkey syscall stores the key for the current user, and the getuserkey syscall allows us to retrieve the stored key.

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

Implementation: For setkey, it traverses through an array and finds an empty slot to set the key for the user. In getuserkey, it traverses the array and checks if the userId is there, then it returns the k0 and k1 associated for they user.

setkey.c (*created file)*
Description: Program that takes in a 16 character hexadecimal key and uses our setkey syscall to store the key for that user.
Implementation: It checks if there is a provided input key, if no, then that means we unset the key by passing in k0 and k1 as 0 for the setkey syscall for the user if the user is in the keys table. However, if there is a provided input key, program splits the 16 character key in half and converts the two halves into two unsigned ints (k0 and k1) and passes them in to our setkey syscall to store it.

protectfile.c (*created file)*
Description: Program that encrypts/decrypts a file using AES.
Implementation: Takes in three arguments: the encrypt/decrypt option (-e or -d), the 16 character hexadecimal key, and the file name to be encrypted/decrypted. See Figures 1-3 below for the logic behind the main(), processArguments(), and cryptFile() functions that we implemented. Alos, a detailed reasoning is listed in the table below.

fs.c (*created file)*
Description: Program that is the vfs (FUSE), and handles the Filesystem operations.
Implementation: Modified the create, read, and write functions. A detailed reasoning is listed in the table below.

File	Modifications	Reasoning
aes.c / aes.h	No changes	a. The necessary files for AES encryption / decryption.
setkey.c	a. int main(int argc, char *argv[])	a. It takes in one argument which is the inputKey. If there is an inputKey, then it checks if it's the correct length and if all characters are hex. It will then,

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

		split it into two halves, and convert each half into unsigned integers k0, and k1. It will store the key into the kernel using the *syscall(564, k0, k1)*. However, if no inputKey, then unset the key for the user if the user has a key in the table already.
protectfile.c	a. main(int argc, char *argv[]) b. processArguments(int argc, char *argv[]) c. cryptFile(char *fileName, char *aesKey) d. *xorBuffers(char *destination, char *bufA, char *bufB, unsigned length)	a. The function will check if user inputs are correct by calling *processArguments*. It will turn on/off sticky bit accordingly, and do the encryption / decryption by calling *cryptFile*. b. It checks the arguments passed into the program such as which encryption mode and if they key is correct length and contains all hex characters. c. It opens the file specified by *fileName*, then repeatedly reads the contents of the file into 16 size chunks. If the contents is already encrypted it will decrypt it and write the decrypted content back to the file. If the contents is plain, it will encrypt it and write the encrypted to the file d. It XORs the corresponding bytes from two input buffers into and output destination
fs.c	a. xmp_create(const char *path, mode_t mode, struct fuse_file_info *fi)	a. Included this function so FUSE has the ability to create files. b. We modified it so that when a

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

	b. xmp_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi) c. xmp_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)	file is read in FUSE, it will check if the sticky bit is set, and if so, retrieve the key for the user. It will take the key and decrypt the contents of the file and display it to the user while leaving the actual file still encrypted at the end. c. We modified it so that when a file is written to in FUSE, it will check if the sticky bit is set, and if so, retrieve the key for the user. It will then write the content to the file and encrypt the file again because the file is decrypted first from xmp_read, then written to here, then we encrypt it so the file stays encrypted at the end.
compile.sh	a. Compiles a protectfile program b. Compiles a setkey program c. Compiles a fs program (which is the FUSE)	
mountfs.sh	a. Makes a directory called fusefs b. Mounts the fusefs as the FUSE directory.	
unmount.sh	a. Unmounts fusefs b. Removes the directory: fusefs	

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

Diagrams:

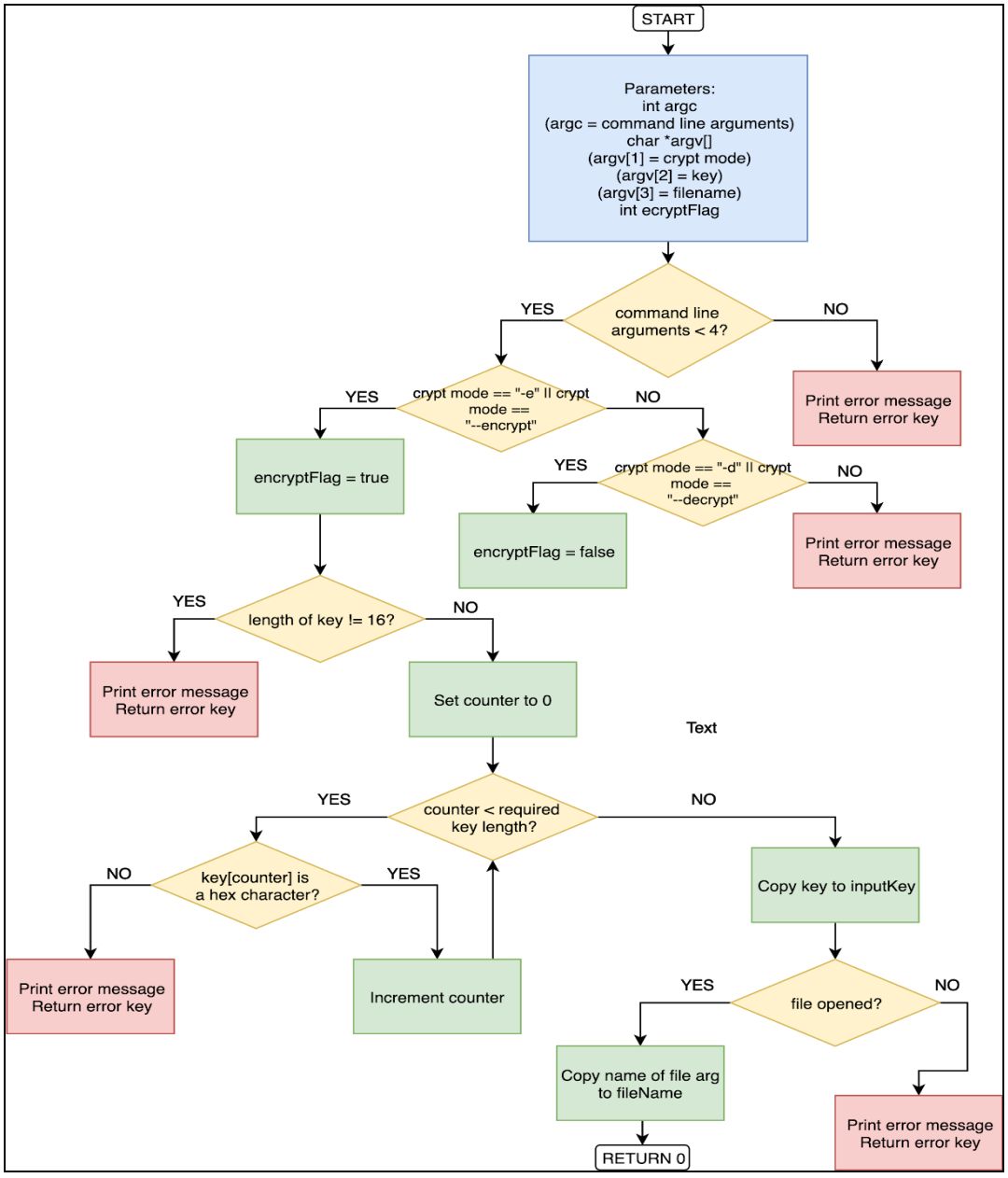

Figure 1: processArguments() Flowchart

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

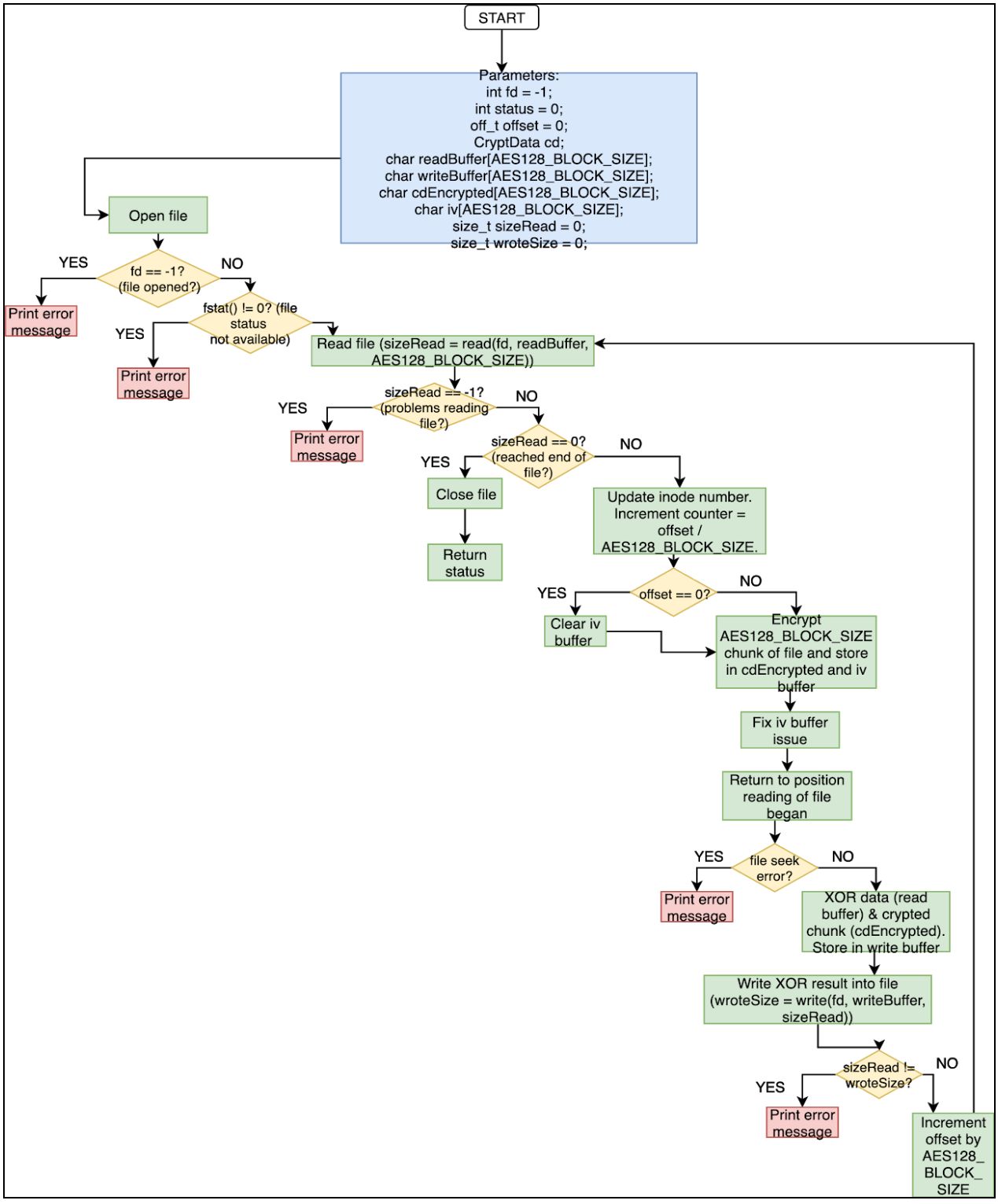

Figure 2: cryptFile() Flowchart

Members: Mackenzie Huynh (Captain)
Anthony Lin,
Khachatur Mirijanyan,
Priya Calaimany
Team: Evanescence
Date: June 2, 2019

DESIGN DOCUMENT

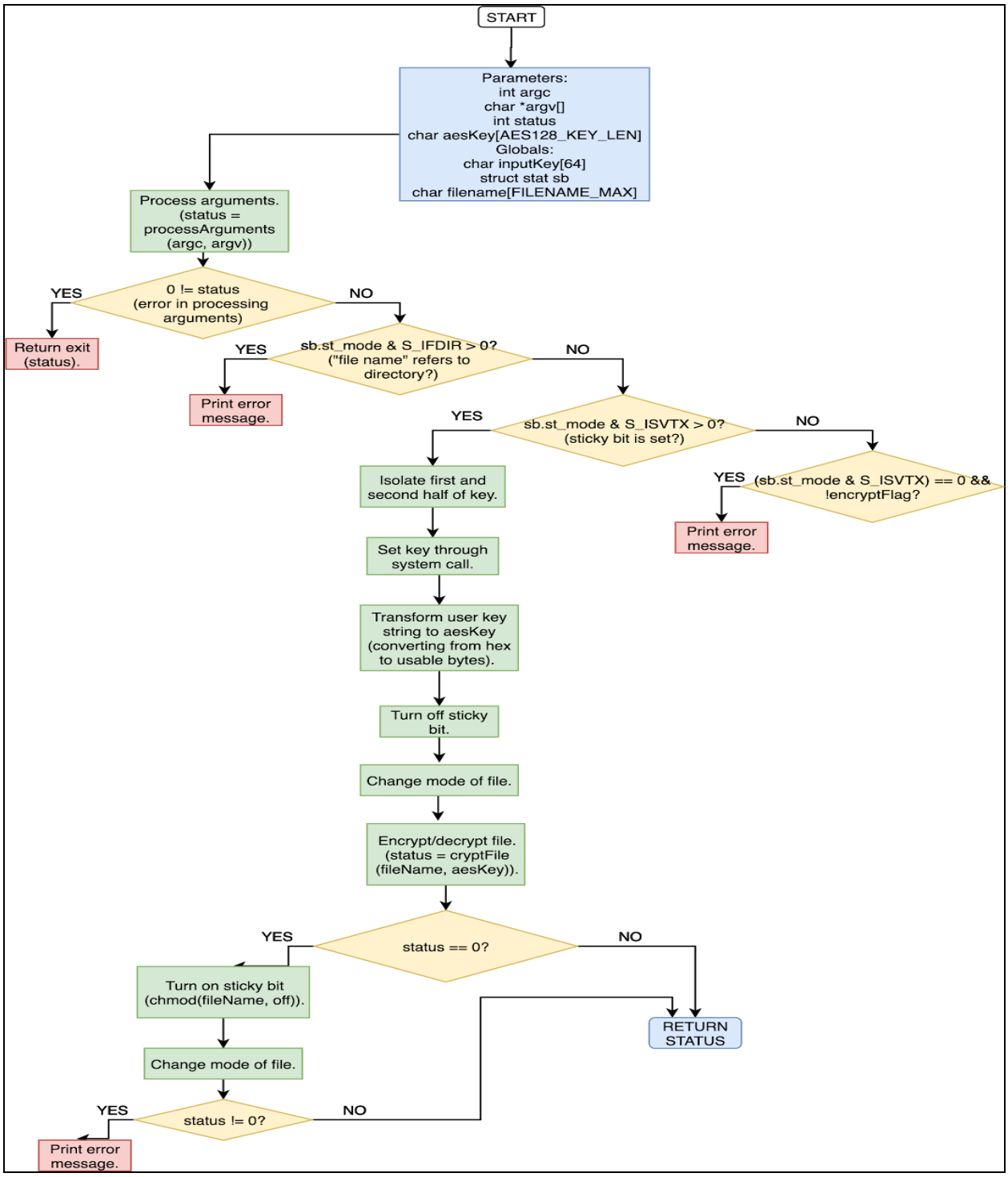

Figure 3: main() Flowchart