EncFS goes Multi-User: Adding Access Control to an Encrypted File System

Dominik Leibenger
CISPA, Saarland University
dominik.leibenger@uni-saarland.de

Jonas Fortmann
University of Paderborn
jonas.fortmann@web.de

Christoph Sorge
CISPA, Saarland University
christoph.sorge@uni-saarland.de

***Abstract*—Among the different existing cryptographic file systems, *EncFS* has a unique feature that makes it attractive for backup setups involving untrusted (cloud) storage. It is a file-based overlay file system in normal operation (i.e., it maintains a directory hierarchy by storing encrypted representations of files and folders in a specific source folder), but its *reverse mode* allows to reverse this process: Users can mount deterministic, encrypted views of their local, unencrypted files *on the fly*, allowing synchronization to untrusted storage using standard tools like *rsync* without having to store encrypted representations on the local hard drive.**

So far, EncFS is a single-user solution: All files of a folder are encrypted using the same, static key; file access rights are passed through to the encrypted representation, but not otherwise considered. In this paper, we work out how multi-user support can be integrated into EncFS and its reverse mode in particular. We present an extension that a) stores individual files' owner/group information and permissions in a confidential and authenticated manner, and b) cryptographically enforces thereby specified read rights. For this, we introduce user-specific keys and an appropriate, automatic key management. Given a user's key and a complete encrypted source directory, the extension allows access to exactly those files the user is authorized for according to the corresponding owner/group/permissions information. Just like EncFS, our extension depends only on *symmetric* cryptographic primitives.

I. Introduction

Usage of cloud storage for backup purposes either requires strong trust into the provider with respect to its confidentiality guarantees, or use of an appropriate end-to-end encryption layer. A solution often used in practice is EncFS [1], which also serves as basis for, e.g., the cloud security service BoxCryptor [2].

EncFS manages a folder hierarchy by mapping files and folders one-to-one to an encrypted folder hierarchy stored in a designated *source folder* of an existing file system. File/folder names and file contents are thereby encrypted symmetrically using a single secret key; metadata like timestamps, owner information and access rights are simply passed through to the *entities* (i.e., files and folders) stored in the source directory.

While confidentiality can also be achieved by volume-based solutions (e.g., Veracrypt [3] or dm-crypt [4]) or specialized backup tools (e.g., duplicity [5]), the mentioned approach has the advantage of being mostly transparent to the cloud storage as it works at the granularity of files as envisaged by typical cloud APIs. Arbitrary files can be read without touching other entities and write operations affect only single entities, which especially preserves the opportunity of creating efficient, server-side snapshots if supported by the provider.[1]

In contrast to other file-based encryption tools, EncFS has a unique feature: It allows to reverse its functionality as to generate a deterministic, encrypted view of an existing (unencrypted) folder on a local file system on the fly. The encrypted view can be synchronized to external, untrusted cloud storage using standard tools like rsync [6] without having to store a local copy and without requiring changes to the local file system. This so-called *reverse mode* might be used, e.g., to create backups of folders that are already otherwise encrypted—avoiding overhead due to two encryption layers in normal operation.

A. Scenario

Exemplarily, EncFS reverse mode can be used in the following scenario that we focus on in this paper: Different members of a household (family members or people sharing a flat) or of a small company share a Linux server for managing individual data and exchanging data with one another. All data are stored in a shared folder; the standard Linux file system access rights are used to restrict access to specific entities. Both the server and its administrator (e.g., a designated member) are ultimately trusted, but the remaining members are not. To ensure that the data remains recoverable in case of a hard drive failure, the administrator has set up a nightly backup to some cloud storage shared by all members: Every night, an encrypted view on the shared folder is mounted using EncFS reverse mode[2] on the shared server, its content is synchronized to the cloud storage via rsync, a server-side snapshot is created, and EncFS is stopped afterwards.

EncFS has two important limitations in this scenario: First, since all members have access to the cloud storage, they might tamper with access rights of files stored there as to gain access to them on the (local) shared server after a recovery. Since access rights are stored in file system metadata, they might not even be present in the remote copy at all—depending on the file system and protocols used by the cloud provider. Second, in case of a failure of the shared server, data can only be recovered by the administrator as she is the only person in possession of the encryption key used by EncFS. This might be a problem, e.g., if she is on vacation. To circumvent this,

[1] Although encryption causes some overhead since small changes to file contents typically change their complete ciphertexts.

[2] Normal operation would be possible as well by using a permanently mounted EncFS folder as shared folder and backing up its source folder.

978-1-5090-3065-1/16/$31.00 ©2016 IEEE

the administrator could provide the respective key to other members, but this would counteract access controls set up on the server: Any member in possession of the key could decrypt any other user's files irrespective of whether she has been granted the required rights on the server.

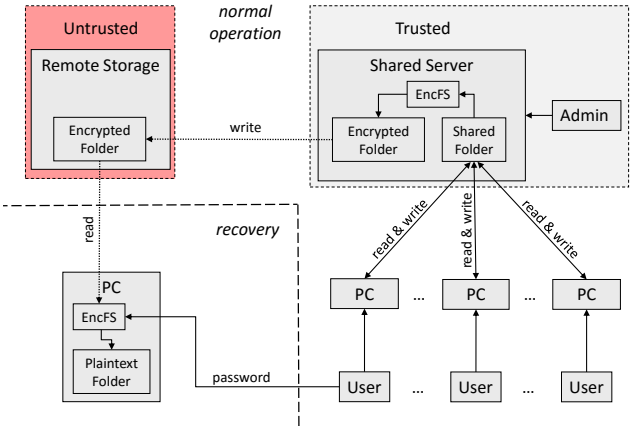

Fig. 1. Scenario. Several users have access to a shared server, which is managed by a designated administrator. Both server and admin are ultimately trusted (light gray). The server stores encrypted backups created via EncFS on external untrusted storage (red) accessible to all users. Using their individual passwords, users shall be able to recover exactly those files they are authorized for directly from the remote storage using an arbitrary PC running EncFS.

B. Goals and Contribution

The contribution of this paper is to improve upon the state of the art of cloud storage security by introducing a solution for encrypted backups on untrusted cloud storage for multiple users that is *easy to set up* and *simple to administer*. For this, we build upon EncFS's unique reverse mode and show how to overcome the aforementioned limitations as to support the usage scenario illustrated in Figure 1: An administrator shall be able to back up shared folders using EncFS's reverse mode just like before, but users shall be able to recover files they have access rights for from these backups directly using their passwords.

To this end, we extend EncFS's reverse mode so that

1) ownership information and access rights for files can be stored and remain recoverable even if a storage provider lacks support for file system metadata,
2) users are able to recover files they have access rights for with respect to the previous scenario even if the secret key managed by the administrator is not available, and
3) required changes to EncFS's architecture and concepts—and, thus, implementation costs—are small. In particular, we do not introduce dependencies to asymmetric cryptographic primitives (which might seem natural) as they are not necessarily required.

We introduce additional file- and user-specific keys and develop a hierarchical key management mapping file system access rights to keys in such a way that users are able to decrypt exactly those files they are authorized to read according to the file system's access rights. We further investigate security aspects of our extension and present a working prototype implementation that is based on a recent development version of EncFS.

C. Structure

Section II explains basics on EncFS and Linux access rights and Section III describes our threat model. The concept is described in detail in Section IV; some words on our implementation follow in Section V. Security is evaluated in Section VI and related work is presented in Section VII; Section VIII concludes the paper.

II. Basics

We present EncFS's architecture in an abstraction allowing to develop our extension and we summarize the traditional Linux file system access rights semantics so that we can derive a consistent mapping from access rights to a key hierarchy later on.

A. EncFS

EncFS [1] is a FUSE[7]-based virtual file system developed by Valient Gough. As mentioned in Section I, it is an overlay file system that encrypts individual entities of a folder hierarchy. It uses a single, symmetric, global key—the *volume key*.

Execution of EncFS requires two folders to be provided: A *source folder* where encrypted representations of files and folders reside, and a *mountpoint* that allows access to the cleartext representation of the managed folder hierarchy. When EncFS is first started with a specific (empty) source folder, some initialization is required. The user may define different parameters affecting EncFS's operation, including choice of an encryption scheme and a password required to decrypt the source folder later on. For the actual encryption, a random volume key is generated. Together with the chosen parameters, it is stored encrypted (using a key derived from the user's password via PBKDF2 [8]) in a configuration file located in the source folder which is evaluated on subsequent starts.

While EncFS is running, de- and encryption are performed on the fly: When a folder is accessed, the names of its files and subfolders are decrypted; when file contents are read or written, they are transparently de- or encrypted, respectively. To be able to determine corresponding paths in the encrypted and decrypted views without requiring a designated database that stores this mapping, EncFS encrypts entity names using *deterministic encryption*. Given a cleartext path, EncFS computes its encrypted counterpart by deterministically encrypting each of its individual components.

EncFS provides a plethora of options that allow to adjust the trade-off between security and efficiency with respect to different operations. Ciphertexts of path components, e.g., can either be encrypted in isolation or depending on the respective parent path (which affects move operations), file contents can be encrypted either deterministically or using a—more secure—randomized scheme, etc. In principle, our concept is independent from a specific choice of these options, so we skip a detailed discussion and refer the interested reader to the EncFS documentation [9], [10] and to EncFS's recent security audit [11].

The *reverse mode* of EncFS is basically identical to its normal operation, but the direction of en- and decryption

is reversed. Here, a folder containing an unencrypted folder hierarchy is used as source folder and the encrypted view is provided at the mountpoint as illustrated in Figure 2. As already described in Section I-A, this encrypted view can be synchronized to some cloud storage provider and later mounted in *normal mode* to recover data. Again, a configuration file is generated on EncFS's first start and stored and only visible in the source folder.[3] There is one major difference from a security perspective, though: As encrypted views are created on the fly but have to be consistent across different invocations to be suitable for backups, deterministic encryption has to be used. Some security options are, thus, currently not available in reverse mode.

Note that these restrictions apply to our prototype implementation as we focus on reverse mode in this paper. They are not conceptional restrictions for our work, though: First, it is straightforward to extend our concept to EncFS's normal mode (although re-implementation of some features would be required which are provided for free by EncFS reverse mode). Second, some reverse mode restrictions could by eliminated with further implementation efforts (e.g., by introducing a persistent cache for IVs as to allow randomized encryption).

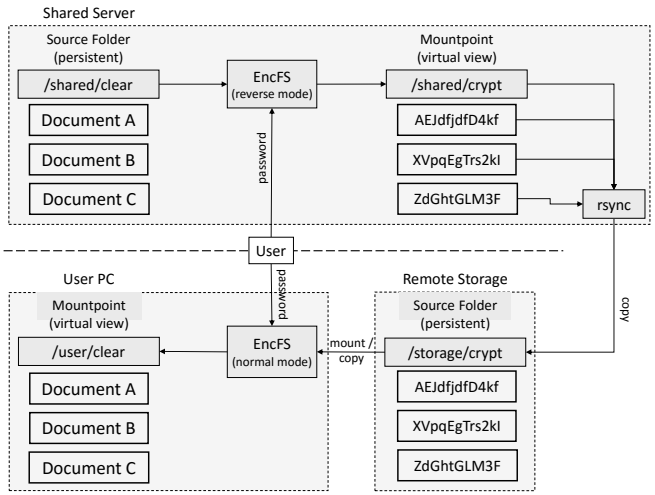

Fig. 2. EncFS reverse mode in backup scenario. The source folder is a folder on the shared server containing some unencrypted data. The mountpoint contains an encrypted version which can be copied to remote storage as a backup. To recover data, EncFS is started in normal mode on a user's system, using the copy of the encrypted view as source folder and providing the cleartext view at another mountpoint.

B. Linux Access Rights

Users under Linux are organized as follows: Each user is identified by a *uid* (user id) and can be in an arbitrary number of groups (each identified by *gid*, group id). Traditionally, access rights to files/folders are organized as follows: Each entity has exactly one owner and one group, identified by uid and gid, respectively. An entity's owner does not have to be in its group. Permissions can be specified with respect to three sets of users: The *owner*, *group*, and *others* (i.e., system users that are neither the owner nor in the respective group). For each set, *read* (r), *write* (w) and *execute* (x) rights can be assigned. The precise semantics of the respective rights depends on the type of the entity they are assigned for (i.e., file or folder), as shown in Table I:

For files, read access allows to read their contents; write access allows modification of contents, but not reading of data written before. Execute access allows execution of binary (executable) file contents, even without permission to read the executed code.

For folders, read allows listing names of child entities. Write allows creation/deletion/renaming of entities, irrespective of permissions of the respective entities. Execute allows to open the folder and is required to access any of its children. A user with execute but no read right can thus access children as long as she knows their names; a user with read but no execute right may see the children's names, but cannot access any of them.

An important aspect of the distinction between owner, group and others rights is that these sets of users are considered non-overlapping. Effective rights are defined by the first matching set of users: If, e.g., rights to a file are exclusively granted to a group, the file owner cannot exercise them even if she is a member of that group. Rights only granted to *others* apply to everyone but the owner and members of the file's group.

Right	File	Folder
Read	Read file content	List names of files and subfolders
Write	Modify content	Create/delete/modify files/subfolders
Execute	Execute binary file	Open folder and access child entities

TABLE I. ACCESS RIGHTS ON LINUX-BASED FILE SYSTEMS

In addition to the described access control model, many file systems support access control lists (ACLs) that allow for more fine-grained access rights. They are, however, less commonly used than the traditional rights that we focus on in this paper.

III. THREAT MODEL

The foundation of our work is the scenario described in Section I-A. We distinguish three parties: The local server administrator (including the server), other local users of the server (including their devices having access to the server), and an external cloud storage provider that is accessible by all local users and used for backing up data from the local server. While read access to the cloud storage is sufficient for local users in our scenario (since all backups are uploaded by the local server), we consider a stronger security model where all users have write access to the cloud storage as well.

We require the admin to be benign, i.e., we assume that she sets up access control on the local server in a correct and effective way. She is completely trusted by all users. The remaining users are considered potentially malicious, i.e., they are attackers who try to get access to data of other users they are not authorized for, either via passive attacks or actively, e.g., by trying to tamper with access rights in an outsourced backup. They are interested in entity names and file contents. With respect to a flat share, this would correspond to a scenario where only a single user—the administrator—is IT-savvy and trusted to keep her system

[3] If using reverse mode, the admin thus has to remember also backing up the configuration file when synchronizing the encrypted view to remote storage. This file, however, only has to be backed up once and is preferably stored at a separate, secure location as offline brute force attacks on a user's password are possible given that file (data is encrypted using a password-derived key).

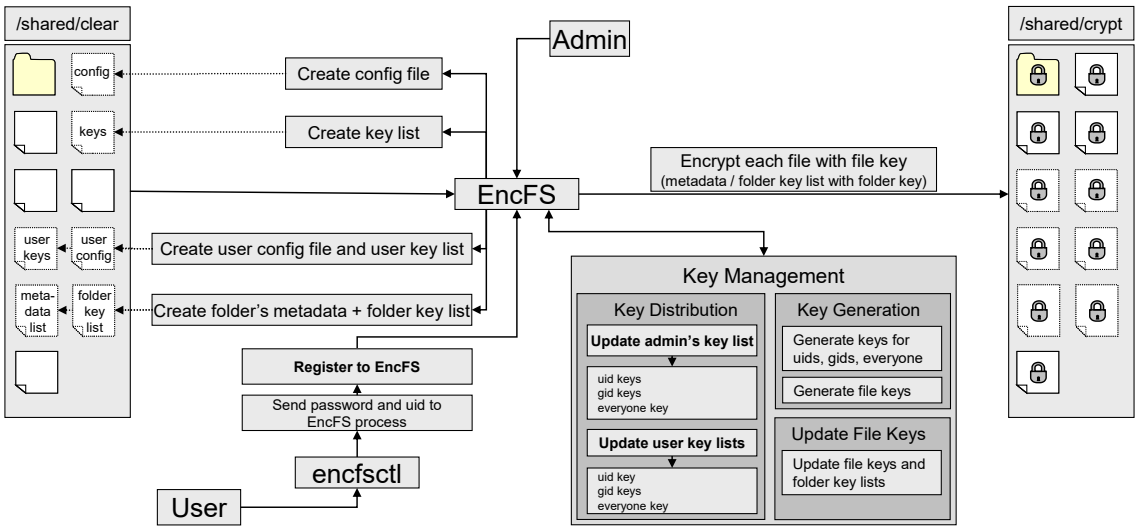

Fig. 3. Multi-user EncFS overview. The admin starts an EncFS process on the shared server which creates the required keys, takes care of key management and encrypts all entities in the source folder with the file keys. Users can set user-specific passwords using *encfsctl*, yielding user config files and key lists.

secure, while the remaining users have good intentions but might be working on potentially infected systems due to lack of IT expertise.

The cloud provider is considered honest but curious, i.e., it is assumed that it stores data correctly (as it has a financial incentive in doing so) but tries to gain information about stored data (file contents and file/folder names).

To be able to prove our work's security, we assume the deterministic encryption scheme used by EncFS's reverse mode is DAE-secure (see [12]) and that the HMAC scheme [13] used for symmetric authentication produces unforgeable MACs.[4]

IV. Concept

Now we present the concept of our multi-user extension. First, an overview of the overall concept is given; afterwards, the main components—key management and metadata management (encryption / authentication of access rights)—are presented in detail before concluding with some specifics on our extension's usage for creation of backups and later recovery.

A. Overview

In plain EncFS, every volume—i.e., every instance consisting of source folder and mountpoint—has a global volume key used to encrypt every single file. Encrypted with a key derived from a password specified by the volume's admin, the volume key is stored in a configuration file. The configuration file is supposed to be stored separately from the actual backup data as to prevent an untrusted cloud storage provider, e.g., from performing brute-force attacks on the volume admin's password. Metadata like ownership information, access rights and timestamps are not specifically considered but passed through to the source folder's entities.

[4]This is not the case in practice since EncFS insecurely uses AES-CBC for deterministic encryption. This issue is out of scope of this paper, though, and could be resolved easily, e.g., by using SIV-mode [12] encryption instead.

The core concept of our extension is illustrated in Figure 3 and explained in more detail below. Every entity gets an individual key, the *file key*[5], which is used to encrypt its name and content and chosen at random when first required. For directories, the key is further used to encrypt two administrative data structures which are serialized to files residing in the encrypted representations of the respective directories: The *metadata list* stores owner, group and specified permissions for a directory's child entities, each symmetrically encrypted and authenticated using the file key of the respective entity as to decouple these metadata from the capabilities of the source file system or used external storage. The *folder key list* contains the file keys of all child entities, each encrypted using a single *uid key* which is generated randomly and made available to the administrator.

Then, further keys are introduced to enable limited access for specific other users: For each user, we introduce a *uid key* that is associated to her uid; for each group, we create a corresponding *gid key* associated to its gid, and we generate an *everyone key*[6] that represents all users of a system at a certain point of time. Each key is generated randomly on its first use and made available to users with respect to their uids and group memberships and to the admin. According to the owner and group information and permissions of respective files, we then create copies of their file key items in the folder key list and encrypt them with respective uid, gid and/or everyone keys instead of the admin's uid key as to allow their decryption to all authorized users (including the admin as she has access to all uid/gid keys and everyone key). Since there is no folder key list that could contain the file key of the root folder, the everyone key is used for the root folder instead, making sure that all users can access the volume root.

Similar to the configuration file containing the encrypted volume key of the admin, each user who wants to be able to

[5]For simplicity, we refer to a key corresponding to a folder as *file* key, too.

[6]We chose this name to emphasize that there is no 1:1 relationship between the *everyone* key (which is distributed to all users) and the *others* group (which encapsulates only users that neither match an entity's owner nor group).

access her files independently from the admin has to generate an individual configuration file containing a randomly chosen *user volume key* encrypted with a key derived from her password. User volume keys are used to distribute uid, gid and everyone keys: For each user, a list containing her uid key, the everyone key and all gid keys of groups she is in are stored in a key list file. The file is encrypted with her user volume key[7] and located in the volume's encrypted root folder so that it is included in external backups.

Note that our extension preserves EncFS's concept of separating password-derived keys from actual backup data: Password-derived keys are used only in user-specific configuration files which have to be backed up *once* at a secure location. Only the key graph consisting solely of *randomly-generated* keys is stored as part of the backup data as it might change frequently.

B. Key Management

According to the traditional Linux file system access rights (see Section II-B), a user is allowed to read an entity's content (file content or names of a folder's children) iff the following two conditions hold: a) She has the execute right for the entity's parent folder, and b) she has read permission for the entity itself. To be able to access the entity, the user further has to know its precise name or she must have a read right for the parent folder allowing her to list its children's names.

Considering the backup scenario described in Section I-A, there is no functional requirement for leaking names of entities to users who are not authorized to access them. Users would only suffer from decreased confidentiality with respect to their own files. For this reason, we continue EncFS's strategy of encrypting file contents and metadata (i.e., the file's name) using the same key, i.e., we reveal plaintext names only to users that are also authorized to access the respective entity. If a user only has execute permission for a directory and a read right for a file in that directory, however, this concept would reveal the file's name to the user although she is not authorized to see it in the local file system. As this would be in conflict with our threat model (see Section III) since it reveals information to users for which they are not authorized, we restrict access to folders to users having both execute and read permission. Note that this decision imposes a limitation on recovery as in principle, it might prevent users from recovering some file contents they had access to. This, however, is only true for contents of files that are *invisible* to the respective user (as they reside in folders the user is *not authorized* to read). We argue that in the considered scenario, such situations are more likely to occur due to configuration errors than being actually intended, so we opted for the much simpler (thus, cleaner) concept that ensures that files hidden from users cannot be decrypted with additional knowledge like path names under any circumstances. In the rare case that such rights are actually intended, the respective files can still be restored by the server administrator, though.

Based on these considerations, we distribute file keys as follows to users having access to their corresponding entity's parent folder: The file key for an entity accessible only to its owner (i.e., the owner has *read* permission in case of a file or *read* and *execute* in case of a directory) is encrypted with the owner's *uid key*; for entities accessible to owner and group, copies of their file keys are encrypted with the *uid key* and *gid key*[8], respectively. For entities accessible by everyone, the *everyone key* is used; for entities accessible only by root, the admin's *volume key* is used. In the remaining special cases (see Section II-B), we determine the list of users that are effectively authorized and encrypt the file key with each of their *uid keys*, respectively. The detailed mapping of rights to keys used to construct *folder key lists* is given in Table II. To prevent users from determining file keys of entities without access to parents, each folder key list is encrypted using its folder's file key. An overview of this hierarchic key management concept is given in Figure 4.

	File permissions (Folder permissions)							
Owner	r (rx)	r (rx)	r (rx)	r (rx)	- (-)	- (-)	- (-)	- (-)
Group	r (rx)	r (rx)	- (-)	- (-)	r (rx)	r (rx)	- (-)	- (-)
Others	r (rx)	- (-)	r (rx)	- (-)	r (rx)	- (-)	r (rx)	- (-)
Keys	everyone	owner uid (if owner $\notin$ group), gid	uid $\forall$ users $\notin$ group	owner uid	uid $\forall$ users $\in$ system $\setminus$ {owner}	uid $\forall$ users $\in$ group $\setminus$ {owner}	uid $\forall$ users $\notin$ group $\cup$ {owner}	volume

TABLE II. MAPPING OF TRADITIONAL FILE SYSTEM ACCESS RIGHTS TO KEYS

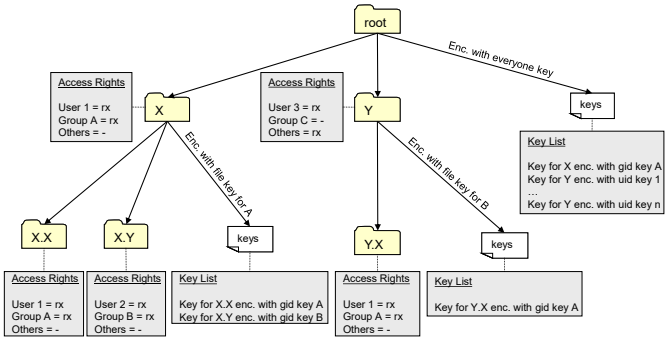

Fig. 4. Example of key management in multi-user EncFS where users 1, 2, 3 are in groups A, B, C, respectively. The root directory contains a folder key list encrypted with the everyone key so that every user can decrypt it. The list contains keys needed for files/folders in the root folder, each encrypted using appropriate uid/gid keys or everyone key w.r.t. the entity's access rights. Every subfolder contains a separate folder key list containing the keys for its child entities. The key list is encrypted with the file key of the folder; its keys are encrypted w.r.t. the rights of the respective entities.

Whenever rights change, key management has to ensure that affected file keys are changed and re-distributed accordingly. Our extension detects such changes by comparing the rights set on the local file system with its internal data structures (i.e., metadata and folder key lists) on each access of an entity. A change is handled as follows: If rights of an entity change, the scope of the change is limited to the respective entity. The extension determines whether there are any users that *lose* access to the entity. If so, a new random file key is generated for the entity and distributed to authorized users using its parent folder's folder key list as described above. In case of a folder, all children's file keys are recursively renewed as well. If the set of authorized users grows due to the change, the file key remains unchanged and only the folder key list is updated.

[7]Note that the *global volume key* is the admin's *user volume key*.

[8]If the owner is in the entity's group anyway, only the gid key is used.

If a user's group memberships change, the affected gid keys are renewed. As the change is not necessarily limited to certain subtrees of the EncFS volume, the whole volume is scanned recursively and the effective rights of each individual entity are compared against the internal data structures. For any affected entity, key renewal and distribution is performed as described before. Added/removed users are handled similarly.

Note that with regard to computation costs, access right changes are relatively cheap due to the nature of EncFS's unique reverse mode we focus on: Even if the whole volume has to be scanned and many keys have to be renewed as described before, this operation affects *only metadata*, i.e., the folder key list files. The reason is that the actual encryption is performed lazily as soon as a folder or file in the mountpoint is accessed. In fact, encryption is always performed on the fly and ciphertexts of file contents are not stored at all, so costs to access a file content are identical irrespective of whether its key has changed since a previous access. Moreover, for access right changes of files/folders within the source folder of an EncFS folder (in contrast to changes to the system's user/group database), even metadata changes (i.e., changes to folder key lists) are computed lazily on access of respective directories. We emphasize that this laziness does not have security implications since all changes are applied as soon as the respective parts of the encrypted volume are accessed (which is trivial considering that the encrypted volume is generated on the fly). After a backup tool has traversed the encrypted volume for synchronization to remote storage, for instance, the external backup includes all lazy changes.

C. Metadata Management

Metadata management is simple. We have a *metadata list* file for each folder that stores metadata for its children: Every line contains the name, owner (uid), group (gid) and permissions of a child entity and is encrypted and authenticated symmetrically with the entity's file key to ensure that it can only be read/modified by users who are at least read-authorized. EncFS's filename encryption function is used for that as it guarantees confidentiality and authenticity.

To ensure that access to metadata of a folder's children requires access to the folder itself, metadata files (i.e., lists of encrypted items) are encrypted with their folders' file keys.

D. Recovery

The presented solution allows users to recover any files they had access to from a *shared folder* given a backup created with EncFS reverse mode, provided that they have created a user volume key and backed up the corresponding user-specific configuration file before. Recovery is performed as follows: A user starts EncFS in normal mode, using the encrypted backup (e.g., the cloud storage) as source folder and specifying her configuration file including her password. EncFS uses the user's volume key to decrypt her key list, yielding her uid key, everyone key and all gid keys of groups she is in; the everyone key allows decryption of the root folder's folder key list.

Whenever a folder is accessed, EncFS extracts the file keys of all entities the user has access to. For this, it tries to decrypt all items of the folder key list with the everyone key, the uid key and all known gid keys one after another, and uses the decrypted file keys to decrypt their corresponding items from the metadata list. All entities whose file keys and metadata have been successfully decrypted are displayed to the user; remaining entities are assumed to be non-accessible due to lack of access rights and therefore hidden. This corresponds to the default behaviour of EncFS, which hides entities whose names could not be decrypted using the global volume key.

E. Consistency Considerations

To achieve strong availability guarantees when backing up data to external storage using our extension, the admin has to consider a few details that do not apply to plain EncFS.

1) Permission changes that cause users to lose rights might result in renewal of *many* file keys. While file keys should be accessible using the user-specific keys at any time, storage of key lists required to access file keys is spread over different files. Without further preparations, interruption of a file-wise synchronization of an encrypted EncFS folder to external storage could thus leave a corrupted backup. If a cloud provider supports snapshots, we recommend to create a snapshot before any synchronization to ensure that a consistent state is available. Otherwise, susceptibility to errors can be reduced by synchronizing new files before changes and deletions. Since an updated file key of an entity implies a new name for its encrypted representation, this strategy ensures that all entities have already been backed up when key lists are overwritten.

2) In contrast to plain EncFS, encrypted representations of our extension's volumes contain uids, gids and permissions of all files and folders. This allows users to recover their own data using their user-specific keys. The additionally stored data, however, might not be sufficient for a successful recovery of the shared folder when performed by the admin after data loss. As only uids/gids but no user/group names are included, the administrator might need to also back up the server's mapping between uids/gids and user/group names, respectively. This could be achieved easily by including the files `/etc/passwd` and `/etc/group` in the encrypted backup.

V. Implementation

We prototypically implemented the concept into the reverse mode of EncFS. We based it upon its `dev` branch [14] as it already contains incompatible changes to its stable version. Further, we consider the security issues uncovered in [11] more likely to be fixed in upcoming non-compatible releases than in the stable version.

Figure 5 gives an overview of the EncFS architecture: The left side shows the original architecture; the right side shows the components that have been introduced or changed as part of our extension.

The most important components we introduced are the *KeyManager* and *MetadataIO*. The former manages (i.e., generates and changes) the newly introduced file-specific keys and distributes it to authorized users; the latter manages metadata of entities in an encrypted and authenticated manner.

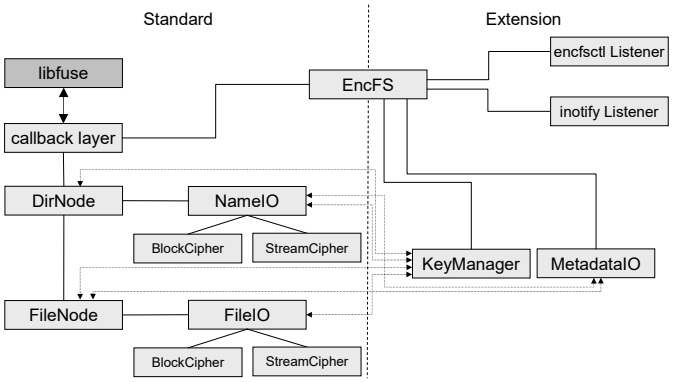

Fig. 5. Extended EncFS, based on EncFS's architecture as illustrated in [15]

MetadataIO replaces EncFS's concept of passing access rights through to the respective entities in a volume's encrypted representation. Instead, entities in the encrypted representation are created with minimum rights and owner/group set to `root`. This hides information about access rights from curious cloud storage providers and prevents other members from tampering with access rights as described in Section I-A.

The administrative tool *encfsctl* has been extended so that users can generate user-specific keys for a mounted EncFS volume, allowing decryption only of files they have access to.

In order to react to implicit access right changes caused by changed users or group memberships, our extension evaluates `/etc/group` and `/etc/passwd` on each start of EncFS and updates user- and group-specific keys accordingly if needed. If users lose access to specific files due to such changes, the corresponding file-specific keys are renewed as soon as they are accessed for the first time. To also detect such changes while EncFS is running, we made EncFS listen to and react to updates of these files using *inotify* [16].

Usage of our EncFS extension is only little different from that of standard EncFS: When initializing a new volume, the user can specify the `--multi` argument to enable the extension. A flag is written to the generated configuration file to ensure that the extension is used during each subsequent mount of the volume irrespective of whether the argument is explicitly provided. On initialization, the extension initializes a folder key list for the root folder and a key list file for the admin user as described in Section IV-A. Together with a third file used for detection of metadata changes, both files are stored as hidden files accessible only to root in the (cleartext) source folder. After initialization, the files contain only the data that are necessary to access the root folder. Folder key lists of subfolders and further uid/gid keys necessary for key distribution are created lazily as soon as the respective folders in the mountpoint are actually accessed (e.g., when the folder hierarchy is traversed by a backup tool). At this point, the only benefit of our extension lies in its ability to encrypt and authenticate metadata, but there is no difference to standard EncFS from a usage perspective: Using the same commands as in standard EncFS, the administrator can unmount/mount the EncFS volume at will using her password and her configuration file and recover any files from a backup created using reverse mode by mounting it in normal mode.

To benefit from the introduced multi-user capabilities, users have to generate individual configuration files. For this, they have to log in once on a terminal and run the command `encfsctl multisetuserpw` while the EncFS volume is mounted, i.e., the EncFS process is running. This triggers creation of a random user volume key for the user invoking the command within the context of the EncFS process, and forces creation of a user-specific configuration file containing that key (protected by a password specified by the user) as described in Section IV-A. The user-specific key list containing the user's uid/gid keys and everyone key—encrypted with the user volume key—is further created in the (cleartext) source folder of the EncFS volume and made available in the encrypted representation. By providing the `--multidecode <uid>` flag, the user can later force usage of that key list when mounting the encrypted EncFS volume using her user-specific configuration file, yielding exactly those files she is authorized to access.

VI. Security Analysis

Depending on its choice of parameters and its usage scenario, EncFS itself suffers from some security deficiencies as shown in an audit [11]. These issues surely affect the security of our extension, but they are not specific to it and can be resolved in isolation. For this security analysis, we assume that our extension is based on an EncFS version in which these deficiencies have already been fixed (as described in Section III).

The analysis is structured as follows: First we show that key distribution is consistent to the semantics of traditional Linux file system access rights, i.e., that users receive an entity's file key iff they are allowed to access the entity according to the file system's access rights. Then we show that file contents and names are only accessible by users in possession of the corresponding file key. Afterwards, we show that changed rights are handled correctly: We prove that neither implicit nor explicit changes result in situations in which a user who lost rights is still able to determine an entity's file key, and we show that even active attackers trying to tamper with rights will fail to determine file keys they are not authorized for.

1) Linux Access Rights and Key Distribution: Assume a user is able to determine a file key of an entity she is not authorized to access given her access to the full encrypted EncFS volume (e.g., a backup stored at the remote storage). File keys have length ≥ 128 bits and are chosen at random by the (benign) server, so trivial attacks (brute force, malicious system) are ruled out. As the only place a file key is stored is in the folder key list of its entity's parent folder, the user must have decrypted the corresponding item to determine it. For this, she must have known one of the keys used for the file key's encryption as in Table II and also the file key of the entity's parent folder as the folder key list is encrypted using that key. Recursively, she must have known a key used for file key encryption (Table II) for *each* path element of the respective entity. As shown in Section IV-B, this is the case iff she is allowed to access every path element (and, thus, the entity itself) according to the file system's access rights model, which contradicts the assumption that the user is not authorized to access the entity.

2) Confidentiality of File Contents and Names: File contents and names are exclusively stored encrypted with the respective entity's file key. Given security of the encryption scheme, confidentiality follows directly from file keys being only known to users authorized to read their respective entities.

3) Handling of Access Right Changes: Given correctness of key distribution as proven above, a user cannot determine any file key corresponding to an entity she is not authorized to read if having access only to a specific state of the EncFS volume's encrypted representation. If rights change and a user has access to different states (i.e., from before and after the change), however, she might be able to still read entities she lost access to if key renewal was handled incorrectly. Assume this is the case, i.e., that any change results in a situation in which a user does not have a read right to an entity anymore but is still able to decrypt it. This can only be the case if either the file key of the entity itself or a set of keys that allow its decryption remained unchanged across the access right change.

This is prevented as described in Section IV-B: Whenever a right changes explicitly (entity permission change) or implicitly (changed users/groups), we determine the effective set of read-authorized users of all entities that could be affected and renew their file keys accordingly. The same applies to uid/gid/everyone keys. Renewal of all file keys whose decryption depends on a specific key whenever the respective key (i.e., a parent folder's file key or a uid/gid/everyone key) is changed ensures that the set of users gaining knowledge of a specific file key can only grow with changes. This excludes prior states from being usable for access right escalation.

4) Authenticity of Access Rights: Access rights of entities are stored in *metadata list* files; each item is symmetrically authenticated using the respective entity's file key. This does not prevent read-authorized users from changing rights in the encrypted representation of a volume, but it prevents users without read permission from gaining access to entities: Since those users do not know the entities' file keys and the used authentication scheme is assumed to have unforgeable MACs (see Section III), they can only compute a correct MAC if they also change the respective file key. Changing that key would clearly prevent access to the entity's name and content. Since every entity gets its own, random file key, they cannot copy metadata items from other files, either. They could only replay old metadata items of the same file that are authenticated using the same file key. This attack, however, could not broaden access rights: If the set of users having access to an entity was larger in a prior state, the file key would have changed with the access right change as described before.

VII. Related Work

In 2014, results of a security audit of EncFS 1.7.4 have been published by Taylor Hornby. [11] It uncovered weaknesses that render it insecure especially in situations in which an attacker has access to several states of a volume—as it is the case in the scenario considered in the paper at hand (see Section I-A). These weaknesses, however, can be fixed relatively easily if compatibility to prior versions is not required. While these fixes are out of scope of our work, we see no obstacles in distributing the extension along with the fixes in the future, given that either of them introduces incompatible modifications anyway. Besides the audit, there is no academic literature dealing with security aspects of EncFS in particular to the best of our knowledge. Work on other cryptographic file systems, however, deals with questions similar to ours: SiRiUS [17] is an overlay file system that encrypts files with individual *file encryption key*s (FEKs) like we do. Keys are distributed to authorized users using metadata files containing FEKs encrypted with user-specific keys, which is similar to our *folder key list* concept. SiRiUS uses asymmetric encryption for key distribution and stores one copy of each file key for each authorized user. Our solution, in contrast, depends only on symmetric schemes and collects users into larger groups as to decrease key distribution efforts. Plutus [18] also aims at low key distribution costs in presence of file-specific keys. Instead of collecting users into groups with shared keys, the authors collect files with similar rights into *filegroups* sharing a key. Using a filegroup's key, users can decrypt a *lockbox* that stores the actual file-specific keys. The authors also introduce *key rotation*, i.e., they compute successor keys after right changes in a way that prior keys remain computable, which allows to postpone re-encryption of files after file key changes, e.g., to the next change of their contents. Security issues of key rotation have later been fixed by *key regression*. [19] Tahoe-LAFS [20], a secure multi-user file system, depends on file-specific keys, too, but has a completely different key distribution concept: Keys are wrapped into *capabilities* that are shared directly with users to grant the respective rights. To ease key distribution for many files, their capabilities are included in folder representations, resulting in semantics different from our system: In Tahoe-LAFS, read access to a folder implies read access to all children.

The concept of encrypting (file) keys using other keys that are made accessible to a group of users as to delegate access rights, which is shared by most of the works mentioned above, is used even more extensively by Grolimund et al. [21]: The authors formalize encryption of a key b using a as a *cryptographic link* $a \rightarrow b$ and present a tree-based key distribution data structure, *Cryptree*. As to minimize the number of keys that need to be distributed when granting specific combinations of access rights, they introduce a number of different keys for entities with corresponding links across them. Again, access right semantics are different from ours: Read permission for a file, e.g., implies access to all parent folders in their system.

Our solution makes extensive use of cryptographic links, though: Folder key lists essentially build an alternative cryptographic tree tailored to Linux's file system access rights model. Links are thereby realized using EncFS's filename encryption function, which is deterministic but considered insecure as worked out in [11]. Secure alternatives for this so-called *key-wrap problem* are presented by Rogaway and Shrimpton [12].

Key assignment schemes in general are discussed by Crampton et al. [22]: The authors present a framework for key assignment in hierarchical access control that embraces a wide range of schemes commonly used in other literature and discuss the respective schemes' security aspects. Note that there is a plethora of more recent work on key distribution in

encrypted file systems and cloud storage in particular which is based on more sophisticated cryptographic primitives like attribute-based encryption (ABE) [23]. While some of those schemes might be applicable to our setting in principle, we opted for a solution based only on symmetric schemes as they are sufficient to achieve our goals and more easy to integrate into EncFS's architecture. Furthermore, there are lots of works that deal with more specific aspects of cloud storage security like, e.g., data deduplication, which are orthogonal to the work at hand. For a more comprehensive overview of such related work, we refer to [24].

All in all, our presented extension borrows techniques from different well-known cryptographic file systems, but arranges them uniquely. We are not aware of any other system that builds key hierarchies almost consistent to a given file system's access rights, nor are we aware of systems allowing on-the-fly creation of encrypted backups that allow partial recovery by authorized users as realized by our extension.

VIII. Conclusion and Outlook

We have presented an extension for the encrypted file system EncFS that adds multi-user support. For this, the extension does not use the global volume key to encrypt data, but it introduces randomly generated, file-specific keys and implements an automatic key management and distribution to authorized users. As in plain EncFS, the administrator of an EncFS volume is the only user in possession of a volume key that allows access to all encrypted data. In addition, however, user-specific keys for local users can be generated. Based on the owner/group/others access rights used by Linux file systems, the extension ensures that users can decrypt exactly those files using their user-specific keys that they are allowed to read according to these access rights. Both explicit (changed file permissions) and implicit right changes (changed users/groups) are considered by the presented concept and its security has been proven. We implemented it in an EncFS developer version and verified functionality in reverse mode.

For future work, we plan to extend the concept to ACL-based rights. The mapping of traditional rights to keys could also be refined: Currently, users authorized for reading files who lack read rights for their parent folders are not able to decrypt them. A more complex key management could solve this issue. Orthogonal efforts are required considering the security options of EncFS. Our extension would immediately take advantage from any fixed security issues of EncFS.

References

[1] https://github.com/vgough/encfs.

[2] https://www.boxcryptor.com.

[3] https://veracrypt.codeplex.com/.

[4] https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt.

[5] http://duplicity.nongnu.org/.

[6] A. Tridgell and P. Mackerras, "The rsync algorithm," Department of Computer Science, Australian National University, Tech. Rep. TR-CS-96-05, Jun. 1996.

[7] https://github.com/libfuse/libfuse.

[8] B. Kaliski, "PKCS #5: Password-Based Cryptography Specification Version 2.0," RFC 2898 (Informational), Internet Engineering Task Force, Sep. 2000. [Online]. Available: http://www.ietf.org/rfc/rfc2898.txt

[9] V. Gough, *encfs(1) – Linux man page*, July 2014.

[10] ——, "encfs/DESIGN.md," https://github.com/vgough/encfs/blob/6909139e703d9208623e840c097ba4f98a743bde/DESIGN.md, Feb 2015.

[11] T. Hornby, "EncFS Security Audit," https://defuse.ca/audits/encfs.htm, Tech. Rep., 2014.

[12] P. Rogaway and T. Shrimpton, "A provable-security treatment of the key-wrap problem," in *Proc. EUROCRYPT '06*, 2006, pp. 373–390.

[13] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104 (Informational), IETF, 1997.

[14] https://github.com/vgough/encfs/tree/dev.

[15] V. Gough, "Encfs presentation," Libre Software Meeting, France, Jul. 2005, https://sites.google.com/a/arg0.net/www/encfs-presentation.pdf.

[16] "inotify – monitoring filesystem events," http://man7.org/linux/man-pages/man7/inotify.7.html.

[17] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS: Securing remote untrusted storage," in *Proc. NDSS*, 2003.

[18] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proc. USENIX FAST*, 2003, pp. 29–42.

[19] K. Fu, S. Kamara, and T. Kohno, "Key regression: Enabling efficient key distribution for secure distributed storage," in *Proc. NDSS*, 2006.

[20] Z. Wilcox-O'Hearn and B. Warner, "Tahoe: the least-authority filesystem," in *Proc. StorageSS '08*. ACM, 2008, pp. 21–26.

[21] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer, "Cryptree: A folder tree structure for cryptographic file systems," in *25th IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2006, pp. 189–198.

[22] J. Crampton, K. Martin, and P. Wild, "On key assignment for hierarchical access control," in *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006, pp. 98–111.

[23] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *2007 IEEE Symposium on Security and Privacy (SP '07)*, May 2007, pp. 321–334.

[24] D. Leibenger and C. Sorge, "sec-cs: Getting the most out of untrusted cloud storage," 2016, arXiv preprint. [Online]. Available: https://arxiv.org/pdf/1606.03368