A study of cryptographic file systems in userspace

Sahil Naphade[a], Ajinkya Kulkarni [b] Yash Kulkarni[c], Yash Patil[d], Kaushik Lathiya[e], Sachin Pande[f]

[a] Department of Information Technology PICT, Pune, India naphadesahil@gmail.com
[b] Department of Information Technology PICT, Pune, India ajinkyakulkarni300@gmail.com
[c] Department of Information Technology PICT, Pune, India yashkulkarni99@gmail.com
[d] Department of Information Technology PICT, Pune, India yapatil2000@gmail.com
[e] Veritas Technologies Pune, India, kaushik.lathiya@veritas.com
[f] Department of Information Technology PICT, Pune, India sspande@pict.edu

Article History: Received: 10 January 2021; Revised: 12 February 2021; Accepted: 27 March 2021; Published online: 28 April 2021

Abstract: With the advancements in technology and digitization, the data storage needs are expanding; along with the data breaches which can expose sensitive data to the world. Thus, the security of the stored data is extremely important. Conventionally, there are two methods of storage of the data, the first being hiding the data and the second being encryption of the data. However, finding out hidden data is simple, and thus, is very unreliable. The second method, which is encryption, allows for accessing the data by only the person who encrypted the data using his passkey, thus allowing for higher security.

Typically, a file system is implemented in the kernel of the operating systems. However, with an increase in the complexity of the traditional file systems like ext3 and ext4, the ones that are based in the userspace of the OS are now allowing for additional features on top of them, such as encryption-decryption and compression. There are several examples of such a file system, most notable being FUSE (file system in userspace).

Owing to the need of individuals and corporations alike, several userspace file systems have been created over the years. In this paper, we are trying to shade light upon the creation of such file systems, along with the issues and the advantages of the same.

Keywords: FUSE, Encryption, userspace file systems, cryptographic file systems, OpenSSL.

1. Introduction

File systems are a common interface for the applications in a machine to access the user and application data. Typically, micro-kernels implement the file systems in userspace, but most file systems are part of monolithic kernels. This allows avoiding the message-passing overheads of the micro-kernel and userspace domains [1]. However, in recent years, userspace file systems have continuously risen in terms of popularity for several reasons, like

1) They allow for creation of stackable file systems which add to the functionality of the traditional file systems.
2) Companies making use of the userspace file systems for their data storage needs, most notable ones being Google file system [2], Apache's HDFS [3], Amazon S3 backed s3fs [4].
3) Several kernel-space file systems migrating to the userspace, for e.g., NTFS [5], ZFS [6].
4) Ability to quickly develop and test the new features, approaches, etc.
5) Ease of implementation and maintenance of a userspace level file system, owing to properly available documentation for the same.

Additionally, the userspace file systems grant a level of stability to the system. Bugs in the kernel-level file systems can cause critical system failures, one of the recent examples being the "File system corruption problem" in 2018 [7]. Userspace bugs, being more contained, allows for better detection and quick handling of the same. Support for multiple languages and multiple platforms is also desired. Keeping the above points in mind, creation of several userspace file system frameworks took place over the years, for e.g., Arla [8], Sun Microsystems's vnode [9], XFS [10]. In this paper, we try to look at and study several file systems that are proprietary or have used the popular FUSE framework. There are several reasons for choosing FUSE (file system in userspace) which is most effective; being the most widely used user-level file system. As of now, by most modest calculations, there are over 100 readily available file systems using FUSE [11]. We will try to explain the high-level design of the FUSE framework and then explore the examples which try to protect the user data by creating a stackable file system to encrypt the same.

2. Theory

We can secure the files of a user via different ways. One of the traditional ways includes hiding the files. But for obvious reasons, it is a very naive way and can be easily exploited. Hence, encryption is a better fit, as it can retrieve the data back when using a correct key. Symmetric key encryption algorithms are faster in such cases.

1) *Encryption:* We can think of the encryption as a transformation of human intelligible text into unintelligible one. The key of an algorithm can be thought as something that locks (encrypts) and unlocks (decrypts) the message as and when required. When a message is encrypted, it is almost impossible to figure out the actual

text, known as plaintext; especially when encoded by modern algorithms like AES, DES, Blowfish, etc.

A symmetric encryption algorithm uses the same key to encode the plaintext and decode the ciphertext. In an asymmetric key algorithm, we use a key pair produced at the same time. One key from the pair is used to encode the message and only other key can decode it. We use these keys where data is to be transmitted over the network. One key is made public, called Public key (which is known by all, and used to encode the message be sent to the user) and the other key is kept with the receptor of the data, called a private key (known only to the receptor).

2) *Decryption:* Decryption is the reverse of encryption, in which it converts the unintelligible text into an intelligible one, through the reverse application of the same cipher algorithm and the key which was used to encode the actual plaintext. In the case of files, the encryption of a file means encrypting the contents of the file, often done in a block-wise manner, where one block length is equal to the block length specific to the chosen algorithm. For additional security, the name of the file may also be encrypted, which will prevent information from leaking in any manner.

AES algorithms: An acronym for Advanced Encryption Standard, AES was proposed in a response to the public call from NIST (National Institute of Standards and Technology) for the development of an encryption algorithm to overcome the weakness of DES algorithm. It makes use of several key sizes to encode the data, viz. 128, 192, and 256 bits. With multiple rounds per key size, it uses a substitution-permutation, more commonly known as SP network to produce ciphertext from the plaintext.

The current best attack on AES-128 requires $2^{126.0}$ operations. For AES-192 and AES-256, $2^{189.9}$ and $2^{254.3}$ operations are needed, respectively. With cluster several million times powerful than any current computer, and operating at the thermodynamic Landauer's limit, it would still take 234 Petajoules to just increment a counter through every key value [12].

As illustrated in Table I, the number of rounds of encryption in AES algorithms varies with the length of the key used.

There are mainly five modes for using the AES algorithm.

1) Electronic Code Book (ECB mode): It is the simplest of all, with a message divided in blocks and each block encrypted differently.

2) Cipher Block Chaining (CBC mode): In this, each block is XORed with the previous ciphertext block, before being encrypted. Thus, each block depends on all the previous plaintext blocks.

3) Cipher Feedback (CFB mode): Similar to the CBC, it makes use of entire output of the block cipher. It converts a block cipher into a self-synchronizing stream cipher.

4) Output feedback (OFB mode): It converts a block cipher to a synchronous stream cipher, which is then XORed with the plaintext blocks to produce the ciphertext.

5) Counter (CTR mode): Similar to OFB, it also converts the block cipher into a stream, in which, by encrypting successive values of a counter it generates a keystream, which is then maintained.

Key size (in bits)	Number of rounds
128	10
192	12
256	14

Table I Relation Between Size of Key and Number of Rounds in AES Algorithms

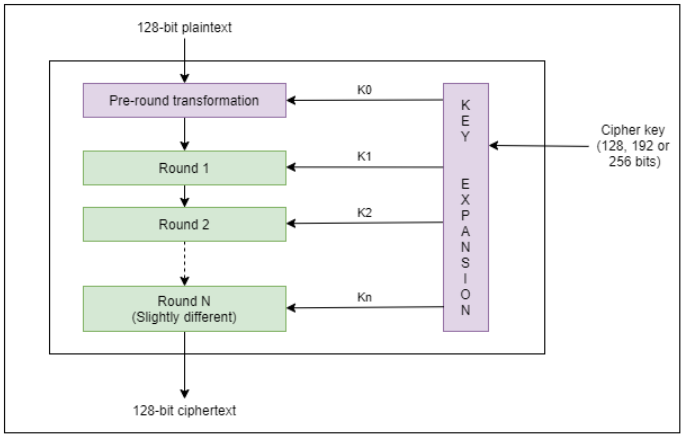

Fig. 1. AES Algorithm

3. Literature Survey

A. FUSE Design

It is the most widely used userspace file system framework. Originally a part of A Virtual file system (VFS), it has since been split into its own separate project, hosted on SourceForge.net, and the code is maintained on GitHub. It was made officially available in the mainstream Linux kernel from v2.6.14. FUSE is now available for several Operating Systems as of now, including FreeBSD, for MacOS as OSXFuse, Dokan (now Dokany) [13] for Windows, etc. This allows for cross-platform usability with a single code base set. There are two levels in which any file system developed using FUSE, called FUSE implementations. First, which is a low-level implementation of the fuse, known as 'fuse_ll_operation' and requires handling of Mutexes and locks is done by the developer, whereas the high-level implementation, known as 'fuse_operation' handles all the mutexes and locks on its own.

Owing to the simple API provided by FUSE for the developers, very little work was done till date to understand the internal architecture, working, and performance of the framework. The architecture of the FUSE is split into two parts, first is implemented in Kernel (Available in Linux kernel since v.2.6.14) and the other being the userspace daemon. This Linux kernel module registers fuse file system driver with Linux VFS. The FUSE driver acts as a proxy for various file systems implemented by different user-level daemons [1]. It registers a /dev/fuse block device, which serves as an interface between the kernel and the userspace daemon. Typically, the daemon reads the FUSE requests from the device and writes the replies back.

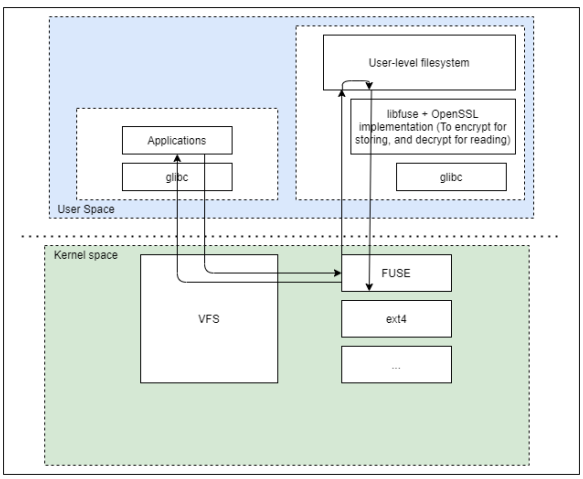

Fig. 2. FUSE architecture

FUSE target (#)	**Request names**
Initialization (3)	INIT, DESTROY, INTERRUPT
File operations (4)	OPEN, READ, WRITE, FLUSH
Directories (7)	MKDIR, OPENDIR, RMDIR, RELEASEDIR, READ-DIR, READDIRPLUS, FSYNCDIR
Attributes (2)	GETATTR, SETATTR
Extended attributes (4)	GETXATTR, SETXATTR, LISTXATTR, REMOVEXATTR
Symlinks (2)	SYMLINK, READLINK
Locking (3)	GETLK, SETLK, SETLKW
Metadata (12)	LOOKUP, FORGET, BATCH_FORGET, CREATE, UNLINK, LINK, RENAME, RENAME2, RELEASE, STATFS, ACCESS, FSYNC
Miscellaneous (6)	BMAP, FALLOCATE, MKNOD, IOCTL, POLL, NO-TIFY_REPLY

Table II Fuse Operations And The Associated Request Names.

B. Implementation details of FUSE

The FUSE kernel driver, when communicating with the userspace daemon; creates a FUSE request structure. In all, there are total of 43 FUSE request types, listed in table II [1]. The function pointers are to be initialized in the structure fuse_operations.

Here, we try to give a basic overview of the functions available in the FUSE library.

The kernel produces the INIT request when a file system is mounted. Similarly, it produces a DESTROY request when a file system is unmounted. After issuing DESTROY request, no further requests from the kernel are expected. Daemon will then perform cleanup and will exit gracefully. It creates INTERRUPT request when any previous request is no longer needed to perform.

The kernel makes READ and WRITE requests when reading and writing a file in the mounted file system, whereas it requests OPEN to open a file for I/O operations only. When OPEN is called, the FUSE daemon has an option to assign a 64-bit file handler to the opened file. It makes a FLUSH request whenever a file is closed and calls RELEASE when there are to be no more references to a previously opened file [1].

GETATTR and SETATTR are used to read and set the attributes, respectively; whereas for extended attributes, GETXATTR, SETXATTR are used. LISTXATTR and REMOVEXATTR are used to list and remove the extended attributes, respectively.

SYMLINK and READLINK are FUSE implementations of symbolic links used in Linux kernel file systems to link the files and folders.

Directory functions like MKDIR, RMDIR and rest are self-explanatory. OPENDIR and RELEASEDIR work similar to OPEN and READ for file but are meant for directories. READDIR is used to read a directory, whereas READDIRPLUS is used to return metadata of each entry.

C. API Levels in *libfuse*

There are two levels at which the FUSE library has been implemented, a lower-level API is more abstracted one, which

1) Receives requests from and parses the responses to the kernel.
2) Facilitates the mounting and configuration of the implemented file system.
3) Abstracts the version differences from the user of the system [1].

Both levels are directly implementable, and the decision lies with the user to select the level, depending on the complexity and requirement of a specific system. It also provides some tutorial examples in the GitHub repository of *libfuse* under ‘example’ directory.

D. OpenSSL

OpenSSL is a robust, fully-featured, commercial-grade toolkit for the transport layer security (TLS) and Secure Sockets Layer (SSL) protocols, which also provides a general cryptography library. With an Apache-style license, it is free to be used and distributed, thus marking the free availability. OpenSSL was released officially with v0.9.1

on 23 December 1998 marking the start of the OpenSSL project. The latest version of OpenSSL is 1.1.1i (at the time of writing) and v3.0.0 will be released in 2021.

OpenSSL supports a multitude of cipher algorithms like AES, Blowfish, Camellia, DES, etc.; hash functions like MD5, MD4, MD2, SHA-1, SHA-3 and public key cryptographic algorithms like RSA, DSA, Diffie-Hellman key exchange, Elliptic curve, etc. It does so by providing a high-level interface to the underlying cryptographic functions through EVP, which is an acronym for "digital EnVeloPe library".

Working with OpenSSL for encryption and decryption comprises following steps:

1) Setting up a cipher context.
2) Selection of cipher algorithm.
3) Setting up key and I.V. (Initialization Vector), which satisfies the size required by the selected algorithm.
4) Encrypting/decrypting the data.

Explanation of steps:

1) Setting up a cipher context can be done by using EVP_CIPHER_CTX_new() function, which returns a new instance of EVP_CIPHER_CTX.

2) Cipher algorithm is set using EVP_EncryptInit_ex(), which initializes the cipher context passed with the cipher algorithm as the argument. The cipher algorithm is an instance of EVP_CIPHER, which is a structure for cipher method implementation. Example given, EVP_aes_256_cbc() will return an EVP_CIPHER instance of AES-256 cipher algorithm in cipher block chaining (CBC) mode.

3) Setting up the key and I.V. (Initialization Vector) can be done using the RAND_bytes() function in OpenSSL library, which takes the size (in bytes) to be produced randomly as an argument. The size of key and IV can be determined from the cipher algorithm by using EVP_CIPHER_key_length() and EVP_CIPHER_iv_length() functions which take cipher context as the argument.

4) Actual encryption and decryption can be done with EVP_EncryptUpdate() and EVP_DecryptUpdate() functions respectively. It encrypts/decrypts the block size of data per iteration, which is 128 bits for AES ciphers.

5) In case of final blocks of data (in a message), which is typically less than 128 bits; if padding is set to ON (default option), then EVP_EncryptFinal_ex() and EVP_DecryptFinal_ex() will encrypt (after adding padding bits) and decrypt the data, respectively. If the padding is set to OFF and remaining data is not a multiple of block size (i.e., 128 bits in case of AES), then an error is thrown.

6) EVP also provides a way to encrypt and decrypt using a single method, which is denoted by EVP_CipherInit_ex(), EVP_CipherUpdate() and EVP_CipherFinal_ ex().

EVP provides additional methods to reset the Cipher context, clear the context, getting the block size respective to the algorithm, etc. Additionally, it supports many encryption-decryption modes, namely ECB (Electronic code block), CBC (Cipher block chain), CFB (Cipher feedback mode), OFB (output feedback mode) and CTR (counter mode).

4. Survey of Existing Solutions

There is a huge number of file systems that are implemented in userspace for the protection of the user files. Many of them are built on top of FUSE (which is one of the many userspace file systems framework) which allows for cross platform utility and some are proprietary.

A. EFS (Proprietary and general-purpose file system)

1) Released on 29 June 2009 by Microsoft, "Encrypting file system" was a component of v3.0 of NTFS file system on Windows 2000, XP pro and server 2003 [5].
2) It provided a transparent file system-level encryption using advanced standard cryptographic algorithms.
3) It used two-step security, with symmetric key algorithm to encrypt the files and using an asymmetric key algorithm to encrypt the aforementioned key (of the symmetric key algorithm to encrypt the file).
4) This encryption was not done at the application level, but the file system level.

B. eCryptFS

1) Enterprise Cryptographic file system is a disk encryption software specifically for Linux. It is POSIX compliant and has been a part of the Linux kernel since v2.6.19.

2) eCryptFS runs in the kernel space of Linux (i.e., In- kernel file system). The kernel code is maintained in Git at Kernel.org and is written in C language.
3) eCryptFS is the basis of Ubuntu's Encrypted home directory, used natively within Google's ChromeOS and is embedded transparently in several network-attached storage devices (NAS devices).
4) It allows users to store and share files on an untrusted server that cannot change or delete files without being detected.
5) It has Red Hat Inc. as its most active developer and is in the active stage of the life cycle.

C. Rclone

1) Rclone is an open-source, multi-threaded, command-line computer program, primarily created to manage the content on high latency storages.
2) It is a FUSE-based file system, written in Golang; created to assist with backup and encrypt the files to the cloud, and restore and decrypt from cloud to local storage.
3) Allows MD-5 and SHA-1 hashes check at all times, with timestamps maintained on files.
4) It supports over 50 back ends, like S3 buckets, Amazon Drive, Google Drive, Dropbox, OneDrive, etc.
5) It is an open-source software maintained on GitHub.

D. EncFS (A FUSE based file system, open-source)

1) Acronym for Encrypted file system in userspace, it transparently encrypts the files while using an arbitrary directory to store the encrypted files. EncFS runs in the userspace.
2) Created in 2003, with the latest stable release v1.9.5 released on 27 April 2018. Typically allows several encryption algorithms like Camellia, AES, and Blowfish.
3) The file system uses OpenSSL to encrypt and decrypt the data and the filenames.
4) A user-supplied password is used to decrypt a volume key, which is then used to encrypt all the filenames and contents.
5) It uses two modes of encryption, namely, stream encoding for file names and partial blocks at the end of the files and block encoding for fixed-sized file blocks of data (uses CBC mode).
6) There are several advantages to using EncFS, including but not limited to cross-platform availability, using on different physical drives, and allowing for faster backups and untrusted servers, with multi-user access.
7) It uses PBKDF2 for password hashing.
8) It is written in C++ and currently in the maintenance phase of life cycle.

E. gocyptfs

1) It is an encrypted file system written in Go language, on top of the go-fuse FUSE library. It derived its inspiration from EncFS with the addition of excellent performance. The project was first released in 2015, under MIT license.
2) It adds to well-known cryptographic primitives, like scrypt for key derivation, AESGCM for file-content encryption, and the first time use of EME wide-block encryption for the file name encryption in an encrypted file system.
3) It offers a significant performance improvement over EncFS, to 258 MiB/s for write from 100 MiB/s (of EncFS); and 289 MiB/s from 185 MiB/s (of EncFS) for reads.
4) Each directory has its own 128-bit directory IV stored in the same.
5) It is an active project, hosted on GitHub.

F. CryFS

1) Released in 2015, it is a FUSE-based solution to support cloud storage like OneDrive, Dropbox and iCloud.
2) Written in C++ under LGPLv3 license, it resulted from a master thesis at KIT university that used chunked storage to obfuscate the file sizes.
3) It is in the active stage of maintenance.

5. Conclusion

We have surveyed FUSE architecture and implementation of OpenSSL library, which needs additional tutorials and walkthroughs to get started with. A lot of the examples use the OpenSSL library to allow for cross-platform utility, and the addition of the FUSE library to different platforms makes them truly cross-platform implementations. While exploring various cryptographic file systems, we noticed that several of the implementations are outdated, with the performance measurement done on legacy software, such as the ext3 file system, which has now been replaced with ext4. Some projects are also in the maintenance phase, while some are in the active stage, with some being replaced with their successors, like EncFS succeeded by gocryptfs.

6. Future Scope

In future work, these issues warrant investigation into performance of the said implementations on newer software and hardware, which is likely to increase the performance. We may also create additional tutorials and walkthroughs on the topics to allow beginners to easily work with them.

References

1. B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: Performance of user-space file systems," 15th USENIX Conference on File and Storage Technologies, vol. 15, pp. 59 – 72, 2017.
2. S. Ghemawat, H. Gobioff, and S. Leung, "The google file system," In Proceedings of the 19th ACM Symposium on Operating Systems Principles, SOSP '03, vol. 19, pp. 29 – 43, 2003.
3. The apache foundation, hadoop. The Apache Foundation. [Online]. Available: http:// hadoop.apache.org
4. Fuse-based file system backed by amazon s3. [Online]. Available: https://github.com/s3fs-fuse/s3fs-fuse
5. Open-source cross-platform ntfs file system. [Online]. Available: www.tuxera.com
6. Zfs for linux. [Online]. Available: www.zfs-fuse.net
7. A file system corruption bug breaks loose. [Online]. Available: https://lwn.net/Articles/774440/
8. A. Westerlund and J. Danielsson, "Arla - a free afs client," Annual USENIX Technical Conference, 1998.
9. D. Steere, J. Kistler, and M. Satyanarayan, "Efficient user-level file cache management on the sun vnode interface," In Proceedings of the Summer USENIX Technical Conference, Anaheim, CA, 1990.
10. F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," In Proceedings of the First USENIX Conference on File and Storage Technologies, pp. 231 – 244, 2002.
11. e. Vasily Tarasov, "Terra incognita: On the practicality of user-space file systems," USENIX, 2015.
12. A. Mathew, C. Kulkarni, and Y. Kulkarni, TEST Engineering Manage- ment, vol. 83, pp. 2137 – 2143, 2020.
13. Dokany, a user mode file system library for Windows. Dokan-dev. [Online]
14. Available: https://github.com/dokan-dev