Cryptographically Enforced Permissions for Fully Decentralized File Systems

Johanna Amann, Thomas Fuhrmann
Technische Universität München
{ amann | fuhrmann } @ so.in.tum.de

***Abstract*—Distributed file systems nowadays work well in many ways. They provide efficient solutions, for example, to distribute data among a global team. But most systems do not address the complex subject of secure user and group management. The systems that do, usually offer only a very limited subset of access permissions that is incompatible to the permissions usually used in Unix-like systems.**

In this paper, we propose a new system for user and group management, which cryptographically enforces access permissions in fully decentralized file systems. Our proposal is twofold: an integrity verification algorithm checks the validity of the current file system state; a cryptographic data protection scheme, added on top of the integrity verification, preserves the privacy of the file system content.

Except for signatures, our system uses symmetric cryptography only. It thus incurs only a reasonable cryptographic cost in the system.

I. Introduction

Files are important – for software systems and for people. Files are written, read, and edited both, locally by individual users and globally by large teams within multi-national organizations. The way to publish, share, and access the files varies from organization to organization. Companies typically have network drives to share the files within a local office. Big companies can afford the bandwidth and administration effort to use network drives globally, too. Small companies typically use communication systems such as e-mail or instant messaging to collaboratively edit the files. Or they use version control systems and web-based systems to enable the team members to access the files from everywhere.

Typically, network file systems rely on centralized components and centralized administration. This results in high cost for bandwidth, highly reliable components, and skilled personnel. The goal of our work is to create an easy-to-use, fully decentralized distributed file system that provides transparent file access to its users. We base our work on structured peer-to-peer techniques, because – when they are deployed correctly – P2P systems are extremely scalable and robust. In this paper, we go beyond these well studied aspects of P2P file systems and address a yet unsolved problem, namely how to realize Unix-like access permissions in such systems.

A. Problem Statement

A Distributed File System (*DFS*), faces a multitude of attack scenarios, which we have to guard against. It is not sufficient to cryptographically secure the client connections, because there are no trusted servers in a peer-to-peer system. Everything that is stored on another peer has to be encrypted and signed in a way that ensures that the data cannot be read by unauthorized clients and cannot be tampered with.

Things get even more complicated, when we consider that usually several users want to interact. They want to be able to read and write certain files within the same file system. That means that we cannot use traditional cryptographic means, because they are not geared to multiple users. We also have to think about group access where group membership may change over time.

In this paper, we introduce a new file system structure. With this structure we can implement nearly the full spectrum of access rights, that Unix users are accustomed to, on a fully decentralized file system. Our system is based on a "plain" P2P file system that is unaware of users and groups. On top of such a plain file system we employ cryptographic means to realize users and groups. We also discuss how to add ACL support on top of our approach.

B. Access permissions

The distributed file system should be as transparent to our users as possible. Thus the access permissions we want to support are modeled closely to the POSIX.1 [1] permissions used in Linux as well as many other systems.

In more detail, we want to be able to have different users and groups with separate access rights. A group consists of an arbitrary number of users. A file or directory is owned by exactly one user and exactly one group. Each user can be a member of an arbitrary number of groups.

The users and groups of our file system do not need to match the users and groups of the underlying Unix system.

A file has separate read and write access flags, that can be set be separately for the user, the group, and all others who can access the system. We assume that everyone who may write to a file also has read permissions to the file in question.

Most systems support additional access rights on top of this simple system. In section VI-A we will discuss other access permissions often found in Unix-like systems and how they could be implemented using our approach. In section VI-B we sketch a way to implement ACLs on top of our proposed access permission system.

C. Previous Work

The ideas we present here extend the Igor File System (IgorFs) [2], [3], which has been developed within our group.

IgorFs is based on a structured P2P overlay network called Igor that provides a *key based routing service* [4] similar to Chord [5]. Unlike a distributed hash table, which offers a publish/subscribe service only, our network is a service oriented overlay network. It routes messages based on a destination key and a service identifier. It is up to the respective service implementation how these messages are handled.

IgorFs is one of the applications that have been built on top of that overlay network. It uses a fully decentralized approach without any central or special nodes. IgorFs uses Fuse [6] to provide applications with transparent access to remote storage resources.

IgorFs uses a distributed hash table (DHT) to distribute and find its data. All objects in the file system are cut into variably sized chunks, encrypted and inserted into the DHT.

Every stored data chunk is identified by an (Id, Key)-tuple. A directory contains a list of the (Id, Key)-tuples of all the contained files and directories. Directories are serialized so that they can be stored in the underlying chunk storage system.

In that plain system, a user can access the whole file system, if she can access the data chunks containing the root-directory. From there, she can recursively access all files and directories stored in the file system.

D. Design Criteria

Our goal is to create a fully decentralized distributed file system (DFS), which efficiently implements Unix-like access permissions. The following criteria guided our design:

Trust in own machine: We assume that the user can trust his/her own machine. Scenarios where an attacker reads the cryptographic material from the machine memory are not a part of our threat scenario.

Untrusted storage: We assume that all other nodes in the network are untrusted. Thus, the access rules have to be enforced by cryptography, not by node-side policies. A client must only be able to decrypt data for which it has the appropriate access rights. Access rights enforcement must not depend on a central authority or on other nodes.

Data integrity: We want to be able to prevent or at least detect attacks on our file system, e. g. illegitimate modifications of the data by third parties. Rollback attacks on the data in our file system should be prevented. That means it should be impossible to replace a current version of a file with an old version of a file, if the client already had knowledge of the current file version.

Data confidentiality: Users must not be able to decrypt data that they do not have adequate permissions for.

User revocation: We want to be able to efficiently revoke users from the system and remove users from groups without the need for out-of-band communication.

No on-line third parties: Our scheme shall work without requiring any third parties to be online and without requiring specific other nodes to be online.

E. Contribution

In this paper, we propose a new way to organize the directory structure of a fully decentralized distributed file system. This new structure allows us to secure the integrity and confidentiality of the data contained in the file system. A client can verify the validity of the file system structure and data on the fly while accessing it. Unix-like access permissions are cryptographically enforced in the directory structure. A malicious node is not able to read or write files for which it does not have the necessary access rights. Even when a malicious user controls (some of) the nodes, attacks are limited to forking and withholding new information. The approach presented here does not depend on any trusted nodes or otherwise centralized components.

In contrast to other works our approach solely relies on symmetric cryptographic primitives to enforce the access rights and should thus be very efficient. Asymmetric cryptography is only used for the signature algorithms.

This paper is organized as follows: In section II we briefly describe our idea and introduce the technical background on which we base our work. Section III outlines our approach for validating the data integrity in a distributed file system. Section IV describes the handling of file access rights. Section V estimates the overhead imposed by our proposal. Section VI explains how to extend our approach to more sophisticated access control mechanisms. Section VII discusses the related work and section VIII gives the conclusions and an outlook to future work.

II. Design Overview

In our scenario, multiple, cryptographically separated file systems can share the network and the attached storage capacity. Each file system is controlled by a so-called *file system superuser*. The superuser creates the file system, generates the keys for new users and groups, adds users to groups, removes users from groups, and removes users from the file system. This is very similar to the *certificate authority* (CA) in public key infrastructure schemes. The superuser typically is the system administrator of your institution.

The superuser is not a centralized component in our file system, because it does not have to be online and it is not bound to a specific node. The superuser simply is the only person who has the keys that are needed to perform some special operations, which are restricted to the root user in traditional Unix systems.

Anyone can start his or her own file system by generating a new set of superuser keys. All such separately created file systems are separated beyond the scope of the access rights described in this paper: neither the superuser nor any user can access the content of another file system. Nevertheless, the encrypted data may share the same underlying storage, so that (partly) identical files can be stored efficiently.

According to our design goals, this is not a security problem. It is rather a consequence of the encryption used by the underlying peer-to-peer storage which was already briefly discussed in section I-C. In our system, the same data will always yield the same encrypted data block. This means, that if you know the unencrypted data, you can prove that a specific

node is saving an (encrypted) copy of the data. If this is of concern, a different underlying block encryption algorithm must be used. In our system, it remains however open, whether the node just caches a copy, or if it actually read its content.

Our architecture consists of two tightly connected parts: The data integrity protection and the data confidentiality protection. Because of their complex interactions, we will explain them first briefly in sections II-A and II-B and then again in more detail later in this paper (sections III and IV).

A. Data Integrity

Our first goal is *data integrity*: We want to be able to detect illegitimate modifications of data when accessing the file system.

It is well known [7] that in a fully decentralized system manipulations of files or directories cannot be entirely prevented. To implement this, it would be necessary to check each change of the file system for validity at the moment it happens. This is impossible without a trusted central authority that can give us the latest state of the file system. Instead, each client has to check the files for inconsistencies on its own. A rollback can only be noticed when, e.g., a client that has already seen more recent changes to the file system notices that they have been reverted. In file systems with a large number of concurrent users, it is a sound assumption that such manipulations will be detected very quickly; in file systems that are only accessed very rarely such manipulations could go unnoticed for a long time.

Our system prevents rollback attacks in the sense that it is able to guarantee that a client that has the knowledge of a current file will never accept an older revision of the file and show it to the user.

In order to do so, we change the directory structure in a way that is transparent to the user. For the user, everything looks like a traditional Unix directory structure (cf. fig. 1). We call this structure, that is exposed to the user the *external directory structure*.

Internally, the file system uses a completely different structure to save the directory hierarchy. We call it the *internal directory structure*. This internal structure is mapped to the external directory structure on the fly, i.e. when a user accesses the file system. It is never directly exposed to the outside. The internal directory structure contains more information than the external directory structure. This information is needed to implement the security features.

The internal structure is enhanced as follows (cf. fig 2): each user is assigned a user-root directory. It contains all the files belonging to that user. It does not contain files of other users. The user-root directory of each user contains a version number, which we can use to check each file in the directory for validity.

The different user-directories are glued together with redirects. That means that, if a user directory seems to contain a file belonging to another user, it contains in fact only an invisible link pointing to the real file location within the other user's invisible internal directory.

B. Data Confidentiality

In addition to the data integrity architecture explained in the previous section, we also propose a new method for keeping the data in our file system confidential.

To this end, we add a dedicated group-root directory for each group in the system. Each group-root directory is split into a private and a public section (cf. fig. 2). The public section can be read by anyone, the private section is encrypted and can only be read by group members.

The (ID, Key)-tuples for each file in our file system are encrypted using symmetric cryptography, unless the files are world-readable.

For each file, there are two different encrypted pointers. The first pointer resides in the home directory of the user (see previous section). If the file is world-readable, this pointer is not encrypted; otherwise the pointer is encrypted with the user's private key, so that only the user in question can decrypt the file.

A second copy of the pointer is placed in the group directory; in the public section if it is world-readable or in the private section if it is only group-readable.

If a file is modified by the file owner, the pointers in the home directory as well as in the group directory are updated. If another group member updates the file, only the entry in the group directory is updated.

Thus, if a node wants to access a file, it has to check both entries and choose the one that has been updated most recently.

Using this structure, we can map most of the Unix permissions to our file system. A detailed analysis of this approach will follow in section IV.

III. Data Integrity

In section II-A we already gave a short overview of the steps that are taken to protect the data in our systems. This section will give a more in depth overview of the new directory structure.

We have to protect against several types of attacks; most prominently we have to be aware of any unauthorized manipulations of files or directories. If we detected a manipulation, we simply abort and output an error message. Alternatively, we could revert to an earlier revision of the file system, where the manipulation is not present. This has the advantage of being fully transparent to the user, but the user won't notice she is working with an outdated file system revision. Even worse, if the user makes changes to the file system, the other clients might not accept the new revision due to the still present inconsistencies.

A. Splitting the directory-tree

As described in section II-B, in our file system, the internal directory representation of the file system differs from the externally visible structure. To the outside, the directory structure still looks like the normal Unix directory structure shown in figure 1, but internally our file system uses shadow-directories that contain the actual files. Redirects make them visible in their respective directories.

Name	Type	Use	Generation
K_u	symmetric	user encryption key	by superuser on user generation
S_u	asymmetric	user signature key	by superuser on user generation
K_g	symmetric	group encryption key	by superuser on group membership change
S_g	asymmetric	group signature key	by superuser on group membership change
K_d	symmetric	directory forward key	by user or group on directory write
(Id, key)-tuple	symmetric	data chunk key	by client upon write

Table I
KEYS USED IN THE FILE SYSTEM

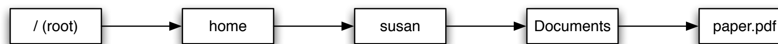

Figure 1. Normal Unix directory structure

Each user is assigned an individual user-root directory r_u, which contains only the user's own files. Every group gets assigned an individual group-root directory r_g, which contains all files belonging to the group. Each file and directory in the file system has a version number v, that is incremented upon each change. Thus each of these user and group directories has a version number v attached to it. All clients track the version numbers of all user- and group directories. If a client notices, that the version $v(r)$ of a directory decremented since the last access, the directory was illegitimately modified by a replay attack.

This approach fully prevents rollback attacks for user-root directories. In group-root directories, rollback attacks are still possible: Assume the head version of the group directory is $v_{current}$. If a node with group access is presented an old version v_{old}, it does not notice the manipulation, unless that node has already read a version v' with $v' > v_{old}$. If the node modifies the group directory sufficiently often, namely until it has produced a version v_{new} with $v_{new} > v_{current}$, all other nodes in the file system would accept that version.

The following approach achieves fork consistency in face of the described attack [8]: All user-roots and all group-roots are extended with a *version vector* that contains the versions of all users and groups. Upon each write, a user increments her user-root version and updates the version vector. Upon writing to a group, the group-root version and version vector are also updated. Thereby, an attacker can only fork the file system. The attacker cannot create inconsistencies.

To avoid the overhead of the version vectors, we recommend to implement this extension only if it is actually needed.

In order to map the normal Unix directory structure to the new internal structure, we use redirects. A redirect R is an internal construct of the file system that points to one or more other objects in the file system, i.e. files, directories or symbolic links. When a node encounters a redirect while accessing the file-system, it automatically follows the redirect and returns the referenced object. The functionality offered by redirects is similar to soft links in Unix. In contrast to soft links, the link is invisible to the clients accessing the file system.

More formally, a redirect is a n-tuple, pointing to n possible locations for a data object by an absolute path in the hidden directory structure. A redirect may not point to another redirect. In the case of our file system, the typical case is $n = 2$ for files and $n = 3$ for directories. For files one of the pointers points to the file's location within the user-root and another one points to the location within the group-root. For directories one part of the redirect points to an object within a user directory. The second and third part of the redirect point to the public and private group directory.

B. Example

Figure 2 shows this new directory structure for our example directory layout. The internal root directory contains a *.users* directory, which in turn contains all the user-root directories. It also contains a *.groups* directory, which contains all the group-root directories. Finally it contains a *.keys* directory, which contains the encrypted access keys for group access.

When a node accesses the file system, it first accesses the visible root-directory. It is contained within the user-root of the superuser and is named "/". This directory has a redirect pointing to the real home-directory, so that it seems to have a subdirectory named *home*. In reality, *home* is not a subdirectory of this directory, but a subdirectory of the user-root of the superuser. (The group links have been omitted for this directory.) The home-directory then contains the entry *susan*, which once again is just a redirect to a directory inside the user-root of the user. The home-directory in turn contains the *Documents* directory. For this directory all three redirects are shown. One points to the user-root. The other two point to the *public* and *private* directories within the group-root. According to their permissions files are placed in either one of these directories in addition to the user-root.

When a user accesses the *Documents* directory within the home directory of the user Susan she will follow the three parts of the redirect and retrieve the *Documents* directory in the user-root of Susan, in the public part of the group-root and if she is a member of the group also the *Documents* directory in the private part of the group-root.

Assume a file named *paper.pdf* that is stored within the *Documents* directory. Because of its access permissions, the

file pointers are stored once in the user-root and once in the *public* group directory. When a user wants to access the file, she has to check, which version of the file is more current. To this end, the version numbers of both files are compared and the more recent pointers are returned. Note we use version numbers not timestamps. Hence we do not require any clock synchronization.

C. Securing the user-directories

The state of each user directory is secured with a Merkle-tree [9], [10].

We already established, that each user has an own root directory r_u with a version number $v(r_u)$. We use a hash tree to secure the directory contents of all subdirectories of r_u.

All directory contents that may not be modified are hashed. The resulting value is saved in the parent directory, which is hashed again. This process is repeated in turn until we arrive at the user's top directory.

When a directory entry changes, the hashes have to be recomputed along the path from the directory to the root, i.e. in the parent directories of the changed directory). Assume e.g. that Susan changes the file *paper.pdf* which is shown in fig. 2. Because of this change, the entry in the directory *Documents* changes and the hash-value of the documents directory also changes. The hash-value of the user-root of Susan will also change and she has to be re-sign it.

The validity of each directory entry can now easily be verified by checking the signature of the user-root and the hash-values of the subdirectories.

Figure 2 shows the hashes above the directory names. The directories with signed hashes are drawn with a shaded background.

IV. Enforcing permissions cryptographically

In this section, we describe how the file system enforces the actual file permissions. I. e. we explain how users may change the integrity protected meta-information according to their access rights.

A. Owner access

As already said, all files that belong to a user u are located somewhere in her user-root-directory r_u. Each user has a encryption key K_u and an signature key S_u (see table I). For world-readable files, the block pointers are stored as plain text. For non-world-readable files, the pointers are encrypted with the encryption key of the owner. Thus only the owner will be able to read the file contents.

If a file is world-writable, the pointers are not included in the Merkle tree and anyone can change them without invalidating the structure. If a file is not world-writable and is changed by the owner, the writer has to recompute the Merkle tree and to sign the new top-hash with the signature key. Thus only the owner can change non-world-readable files in the user-root.

Table II shows a few example directory entries of the home-directory of Susan. All values printed in italics are secured by the hash-tree, all other values are not part of the hash-tree.

B. Group access

As we already explained in section II-B, each group has a group-root, just like the user-roots for the user directories. The group-roots are contained in the *.groups* directory which is rooted at the top-level. It works in the same way the *.users* directory works: it is protected by a Merkle-tree and contains all the files and directories that belong to a certain group. The *.groups* directory is split into two subdirectories, named *public* and *private*.

Figure 2 shows a simple example for the directory structure containing this group directory. Note that the file system contains the directory entry for the directory *Documents* three times: The first entry is located in the *.users* directory. This is the "master" copy of the directory. It works as presented in section III-A. Two further directory entries are created in the *.groups* directory, one in the public and one in the private part.

Files are treated similarly. Each file entry is stored at two out of three possible locations: There is always a copy in the user-root directory. For each file that is world-readable, an additional copy is placed in the *public* subdirectory of the group. If the file is not world-readable the copy is placed in the *private* subdirectory of the group. A group-member will only be able to change the version in the group-directory, not the entry in the user-root, which is protected by the hash tree of the owner.

Because of this, when a file in a user-directory is requested, any of the two copies can be more up-to-date. The file system has to look up the entry in the user-directory as well as the entry in the group-directory to find out, which is the more recent one.

Each group has an encryption key, which is used to secure the private group directory. This key is only known to the current members of the group. Thus it has to be changed when users join or leave the group.

The key is stored in the file system using the *subset difference* algorithm [11]. It allows storing an item in a way that enables groups of users to decrypt it with very little overhead. This scheme requires storage costs of $\frac{1}{2}\log^2 N$ for each user (where N is the total number of users). The message length is at most $O(g)$, where g is the number of users in the group, but it can be much better under good circumstances. The user only has to execute a single decryption.

Each time, the superuser adds or revokes a user from the system, the file system superuser also generates a new group key. This key is secured with the subset difference algorithm and then stored in the file system.

The pointer of the *private* subdirectory is encrypted with the group key; only this directory pointer is encrypted, all subdirectories are not encrypted. That means, that only a member of the group, who knows the current key, can enter the group's private directory.

All files, that are not world-readable are placed in the *private* subdirectory of the group directory.

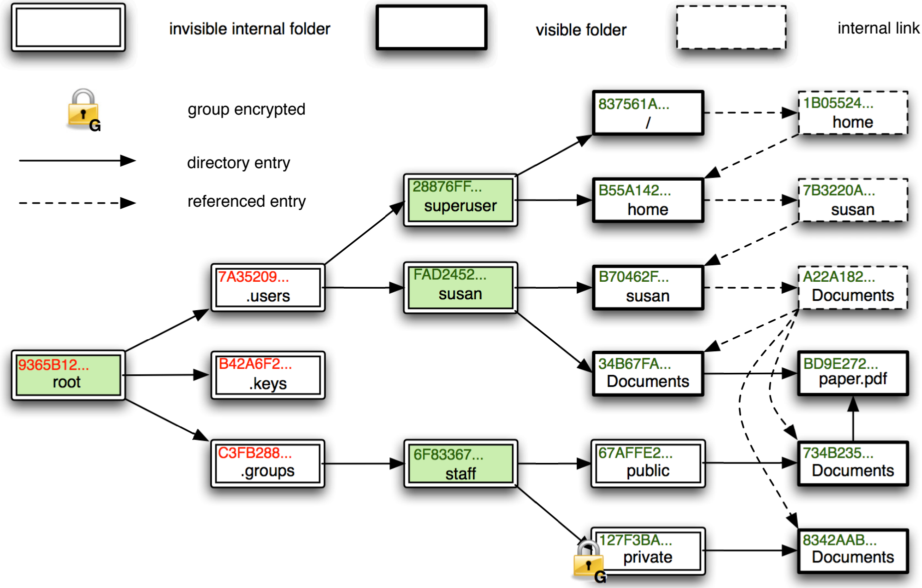

Figure 2. Internal directory structure

Type	Name	Owner	Group	Permissions	Version	Location	Forward Key	Hash
Redirect	*Documents*	*susan*	*staff*	`rwr-r-`	12	(Id, Key)	B2A8399...	*A22A182...*
File	*Secret*	*susan*	*staff*	`rw----`	5	*E(Id, Key)*		
File	*Calender*	*susan*	*staff*	`rwr-r-`	6	*(Id, Key)*		
File	*Share*	*susan*	*staff*	`rwrwrw`	95	(Id, Key)		

Table II
EXAMPLE DIRECTORY ENTRIES FOR `/.users/susan/susan/`

C. Forward Keys

In Unix systems, a file or directory may only be read, if the user has access to all directories along the path up to and including the object in question.

Figure 3 illustrates the problem with an example: The *home*-directory contains a directory named *susan*, which is private to Susan. It contains a directory named *Documents* which happens to be public. In a normal Unix system, we would not be able to access this directory, because we cannot access the parent directory. In the system described so far we could circumvent the encryption and access the *Documents* directory directly.

To prevent such an access via the hidden directory structure, we add a directory key K_d to each directory in the structure. This key is used to encrypt the directory contents. It is stored in the visible parent-directory. Because each directory (except for the root directory) has got exactly one parent directory, there is exactly one key for each directory. In our example, the directory *susan* is encrypted with the directory key of the *home* directory $K_d(home)$, and with the user-key. *Documents* is encrypted with the K_d*(susan)* key.

Thus a user now can only access the *Documents* directory if she can access the *susan* directory that contains K_d(susan). K_d is changed on each write operation to a directory to prevent users who may no longer access a directory from gaining unauthorized access.

V. Communication Overhead and Cryptographic Cost Analysis

In this section, we briefly describe the communication overhead and cryptographic cost that our proposal adds to a plain distributed file system. As mentioned in section I, our work was inspired by the goal to equip an existing P2P file system with user and group access permissions. Thus, our design is such that it can be implemented on top of a plain P2P file system. Each file system operation, both, in the plain and in the so enhanced file system causes one or more operations in the underlying P2P network. In the following, we discuss the number of these operations as well as the associated cryptographic cost. We especially discuss the particular case of the P2P file system that has been developed in our group. Other P2P file systems would lead to very similar results.

In the plain scenario, upon each file lookup, the file system has to fetch all the directory blocks, starting from the root block down to the directory in question. The number of operations in the P2P network thus depends on the length of the path we are trying to access. For a file in a subdirectory at depth n, we need exactly n directory lookups.

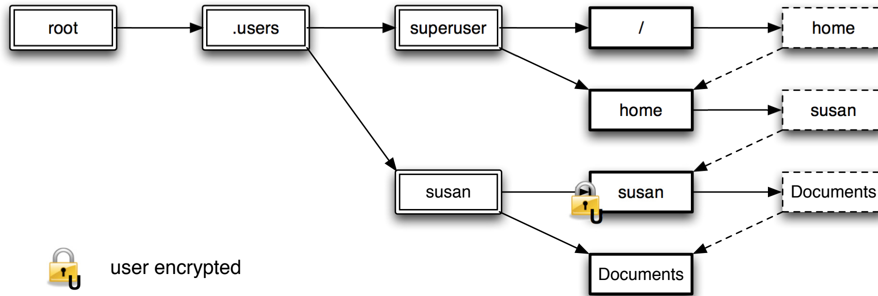

Figure 3. Evading access restrictions without forward keys

In our enhanced file system the communication cost depends on who owns the directories and files. Assume first a directory structure where all directories and files belong to the superuser. Here, we need $n+3$ directory lookups, because there are three hidden directories (*root*, *.users*, *superuser*), which we must read before we can read the first visible directory.

The internal links do not account to the communication cost, as long as all the directories of a user are stored in the user's hidden directory. (E. g. '/' and 'home' are both stored in the superuser's hidden directory.) This is the $d = 1$ case that we used for the examples in this paper. If a user owns a large number of directories, we need to introduce one or more indirections, i. e. further hidden directories below the users' directories that contain the visible directories (cf. fig. 4 with $d = 2$). When taking this into account, we need at most $3+nd$ lookups when all files belong to the superuser.

In a file system with multiple users, the communication cost will be larger, because when we encounter a directory that belongs to another user, we have to resolve an indirection. Let u be the number of users (including the superuser) that own directories in the path to our file. Then the communication cost is at most $2 + u + nd$ directory lookups: we have to read the *root* and *.users* directory, the user-root of each affected user and the n directories we traverse at depth d in the different user-roots.

When considering the presence of groups the communication costs are even larger; when looking up a file, we have to check if the version in the group or in the user-root is more current. That means we have one additional lookup into the group-root of the affected group with a cost of $3 + d$ lookups (one lookup for *.groups*, one for the group, one for the *public* or *private* group directory and d for the specific subdirectory) incurring a total cost of $5 + u + (n + 1)d$. In practice, many of the affected entries can be cached, so that these numbers should be considered as worst-case analysis.

In addition to the communication cost, our proposal also incurs an additional cryptographic cost. We assume that in the plain scenario all directories and files are already cryptographically protected. The file system, that we use as basis for our work, for example, encrypts all directories and files with a symmetric key that is stored in the directory pointing to that file.

Besides this basic cryptographic cost, there is an additional cost:

- Encryption and decryption of the directories with the user or group key.
- Creation and validation of the signed hash trees for the user and group directories.

Nevertheless, the absolute overhead of this cost is small: Hashing is a relatively cheap operation. The underlying file system already hashes all its file system blocks twice (plain text and cipher text). Our proposal adds a third hash operation.

Moreover, our proposal only uses symmetric cryptography. The underlying file system already encrypts all blocks with AES. If a file is not world-readable, our proposal adds two further AES encryptions per file, and one further AES encryption per directory.

Only the signature verification of the hash tree adds a more substantial cryptographic cost. However, this verification is a rare operation: We only have to verify one signature per user-root directory that we traverse, and we can cache the result for subsequent operations.

VI. Further extensions

In this section, we will discuss possible extensions to the approach outlined in this paper. These extensions are not necessary for the basic functionality, but provide additional features that could be important in some use-cases.

A. Differences to POSIX

Our approach, as we have presented it here, implements parts of the POSIX.1 access permission features. There are however a few differences, which we describe in this section.

1. In our current approach we have assumed, that a person, that may write to a file can also read it. This does not have to be true; in Unix there can be files that a user can only write, but not read. Our current approach does not deal with this scenario. If necessary it could be implemented as follows: In addition to the symmetric key each user and group needs a set of asymmetric keys. The public keys are published in the file system. If a user wants to write to a file, to which she has only write access, she uses the public key of the user or the

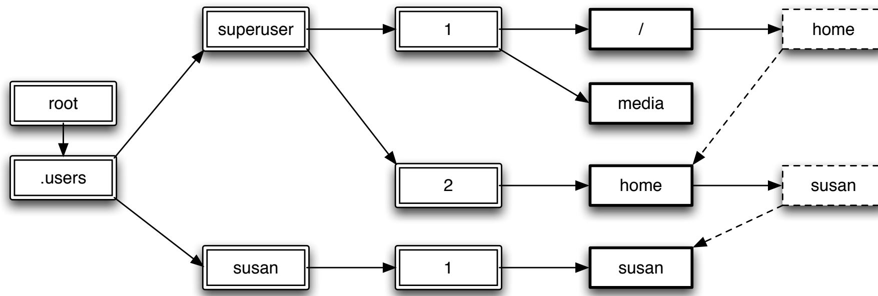

Figure 4. Internal directory structure with $d = 2$

group to encrypt the new file pointers. After that, she will not be able to read the file contents via the file system, because she is not able to decrypt the pointers herself. The group or owner of the file however will be able to decrypt the pointers with their private keys.

The asymmetric cryptography that is needed for this feature introduces a high cost and complexity. It is thus advisable to only implement this feature if it is really needed. Moreover, the mechanism also introduces some backward confidentiality problems when it is used with groups: The symmetric key of the groups can be changed each time a user is added to or removed from the group, because it encrypts only a single directory pointer (the pointer to the *private* group directory). However, files encrypted with the public key of the group could potentially be spread throughout the whole group directory tree. Changing the asymmetric key upon each group change would thus take very long (each file would have to be found and re-encrypted). But if the key is not changed, each new member of the group can access every file that was encrypted with public key cryptography. This includes files that have been deleted before the user was added to the group.

2. In addition to the read and write permissions for files and directories presented in this paper, Unix also provides the execute flag. A person that only has a separate execute flag on a file may run the program but may not read the binary program data. A person that only has the execute flag on a directory may access the directory and any subdirectories and files in the directory she has access to. The person may however not list the directory contents. She has to know the exact name of the object she wants to access.

We did not try to implement that features. For files, this is due to the simple necessity that in a distributed setting a node must be able to read a file before being able to execute it. For directories the problem is more serious. It might be possible to encrypt a directory in a way that makes it possible to access the files and directories contained within it only when the exact file names of the items are known. We feel however that this is out of scope for our current paper and leave this as an open question.

3. A third shortcoming of our approach is that a user has to be in the group the file belongs to. This is a sound assumption, because in Unix a user may only change the group of a file she owns to a group of which she is a member. Only the superuser can override this restriction.

This problem can once again been circumvented by using public key cryptography; a file could be encrypted with a group-public key in addition to the key of the file owner. This approach introduces the same backwards-confidentiality problems mentioned above.

4. POSIX defines some special file permissions: to be more exact the set-user-ID-on-execution bit, the set-group-ID-on-execution bit and the sticky bit. The setuid and setgid bits are easy to implement; they can be secured with the hash-trees just like the rest of the data in the file system. The problem is the user- and group- mapping from IDs on the global network file system to IDs on the local system. We feel that the question of ID mapping is out of scope for the current paper and leave it as an open question.

Today, the sticky bit is only important when set on directories. A directory with a sticky bit set becomes append-only, or to be more exact a directory in which the deletion of files is only allowed by the superuser, the owner of the directory or the owner of the file. It is not sufficient to have write access to the directory to remove a file from it. Implementing the sticky bit is not an easy task; we again leave it as an open question.

5. Another special case of current POSIX file systems has been disregarded altogether in this approach, because we consider it to be of very low importance. Moreover, in our opinion, it introduces a privacy problem. In a traditional file system, if a directory owned by A contains a directory owned by B and A has no write-access to B, A may delete the subdirectory of B, if it is empty. This even holds true, if A may not read or access the directory in question. We chose not to implement this special case because of the privacy problems that it could cause. We do not want the user A to be able to notice that a directory of B is empty, when she does not have read access to it.

B. ACL support

The approach presented in this paper does not support access control lists.

We will now present a method to implement ACL support on top of our proposed design. The ACL support presented

here is based on the ACLs used in Linux, which are based on a POSIX draft [12].

The drawback of ACL support is an increased amount of cryptographic and lookup operations. On each change of a file or directory, the affected pointers have to be encrypted for each user and group that has read-permission. This means that the encryption time rises linearly with the number of ACL entries. The same applies to the lookup overhead; we have to look up the directories of every user and every group that has write-access to the data. The approach also introduces backwards-confidentiality problems. Hence, we recommend using it only when needed.

ACLs allow users to set more fine-grained access permissions on their files and directories. In more detail, it allows the file owner to give any other user or group read, write or execute permissions.

To support ACLs with our system we have to modify our encryption system to allow for these possibilities. Each file or directory has its own symmetric key, which encrypts its pointers. This symmetric key has to be made available to all users and groups with read-permissions by encrypting it with the user's or group's public key. For groups this means, that all files with ACL entries are saved in the *public* group directory.

A second problem that arises with groups is that the asymmetric key pair has to change with each user revocation. We propose using the key rotation scheme introduced in Plutus [13] for this purpose. Using this scheme, the file system superuser generates a new symmetric encryption key k_i for the group. Current group members can derive all previous key values k_{i-j} from the new key; revoked members however cannot deduce the value. The group key is saved in the file system using the approach presented in section IV-B.

For each group revision a new asymmetric key pair has to be created. The private key is encrypted with k_i and stored together with the unencrypted public key in a readily accessible location on the file system. Thus, all group members with a valid current key can determine all previous symmetric and asymmetric keys.

This approach has the same backwards confidentiality problems we have previously discussed in section VI-A. E. g. new group members will be able to read the content of files that were deleted before they joined the group.

When a user other than the owner or a group changes the file, the new version is saved in the user's own user- or group-root. When reading a file, the user- and group-roots of all users and groups that may write to the file have to be searched for most recent version. This constitutes a significant overhead. Hence we recommend to use ACLs only when needed.

VII. Related work

Most fully decentralized file systems do not have any kind of user management; many of them allow anyone to read the data in the file system; typically only a limited group of people is able to modify the data. This is especially true for many of the early systems. For example, CFS [14] uses a file system layout that is inspired by the directory service proposed by Fu et al. [15], where only one person can write to the file system, and everyone who has read access can read all data in the system. Nevertheless, the file system layout used in CFS has got certain similarities to our approach.

An example for a file system that integrates user management is the Ivy file system [16]. Ivy is a log-based file system, where each log entry is signed by a user. It is possible to exclude a malicious user's changes from the file system by not accepting log files. Other than that, there is no user or group management; each user may read or write every file in the file system.

The Pastis file system [17] uses certificates on a per-file basis, which does not scale very well. Also, Pastis does not provide read-protection of any kind.

SiRiUS [18] implements a read write cryptographic access control for file level sharing, which is layered on top of an insecure network file system, e. g. a peer-to-peer file system. Unlike our system, SiRiUS does not provide support for groups of users; instead, access to a specific file can be given to a set of users manually. SiRiUS does not try to provide a unified file system to a set of users, instead each user owns a separate file system. These systems can be combined with union-mounts, but it then requires searching the file system of every user that has access to a specific file for the most recent version. SiRiUS is in some settings prone to certain kinds of rollback attacks. Unlike our system, SiRiUS requires the use of asymmetric cryptography.

Secure Network Attached Disks (SNAD) [19] and the Secure Untrusted Data Repository (SUNDR) [8] both require some kind of server support to enforce security.

The Keso File System [20] uses public key cryptography to secure directories. Unfortunately, it partially relies on other nodes to be online to verify the validity of newly inserted information. The node that is responsible for the ID at which the new version of a directory is saved, has to verify that the changes to the directory are valid. If this node is malicious, parts of directories can be tampered without anyone else noticing, unless they retrieve the whole change history for a specific directory.

Plutus [13] introduces a scheme that enables users to share files for reading or writing in a cryptographically secure way. Unlike our approach, however, Plutus does neither handle groups, nor offer protection against rollback attacks. It also depends on asymmetric cryptography.

The Farsite distributed file system [21] uses an approach mentioned in [22]. Farsite uses an encryption process that enables other users to verify the validity of the directory entries without decrypting the data block. Unfortunately, this approach also depends on a central entity. This so-called *directory group* consists of several clients in the file system that are trusted to a certain extent. A directory group decides if a write request is legitimate and may be committed to storage.

Another notable approach is Wuala [23]. It uses cryptrees [24] to securely encrypt files and directories and manage the access to these items. But the cryptree approach has some

distinct disadvantages; it assumes that a person that can read a directory can read any subdirectories of this directory too. The same problem also applies to write operations.

Identity based encryption (IBE) was first proposed in 1984 [25], the first usable implementation was not available until 2001 [26]. This work has been further extended recently by different authors, e.g. to allow attribute based encryption (ABE) of objects [27].

In principle ABE and IBE schemes could be used to implement ACLs in a distributed systems. To allow some arbitrary user or group to access a file, we would just have to encrypt the current file keys with the user- or group-ID.

However there are some serious problems when trying to use this approach with groups: there is no efficient way to revoke the key of only a member of a group; IBE and ABE also are both asymmetric cryptosystems.

VIII. Conclusion and Outlook

In this paper we have proposed a new file system structure to implement user and group permissions in peer-to-peer file systems. To the best of our knowledge, our proposal is the first that provides cryptographically enforced Unix-like access permissions in a fully decentralized manner. Our design is based on a hidden file system structure, which adds the required meta-information to a plain peer-to-peer file system.

Our proposed technique consists of two tightly coupled mechanisms: an integrity verification algorithm checks the validity of the file system state upon access; a cryptographic data protection scheme preserves the privacy of the file system's content. With their help, we can handle user and group access, as well as most of the POSIX specialties, e. g. file system subtrees with mixed ownership. We discussed where and why our proposal deviates from the POSIX standard. We also described the communication overhead and the cryptographic overhead that our system adds to a plain peer-to-peer file system. We further examined how ACL support can be added on top of our scheme.

Currently, we are about to release an open source implementation of our proposal. We expect that many interesting questions arise from the practical use of our system. It would e. g. be interesting to analyze the overhead that our system incurs in a real-world setting. Such an analysis could indicate further optimizations to our design.

References

[1] "IEEE Std 1003.1, 2004 Edition. The Open Group Technical Standard Base Specifications, Issue 6. Includes IEEE Std 1003.1-2001, IEEE Std 1003.1-2001/Cor 1-2002 and IEEE Std 1003.1-2001/Cor 2-2004," in *Standard for Information Technology - Portable Operating System Interface (POSIX). Shell and Utilities*. IEEE, 2004.

[2] K. Kutzner, "The Decentralized File System Igor-FS as an Application for Overlay Networks," Ph.D. Thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, 2008.

[3] J. Amann, B. Elser, Y. Houri, and T. Fuhrmann, "IgorFs: A Distributed P2P File System," in *Proc. Eigth IEEE Int. Conf. Peer-to-Peer Computing (P2P'08)*. IEEE Computer Society, 2008, pp. 77 – 78.

[4] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica, "Towards a Common API for Structured Peer-to-Peer Overlays," in *Proc. 2nd Int. Workshop on Peer-to-Peer Syst. (IPTPS '03)*, Berkeley, CA, USA, 2003.

[5] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proc. ACM SIGCOMM '01*, 2001, pp. 149–160.

[6] "File System in Userspace," *http://fuse.sourceforge.net*.

[7] D. Mazières and D. Shasha, "Building Secure File Systems out of Byzantine Storage," in *PODC '02: Proc. twenty-first annual symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2002, pp. 108–117.

[8] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure Untrusted Data Repository (SUNDR)," in *OSDI'04: Proc. 6th Symp. on Op. Sys. Design & Impl.*, Berkeley, CA, USA, 2004, pp. 121–136.

[9] R. C. Merkle, "Secrecy, Authentication, and Public Key Systems." Ph.D. dissertation, Stanford, CA, USA, 1979.

[10] ——, "A Digital Signature Based on a Conventional Encryption Function," in *CRYPTO '87: A Conf. Theory and Applications of Cryptographic Techniques on Advances in Cryptology*. London, UK: Springer-Verlag, 1988, pp. 369–378.

[11] D. Naor, M. Naor, and J. Lotspiech, "Revocation and Tracing Schemes for Stateless Receivers," in *Adv. in Cryptology - CRYPTO 2001: 21st Annual International Cryptology Conf.*, ser. LNCS, vol. 2139. Springer Berlin / Heidelberg, 2001, pp. 41–62.

[12] "Posix 1003.1e / 1003.2c Draft Standard 17 (withdrawn)." IEEE, 1997.

[13] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *FAST '03: Proc. of the 2nd USENIX Conf. on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2003, pp. 29–42.

[14] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *Proc. 18th ACM Symp. Oper. Sys. Princ. (SOSP '01)*, Lake Louise, Banff, Canada, Oct. 2001.

[15] K. Fu, M. F. Kaashoek, and D. Mazières, "Fast and Secure Distributed Read-Only File System," in *OSDI'00: Proc. 4th conf. on Symp. on Oper. Sys. Design & Impl.*, Berkeley, CA, USA, 2000.

[16] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A Read/Write Peer-to-Peer File System," in *OSDI*, 2002, pp. 31–44.

[17] J.-M. Busca, F. Picconi, and P. Sens, "Pastis: A Highly-Scalable Multi-user Peer-to-Peer File System," in *Euro-Par 2005 Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 3648/2005. Springer Berlin / Heidelberg, 2005, pp. 1173–1182.

[18] E. jin Goh, H. Shacham, N. Modadugu, and D. Boneh, "Sirius: Securing Remote Untrusted Storage," in *Proc. Network and Distributed Systems Security (NDSS) Symp. 2003*, 2003, pp. 131–145.

[19] E. Miller, D. Long, W. Freeman, and B. Reed, "Strong Security for Distributed File Systems," in *Proc. 20th IEEE Int. Performance, Computing, and Communications Conference*, 2002, pp. 34–40.

[20] M. Amnefelt and J. Svenningsson, "Keso - A Scalable, Reliable and Secure Read/Write Peer-to-Peer File System," Master's thesis, KTH/Royal Institute of Technology, Stockholm, May 2004.

[21] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 1–14, 2002.

[22] J. R. Douceur, A. Adya, J. Benaloh, W. J. Bolosky, and G. Yuval, "A Secure Directory Service based on Exclusive Encryption," in *ACSAC '02: Proc. 18th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2002.

[23] "Wuala," *http://wua.la*.

[24] D. Grolimund, L. Meisser, S. Schmid, and R. Wattenhofer, "Cryptree: A Folder Tree Structure for Cryptographic File Systems," in *SRDS '06: Proc. of the 25th IEEE Symp. on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 189–198.

[25] A. Shamir, "Identity-Based Cryptosystems and Signature Schemes," in *Proc. of CRYPTO 84 on Advances in cryptology*. New York, NY, USA: Springer-Verlag New York, Inc., 1985, pp. 47–53.

[26] D. Boneh and M. K. Franklin, "Identity-Based Encryption from the Weil Pairing," in *CRYPTO '01: Proc. 21st Ann. Int. Crypt. Conf. on Adv. in Crypt.* London, UK: Springer-Verlag, 2001, pp. 213–229.

[27] V. Goyal, O. Pandey, A. Sahai, and B. Waters, "Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data," in *CCS '06: Proc. 3th ACM conf. on Computer and communications security*. New York, NY, USA: ACM, 2006, pp. 89–98.