Adding Cryptographically Enforced Permissions to Fully Decentralized File Systems

Bernhard Amann, Thomas Fuhrmann

TUM-I1107
April 11

Adding Cryptographically Enforced Permissions to Fully Decentralized File Systems

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

April 6, 2011

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 report, 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.

Contents

1 Introduction

Files are important – for software systems as well as 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 their files. Or they use version control systems and web-based systems to enable the team members to access their files from everywhere.

Typically, network file systems rely on centralized components and 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 technical report, we address a yet unsolved problem, namely how to realize Unix-like access permissions in P2P file systems.

1.1 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 between peers in the network, because there are no trusted entities 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 share 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 technical report, we describe a new file system structure. With this structure we can implement nearly the full spectrum of access rights that Unix users are accustomed to in a fully decentralized manner. 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.

1.2 Access permissions

A distributed file system should be as transparent for its users as possible. Thus, we want to support access permissions that are modeled closely to the POSIX.1 [40] permissions which are used in Linux as well as many other systems.

In more detail, we want to be able to have 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 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 Chapter 9, we discuss other access permissions often found in Unix-like systems and how they could be implemented using our approach. We also sketch a way to implement ACLs on top of our proposed access permission system.

1.3 Design criteria

Our goal is to create a fully decentralized distributed file system (DFS) that 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 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.

1.4 Contribution

In this technical report, we further detail our work on cryptographically enforced access rights for fully decentralized file systems, which was first presented in [6]. There we have proposed a method 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.

While all the basic ideas were already presented in [6], this technical report gives much more details about the actual way the access rights can be implemented in a real system. It details special cases that are not obvious on a first glance, presents the way user-management works in more detail and describes the implementation details of the file system operations.

This technical report is organized as follows: In Chapter 2 we give an introduction to the technical background of our work. Chapter 3 discusses the related work on this subject. It discusses the underlying network structures as well as the cryptographic algorithms we use. In Chapter 4 we briefly describe our idea and introduce the technical background on which we base our work. Chapter 5 outlines our approach for validating the data integrity in a distributed file system. Chapter 6 describes the handling of file access rights. Chapter 7 shows how the different file system operations are implemented in practice. Chapter 8 estimates the overhead that our proposal imposes. Chapter 9 explains how to extend our approach to more sophisticated access control mechanisms. Finally, Chapter 10 gives the conclusions and an outlook to future work.

2 Technical background

The cryptographic approach presented in this technical report can be implemented on top of any kind of block based distributed storage system. In our research group we have developed such a system, namely a fully decentralized file system, the Igor File System (IgorFs) [47, 5].

In the remainder of this report we assume that our cryptographic approach is implemented on the top of IgorFs. It is thus important to know a few technical facts about the Igor file system.

2.1 Design Goals

The origin of IgorFs can be found in the area of bioinformatics. Meanwhile it's also applied in high energy physics.

In bioinformatics, there are many large databases containing information about genes, proteins, molecules, etc. These databases can reach sizes of several terrabytes with millions of files. The databases change daily - however only a very small amount of data changes at once. One example for such a database is the worldwide protein data bank [81].

Research applications e.g. analysis software and simulations need access to the current version of the databases. However, they typically only access very small parts of the databases.

Usually, sites set up a local mirror to allow the software and researchers to access the current data. However, this is very wasteful because the mirror has to contain the whole huge research database and has to sync itself regularly to the master copy.

The best solution would be a file system that only contains the data needed for a specific analysis or simulation. To this end, it only needs to download the deltas for that specific data, when a new version of the database is available.

The result of our research into these problems was IgorFs, a shared distributed file system with a single writer for each dedicated file system tree. The only information needed to access an IgorFs file system is the identification and the cryptographic key of the file system root block. In IgorFs, a local node only downloads the information from the network, which is requested by the applications accessing the file system. All downloaded information is automatically cached. When a new version of a file system tree is generated, only the changed parts of files that are accessed are downloaded from the network.

IgorFs also features an automatic notification system for new version - clients are aware of newly inserted data very quickly after it has been made available to the network.

The file system also encrypts all transmitted data. It offers an interface to an external user-management system e.g. to handle subscriptions. Hence it is possible to give paid access to institutions and to revoke the access later.

The design criteria of IgorFs make it suitable for a large field of applications. For example in the area of high-energy physics the CERN is currently optimizing the distribution of their research software to the associated computing centers. Just like the biology databases, their software packages are huge and include many libraries, of which only a small number is used depending on the exact simulation parameters. The software repository changes up to a few times per week and it is important that new versions are quickly available at the computing centers after they have been published.

At the moment each computing center has it's own software repository that is updated separately from each other. Because only a single server per computing center is hosting all the software, there are performance bottlenecks when many simulations start at the same time. IgorFs also suits this scenario and could significantly decrease the administrative overhead and increase the performance at the same time.

Our research group is collaborating with the CERN in this field.

2.2 Igor

IgorFs is based on a structured P2P overlay network [51] called Igor, which provides a *key based routing service* [21] similar to Chord [74], Kademlia [52], Pastry [69], and others.

Unlike a distributed hash table (DHT) [78], which offers a publish/subscribe service only, Igor is a service oriented overlay network. It routes messages based on a destination key and a service identifier [22].

Applications that want to take part first connect to an Igor node. Usually the Igor node and the application reside on the same physical machine. The application submits its service identifier to the Igor node. After that the Igor node will receive messages that are sent to this service identifier. Nodes that do not run a service will not receive messages for that specific service.

Igor exports a very simple API which is similar to the well-known Berkeley socket API [75, sec. 6.1.3]. An Igor socket has to be bound first, after that messages can be sent and received. At the end of the session the socket is unbound and closed.

Igor uses Proximity Route Selection (PRS) and Proximity Neighbor Selection (PNS) [35, 43, 27] to exploit the proximity of nodes in the underlying Internet topology. Thereby, services can easily benefit from local nodes running the same service.

2.3 IgorFs

IgorFs is a fully decentralized distributed file system. The different IgorFs nodes use the Igor overlay network to communicate with each other. All nodes are coequal. There are no nodes with special properties or special trust in each other. Most importantly there are no centralized nodes because they would introduce a single point of failure and a performance bottleneck.

In IgorFs, a File System is identified by a unique file system root block. Every node can create a new file system by generating a new root block. To access an existing file system, a node needs to be able to access its root block, i.e. it needs the block identifier and cryptographic key. This will be explained in detail in Section 2.3.2.

Every node can create its own IgorFs file system. It can also open other IgorFs file systems if it has the corresponding key material.

IgorFs is implemented as a process network [41]. Independent modules communicate solely via messages.

The most important modules of IgorFs are:

- The *Application Interface* module (i.e. the FuseInterface or NfsInterface) receives operating system calls, converts them into IgorFs calls and sends them to the File and Folder Module.
- The *File and Folder Module* handles all file and directory operations. All file system requests are handled here. The data needed to satisfy the operations is read or written with the BlockTransfer Module.
- The *BlockTransfer Module* sends and retrieves data chunks via the BlockCache.
- The *BlockCache Module* handles requests for data chunks. If a chunk is found in the cache it is returned, otherwise it is fetched via the BlockFetcher.
- The *BlockFetcher Module* transfers modules from remote nodes to the local node when they are requested.
- The *PointerCache Module* is a DHT-like system, which stores the information about which blocks are stored on which nodes.
- The *IgorInterface Module* is responsible for the communication with the Igor daemons. All inbound and outbound messages pass through this module.

The functionality of these different modules will be described in more detail in the following sections.

Figure 2.1 shows the modules and which modules communicate with each other. In the figure, cornered blocks are IgorFs modules; round blocks are external components.

2.3.1 Application interface

There are two different ways to attach IgorFs to the operating system. The preferred option is to mount IgorFs into the Unix directory tree, just like any other kind of file system. In Unix, the kernel is responsible for handling the file system calls made by applications. Calls are handled by the part of the kernel responsible for the respective file system. However, implementing file systems in the operating system kernel is a complex task.

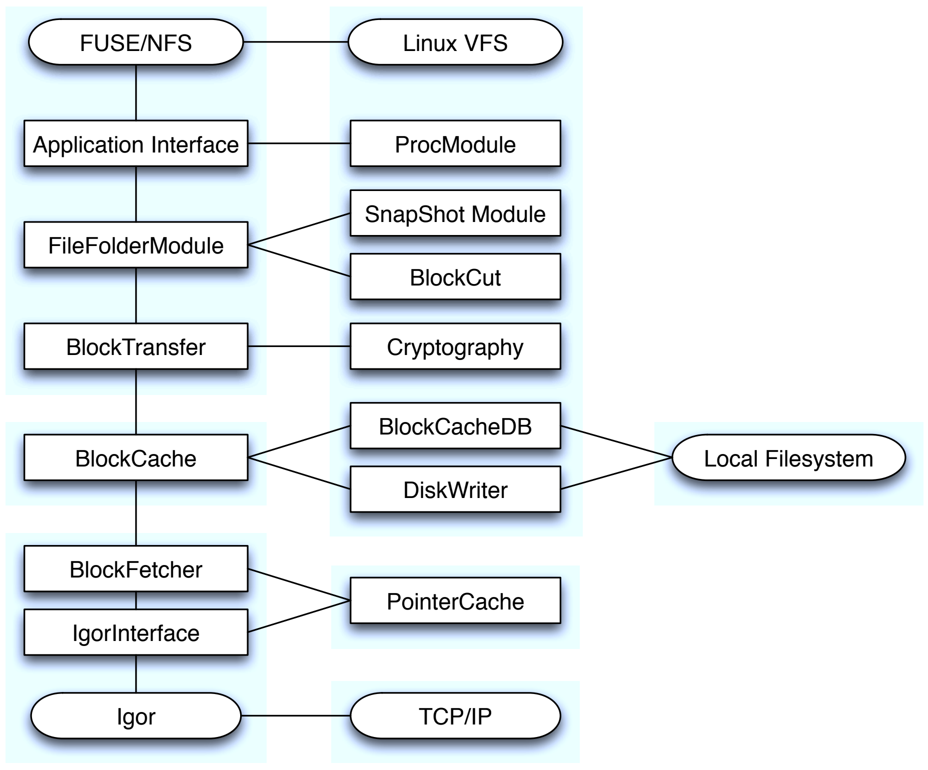

Figure 2.1: Module communication in IgorFs [adapted from 47]

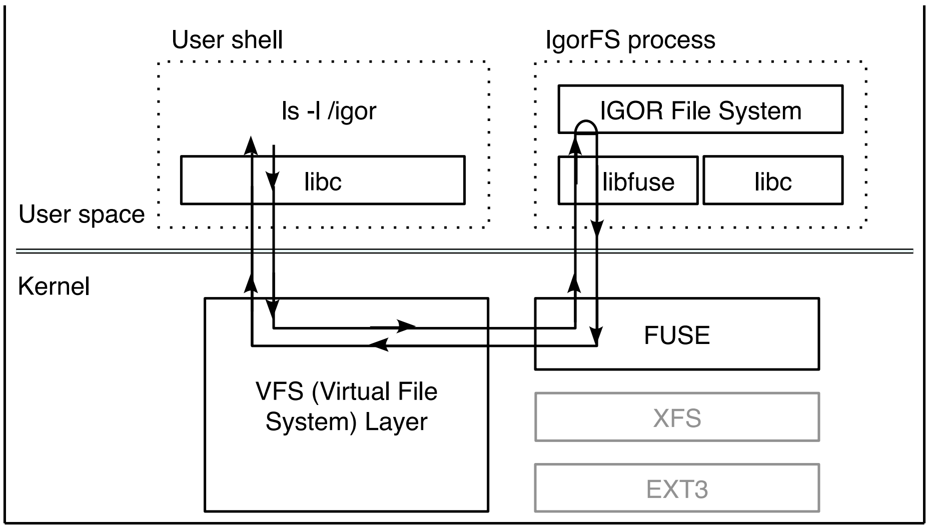

Figure 2.2: FUSE sample request [adapted from 77]

With Fuse (Filesystem in Userspace) [30], it is possible to implement file systems completely in the userspace. Fuse works by using a kernel module that re-routes requests from the kernel's VFS layer to a userspace program. Figure 2.2 shows this approach in more detail. With this approach, IgorFs looks just like any any other local file system to the user.

The second way to attach IgorFs to the operating system is via NFSv3. IgorFs can re-export the current IgorFs file system via NFS. A client can then mount the IgorFs file system via its local NFS client software. This approach is more cross platform compatible, but has a lower performance and misses some advanced features such as extended attributes.

2.3.2 File storage

All files that are stored in IgorFs are first cut into variably sized chunks. The points at which a file is split into different chunks is determined using a rolling checksum algorithm [see 48, sec. 3.2]. As a result, modifications to parts of a file affect only a few chunks in the vicinity of the modified parts.

Each data block B in IgorFs is identified by a 256 bit wide identifier I which is assumed to be globally unique. Each block is encrypted with its own 256 bit key k. Both, the Id I and the key k are obtained by hashing the block with a cryptographic hash function H. $k = H(B)$ is obtained by hashing the the original data. $I = H(E_{H(B)}(B))$ is obtained by hashing the encrypted data block. E denotes the encryption function. Thus, independent IgorFs instances map the same clear text block to the same cipher text block. Blocks with identical content have the same identifier I. Thus, if two nodes insert two files into IgorFs that have shared areas, these shared areas will be mapped to the same encrypted data blocks. This is similar to the implementation of *Content Hash Keys* in Freenet [18].

In order to be able to read a block, an IgorFs instance needs to have the block's (Id, key)-tuple.

2.3.3 Directories

Directories are handled similarly to files. A directory consists of a list of file names, file properties and the (Id, key) tuples for the file chunks. This list is serialized into a data block, which is stored in IgorFs just like an ordinary file.

Thus, when a node can read a directory it can recursively determine all the (Id, key) tuples for all files and subdirectories stored in the directory. When a node can read the root-directory it can recursively read the contents of the whole file system.

Multiple, separated file systems can share the network and the attached storage capacity. They just use different root-blocks.

2.3.4 Block transfer and caching

In IgorFs, data chunks are not stored directly in the DHT. Instead, IgorFs uses an indirect approach. When a new chunk C is inserted, it is stored on the local node only. The information from which node the block can be retrieved is then inserted into the PointerCache (see Section 2.3).

When a node requests a chunk, it first asks its local pointer cache for the location of the data block. The local pointer cache either knows the location or sends a request to the remote pointer cache that is responsible for the Id area of the request.

After the pointer cache returns the data location, the block fetcher module retrieves the actual data and stores it into the local block cache.

2.3.5 File system updates

Whenever a file changes, all directories on the path from the file to the root also change. The reason for this is that whenever a file changes, it also gets a new content hash and hence a new (Id, key)-tuple. This new tuple has to be stored in the directory, which in turn also gets a new (Id, key)-tuple. The changes propagate up in the file system tree until they arrive at the root-directory.

Hence, when a node wants to see the current version of a specific file system, it only needs to know the current (Id, key)-tuple of the root-directory. IgorFs uses a publish/subscribe system to share that key among the peers.

3 Related work

In this Chapter we discuss the related work in the area of distributed file systems. We also briefly discuss the related work in the area of cryptography, which is of interest when considering distributed file systems.

3.1 Distributed file systems

Many distributed file systems have some kind of integrated user management. Unfortunately, many of these systems use assumptions that no longer hold true for fully decentralized distributed file systems. Systems like NFSv3 [16], NFSv4 [72], AFS [38] or NASD [31] either use host-based authentication or third-party authentication like Kerberos [59] to provide security. These systems require the storage server to be trustworthy and furthermore require a central entity for authentication.

In contrast, 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 [20] uses a file system layout that is inspired by the directory service proposed by Fu et al. [28], 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.

There are also systems like e.g. Total Recall [11], which have another focus: They aim at data availability and do not offer any kind of user management, because it is out of their respective scope.

In the following sections, we briefly discuss the most prominent distributed file systems that include some kind of user management. For each system, we also describe the differences to our approach.

3.1.1 Ivy

The Ivy file system [57] is a multi-user read/write peer-to-peer file system without centralized or dedicated components. Ivy is a log-based file system, where each log contains a change set for the file system. Each entry is signed by a user. The logs are stored in the DHash [19] distributed hash table.

A participant finds data by consulting all log entries and performs modifications by appending them to its own log. It is possible to exclude a malicious user's changes from the file system by not accepting their log entries; however this can lead to inconsistencies in the file system that have to be resolved manually.

Other than that, there is no user or group management; each user may read or write every file in the file system. Thus Ivy only has a very basic kind of user-management in comparison to our much broader approach.

3.1.2 OceanStore

OceanStore [46, 12, 66] is a global persistent data store. It has been designed to scale to billions of users. OceanStore assumes that any server in the infrastructure may either crash, leak information or become compromised.

OceanStore supports access control via file-based ACLs. Writes are only committed if a quorum of servers accepts the write operation. Revocation requires data re-encryption. Unlike our system, OceanStore partially relies on trusted servers for writes and for consistency management.

Unfortunately,the description of how access control lists are handled in OceanStore is very vague. Groups are apparently handled, however there is no detailed description of how group cryptography works in OceanStore. It is also assumed that a certain set of nodes is mostly available and reliable. This assumption does not hold in a pure P2P setting.

3.1.3 SiRiUS

SiRiUS [32] is a secure file system that has been designed to be layered over an insecure network and P2P file system. SiRiUS assumes untrusted storage and implements read write cryptographic access control for file level sharing. Access control information is stored together with the file data on the servers. The file is split into a data and a meta data part.

Unlike our system, SiRiUS heavily depends on asymmetric cryptography. Each user maintains a key for asymmetric encryption and a signature key. Each file has a unique symmetric encryption key and a unique signature key. Possession of the encryption key gives read access to the file, possession of both keys read-write access.

Freshness of the file system is guaranteed using a hash-tree. Because of this, each change has to propagate to the root-directory.

In contrast to 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 this approach requires searching the file system of every user that has access to a specific file for the most recent version. SiRiUS is prone to certain kinds of rollback attacks. The freshness guarantees in SiRiUS only apply to the meta data. Hence, data files can be reverted to a previous revision because the previous revision also has a valid signature. It is, however, impossible to revert permission changes on files. Only the file content can be reverted to a previous revision. A method to prevent such rollback attacks is presented in the paper – however it has scalability problems. It has a complexity of $O(n)$ with n being the number of users that can write to the file.

Another problem of SiRiUS is user revocation. When a reader leaves the group all files in the group have to be re-encrypted.

3.1.4 Plutus

Plutus [42] introduces a *key rotation* scheme, which enables users to share files for reading or writing in a cryptographically secure way. Files with the same access rights are grouped in file groups. Keys for a file group are rotated forward by the file group owner using the RSA decrypt operation with a private key. Keys can be rotated backward by users using the RSA encrypt operation and the public key. Every user who possesses the current file group key can determine all the previous file group keys. Hence, the scheme does not require re-encryption of data upon user revocation, but only the owner can generate new versions of this key.

Unlike our approach, however, Plutus can neither handle groups, nor offer protection against rollback attacks. It also depends on asymmetric cryptography.

3.1.5 Celeste and Pacisso

Celeste [9, 17] is a highly-available, ad hoc, distributed, peer-to-peer data store. Celeste itself does not support file level security, however [45] [see also 44] describes a security model named PACISSO, which is layered on top of Celeste.

Unlike our approach, PACISSO needs active online nodes participating in it's protocol to function. An object owner designates a number of gatekeepers, which are responsible for granting read or write access to her data. If enough of these gatekeepers are compromised, they can deny legitimate read accesses or allow unauthorized partys to change the objects under their control (i.e. they can change, but not read any stored data of the object owner).

PACISSO uses a secure version identifier to determine the most current version of an object. This is similar to the way the most recent file system revision is determined in IgorFs.

3.1.6 Other systems

The Pastis file system [15] uses certificates on a per-file basis, which does not scale very well. Also, Pastis does not provide read-protection of any kind [15, p. 1178].

The Distributed Reliable File System [64, 65] is a distributed, decentralized read-write p2p file system. It is based on the TomP2P DHT [76], which uses a Kademlia [52] based routing protocol. It features support for multiple writers, however it does not support any kind of read- or write-protection.

Secure Network Attached Disks (SNAD) [56] requires server support to enforce security. It furthermore depends on timestamps to prevent replay attacks. Timestamps are impractical in P2P settings because clocks are often not synchronized.

The Secure Untrusted Data Repository (SUNDR) [49] also require server support to enforce security. Furthermore, SUNDR only addresses data integrity but not data confidentiality.

The Keso File System [7] uses public key cryptography to secure directories. 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.

Freenet [18] is a P2P network that permits the anonymous publication, replication and retrieval of data. Access control is not a part of the system design.

The Farsite distributed file system [3] uses an encryption process that enables other users to verify the validity of the directory entries without decrypting the data block. This approach is also mentioned in [23]. 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.

Cepheus [29] uses a key server for group key distribution. It also relies heavily on asymmetric cryptography. Furthermore, the design is not a pure P2P design but depends on storage servers.

Another notable approach is Wuala [80]. It uses cryptrees [34] to securely encrypt files and directories and manage the access to these items. But the cryptree approach has some distinct disadvantages because 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.

3.2 Cryptographic algorithms

IgorFs uses both, symmetric and asymmetric cryptography. Asymmetric cryptography is typically only used to sign data. This section will give a short overview over the different cryptographic algorithms that are used in IgorFs. Due to the modular architecture of IgorFs all cryptographic algorithms in IgorFs can easily be replaced by other algorithms when they are e.g. found to be insecure.

For information about alternative cryptographic algorithms, that are not used in this paper but could alternatively be used to implement some of the still missing security features please see Section 9.3.

3.2.1 Hashing Algorithms

IgorFs uses the SHA-1 [24] and SHA-256 [62] algorithms to generate content hashes. At the time of writing, these are among the most widely used hashing algorithms. They are generally accepted to be secure.

3.2.2 Symmetric encryption

We use AES [37, 61] with 128 bit keys as the symmetric encryption algorithm. At the time of writing, AES is the standard symmetric cipher used in nearly all current applications, for example to secure HTTP connections.

3.2.3 Signatures

We use Elliptic Curve Cryptography (ECC) signatures, because they provide approximately the same security as traditional signature algorithms such as RSA [68, 67] or ElGamal [26] while requiring significantly shorter keys.

E.g. to achieve the same level of security as 1024-bit RSA, ECC requires a key size of only about 171-189 bits [79, p. 4]. Furthermore the complexity of the operations is much lower than for traditional algorithms [25, p. 340].

In IgorFs, we chose to use the Curve P-348 [60, p. 32], with a key length of 348 bits. It was specified by the NIST for use by US government entities. We use the Elliptic Curve Digital Signature Algorithm (ECDSA) [2] to generate our signatures.

To sign our data, we always first hash the contents with SHA-1. The resulting hash is then signed with ECDSA.

SHA-256 is not used for the content-hashes for speed reasons - it is significantly slower than SHA-1. SHA-256 is used for the generation of keys for the subset difference algorithm discussed below.

3.2.4 Subset difference revocation

IgorFs requires the encryption of messages for a variable group of participants at several places. New versions of the root-block are for example encrypted in a way that enables only current subscribers to the file system to decrypt it. It should be easy to add and revoke members of existing groups.

To achieve this goal we use the Subset-Difference (SD) revocation scheme[1], which has been proposed by Naor et al. in 2001 [58]. This scheme allows a publisher to send an encrypted message that every authorized receiver can decrypt but none of the revoked receivers can.

The basic scheme proposed by Naor et al. has the following properties: The receivers are stateless, i.e. they do not have to save any information. The scheme requires the receivers to store $\frac{1}{2}\log^2 N$ keys. The message length is at most $2r$ keys, with r being the number of revoked users. It should be much shorter for most cases. The algorithm guarantees complete security, even if all revoked users collude their keys [50]. The computing requirements are $O(1)$ for the receiver and $O(r)$ for the sender.

There are several papers handling extensions to this scheme to make it more efficient for different kinds of operations. Halevy and Shamir [36] e.g. describe a scheme that allows the definition of a subset of users by a nested combination of inclusion and exclusion conditions. The number of messages (or the message length) is proportional to the complexity of the description rather than the size of the subset.

See [4] for a more in-depth explanation of the encryption scheme and it's use in IgorFs.

[1]The algorithm is also e.g. used in BluRay Discs. The encryption system used there is called AACS (Advanced Access Content System) [1].

4 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 an 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 physical machine. 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 2.3. 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 the two following sections of this chapter and then again in more detail in Chapters 5 and 6.

4.1 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 [53] 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 the change 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. It also prevents malicious nodes from only showing selective new files to a client. If a node sees a current file change of another user, it can also verify the current state of the other files owned by this user.

In order establish this system, 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. Figure 5.1). 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. Figure 5.2): 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.

This integrity protection scheme will be explained in more detail in Chapter 5.

4.2 Data confidentiality

As a second step the data integrity architecture introduced in the previous section is extended with 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. Figure 5.2). 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,

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
D_u	symmetric	user key	by superuser on user generation
K_d	symmetric	directory forward key	by user or group on directory write
Id, key	symmetric	data chunk key	by client upon write

Table 4.1: Keys used in the file system

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 Chapter 6.

4.3 Client keys

The cryptographic keys used in our file system are shown in Table 4.1. The cryptographic algorithms used with these keys were discussed in Section 3.2.

Each user has a symmetric user encryption key K_u and an asymmetric user signature key S_u. Both keys are generated by the superuser when the user is first added to the file system. K_u is used to encrypt the user's private data. This key may never be divulged to another node. S_u is used by the user to sign the user's changes to the directory structure. The private portion of this key may never be divulged to another node.

Each group in our file system has a symmetric group key K_g. This key is generated by the superuser when the group is first created. It is regenerated by the superuser upon each change in group membership. This is done to prevent new users from using the cryptographic material to read old data that has already been deleted from the file system. It also prevents old users that have been revoked from the system to read new material written to the file system.

The group signature key S_g is handled accordingly.

Each user has a set of subset difference keys D_u, which is generated by the superuser when the user is first added to the file system. The subset difference algorithm was discussed in Section 3.2.4.

K_g and S_g are encrypted using the subset difference algorithm. The encrypted keys are stored in the file system. The exact storage locations are shown in Section 5.2. Current members of g can use D_u to decrypt K_g and S_g. Thus, every group member can retrieve the current group key directly from the file system.

Each directory has a forward directory key K_d which is a simple symmetric key that is regenerated on each directory write operation. The exact function of this key will be discussed in Section 6.2.

The (Id, key)-tuples for data chunks were discussed in Section 2.3.2.

5 Data integrity

This chapter gives an in-depth explanation or our new internal directory structure which we have briefly introduced in Section 4.1

This structure protects the file system against several types of attacks. Most prominently the file system has to be aware of any unauthorized manipulations of files or directories. If it detects a manipulation, it simply aborts with a read error and output an error message.

Alternatively, we could have chosen to 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. However, 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. Hence we did not choose this alternative.

5.1 Splitting the directory-tree

As described in Section 5, 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 5.1, but internally our file system uses shadow-directories that contain the actual files. Redirects make them visible in their respective directories.

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, which is incremented upon each change.

Because each file has an owner and a group, files are can be present two times in the directory structure. This does not impose a significant storage overhead because only the pointers to the data are stored in the directory structure, the actual data still is present only once on the node.

In order to map the normal Unix directory structure to the new internal structure, we

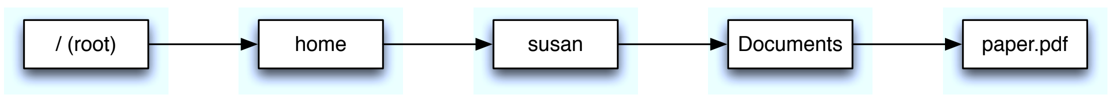

Figure 5.1: Normal Unix directory structure

use redirects. A redirect R is an internal file system construct similar in functionality to a soft link or symbolic link in Unix. Formally, R is an n-tuple, pointing to one or more other objects in the file system, i.e. files, directories or symbolic links. A redirect may not point to another redirect. When encountering a redirect, the file system internally resolves it and returns the referenced data. The redirect is thus invisible to the user of the file system. The resolution of redirects depends on the type of object that is referenced.

Redirects are used in two cases in our file system. For directories, a redirect is typically a 3-tuple[1], pointing to the directory locations in the user-root and in the public and private parts of group-root. When encountering such a redirect, the file system reads the referenced directories, merges their entries, and eliminates duplicates by removing the items with the lower version numbers.

For other objects like files or symbolic links, R is typically a 2-tuple, pointing to the object locations in the user- and group-root. The system returns the item with the higher version number in this case.

5.2 Example directory structure

Figure 5.2 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, which were introduced in Section 4.3.

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 recent. To this end, the version numbers of both files are

[1]See Section 6.6 for all possibilities.

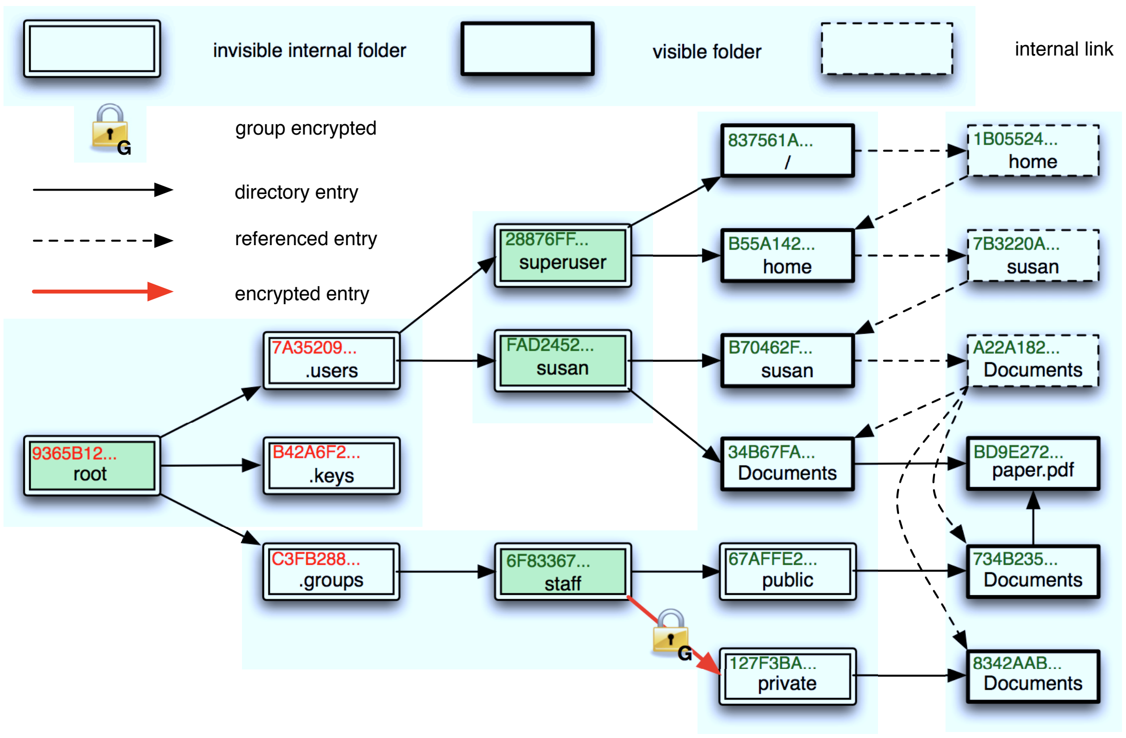

Figure 5.2: Internal directory structure

compared and the more recent pointers are returned. Note we use version numbers not timestamps. Hence we do not require any clock synchronization.

5.3 Securing the user-directories

The state of each user directory is secured with a Merkle-tree [54, 55].

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.

Depending on the access permissions of a file, different bits of information may not be

Type	Name	Owner	Group	Permissions	Version	Location	Forward Key
Redirect	**Documents**	**susan**	**staff**	`rwr-r-`	12	**targets**	B2A8342...
File	**Secret**	**susan**	**staff**	`rw----`	**5**	**E(Id, Key)**	
File	**Calendar**	**susan**	**staff**	`rwr-r-`	**6**	**(Id, Key)**	
File	**Share**	**susan**	**staff**	`rwrwrw`	95	(Id, Key)	
File	**Feedback**	**susan**	**staff**	`rwrw--`	**40**	**E(Id, Key)**	

Table 5.1: Example directory entries for `/.users/susan/susan/`

Attribute	Hashed	Description
Name	Always	The object name
Type	Always	The object type
Device	Always	The object device number (if device)
Mode	Always	The object mode
Owner	Always	The object owner
Group	Always	The object group
Size	Partially	The current object size
Access time	No	The last object access time
Creation time	Always	The object creation time
Modification time	Partially	Time of the last modification
Content pointers	Partially	Pointers to the object contents
Forward Key	No	Directory Forward key (cf. Section 6.2)

Table 5.2: All file attributes and their protection

modified by another user. For example a user without write-access to a directory may not change the name of a file. All attributes that may not been changed are combined into a hash-value (see Table 5.2).

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 user-root directory. In this directory, the hash-value is also computed and then signed with the user's signature key S_u.

When a directory entry changes, the hashes have to be recomputed along the path from the directory to the root (i.e. in all the parent directories of the changed directory). Assume e.g. that Susan changes the file *paper.pdf* (shown in Figure 5.2). Because of this change, the entry in the directory *Documents* changes and thus the hash-value of the documents directory also changes. These changes propagate up the directory tree. Since the hash-value of the user-root of Susan also changes, she has to sign it again.

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

5.4 Calculating the hashes

Table 5.1 shows a few example directory entries of the home-directory of Susan. All values printed in bold are secured by the hash-tree, all other values are not part of the hash-tree.

For files that can only be written by Susan herself (*Secret* and *Calendar*) all the information about the file is included in the directory hash. Any manipulations of the meta data of these files will invalidate the hash.

For files that are writable by everyone (*Share*), the version and the location pointer are not protected by the directory hash. Every user may change the file contents by adjusting the content pointer and increment the file version number.

For files that are writable by the group (*Feedback*), all the information is protected. The group members can change the file contents in the group-root, but they cannot change the content in the user-root.

For redirects (*Documents*), all information about the redirect is protected with the exception of the version number and the forward key, which will be discussed in Section 6.2.

The group-roots are secured in exactly the same way as the user-roots. They are signed with the group signature key S_g instead of the individual user signature key S_u.

The tables in Section 6.6 show a detailed list of all possible file access rights combinations and which protection is used in which case.

5.5 Rollback attacks

A rollback attack is an attack in which current data in the file system is replaced with data that has previously been written to the system, without anyone noticing.

As we have already said, every object in our file system has a version number v attached. Thus, all user-roots and group-roots also have a version v. All clients track the version numbers of all user- and group directories that they access. If a client notices that the version $v(r)$ of a directory has been decremented since the last access, the directory was illegitimately modified by a replay attack. In this case the user can be notified, for example by sending messages to the system log server.

This approach fully prevents rollback attacks for user-root directories. A client will never accept a version version with a decremented value of v, all files within a user-root are protected with a content-hash and both the hash and v are signed by the owner.

In group-root directories, rollback attacks are still possible by other group-members: 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 an adversary takes an old version and 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. I.e. the adversary has successfully performed a rollback attack and removed a change of another node from the version history.

5.6 Fork consistency

Rollback attacks in distributed systems have already been discussed extensively in the literature [53]. The maximum level of consistency that can be achieved in a system like ours is called *Fork Consistency* or also *Fork Linearizability*: "Fork consistency is the strongest notion of integrity possible without on-line trusted parties." [49, p. 124].

Fork consistency means that a malicious server can fork a file system at a specific point of time and not show modifications made by other nodes. A node that only receives information from the malicious server cannot tell that it obtained an outdated instance of the file system. It is impossible to prevent this kind of attack without an online third party. The node could consult this third party and ask for the most recent file system revision. However, the server may never show any other update to the target client and it may also never show any of the target client's modifications to the network. In such a case, the target client and the other clients cannot detect the maliciousness of the server.

Fork consistency can be added to our file system structure just as it was done in [49]: 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.

6 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.

In Chapter 5, we already covered that the locations at which an object is stored depends on the access rights of the object. In the first section of this Chapter, we describe how access permissions are enforced for files and symbolic links. The second section describes the access protection for directories.

6.1 Enforcing permissions for files

As discussed previously, each file stored in the file system can either be readable by the owner, by the owner and the group, or by everyone who may access the file system. The location of the file in our directory structure depends on its access rights.

We already established that all files that belong to a user u are located somewhere in her user-root-directory r_u. Each user has an encryption key K_u and an signature key S_u (see Table 4.1). Likewise, as we already discussed in Section 4.2, each group has a group-root, just like the user-roots for the user directories. The *.groups* directory is split into two subdirectories, named *public* and *private*.

All entries in the *public* directory are stored unencrypted. The pointer to the *private* directory is encrypted with the symmetric group key K_g (see Table 4.1). K_g is encrypted using the subset difference scheme presented in Section 3.2.4 and stored in the *.keys* directory. All current group members can decrypt it. The same holds true for the group signature key S_g.

The decision graph in Figure 6.1 shows where a file that belongs to the user Susan and to the group staff is located, depending on its access rights. It also shows if the pointers are stored encrypted or in plain text.

For files that are only readable by the owner, only a single pointer encrypted with K_u is placed in the user-root. The owner can decrypt the pointer using K_u and read the file contents. An example for this is the file *diary.txt* in Figure 6.2.

For files, that are readable by the owner, a second pointer is placed in the private part of the group-root. The owner can still decrypt the pointer using K_u via her user-root. Group members can access the private part of the group-root using K_g and thus also access the file pointer. An example for this is the file *meeting.txt* in Figure 6.2.

In case this file is also writable by group-members, group members cannot change the pointers in the user-root. They can only change the pointers in the group-root. All users accessing the file have to search the user-root and the group-root for the most recent file

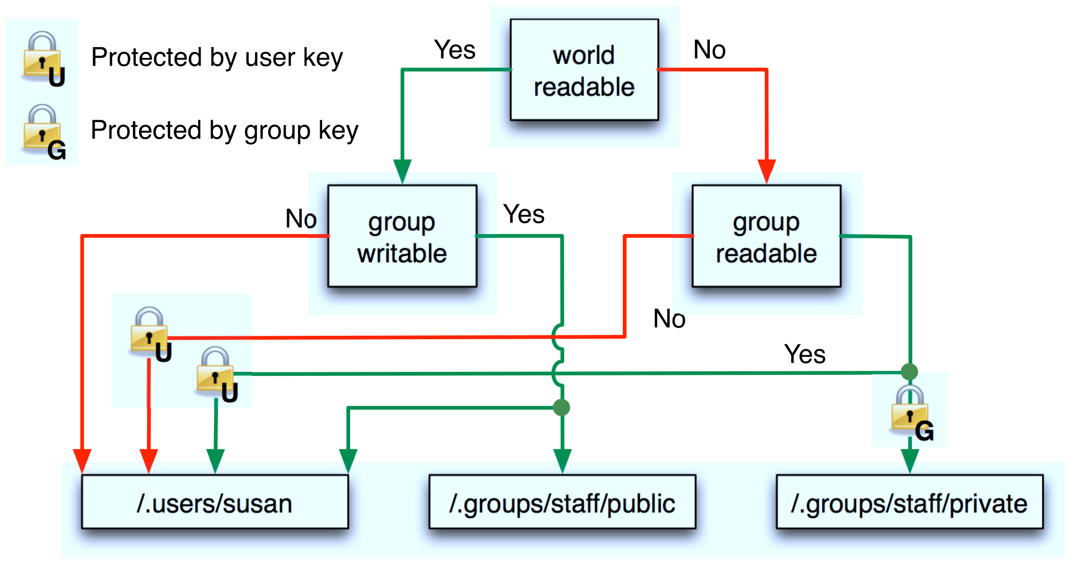

Figure 6.1: Decision graph for file and symlink locations

version. An example for this is *meeting.txt* in Figure 6.3 where the second set of pointers no longer points to the same data as the pointers in the user-root.

For files that are world-readable, the file-pointer in the user-root is stored unencrypted. If the file is also group-writable, a second pointer is stored in the public part of the group root. An example for this is *contact.txt* in Figure 6.3. If the file is world-writable, this second pointer is not necessary because everyone may change the contents directly in the user-root. In this case, the content-pointers are not protected by the hash-tree (see Section 5.4).

6.2 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. The file system structure presented here allows a user in certain conditions to access files he/she should not be able to access.

Figure 6.4 illustrates the problem with an example: There, 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

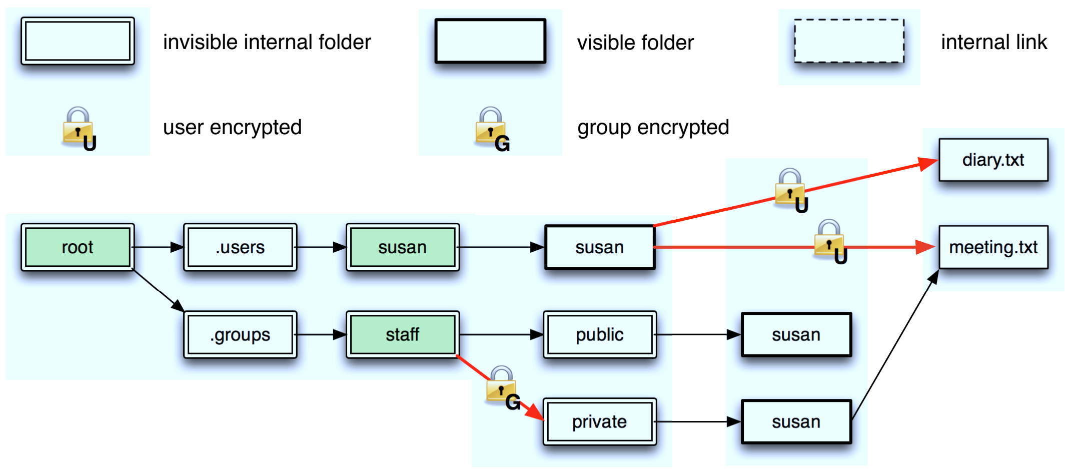

Figure 6.2: Example directory structure

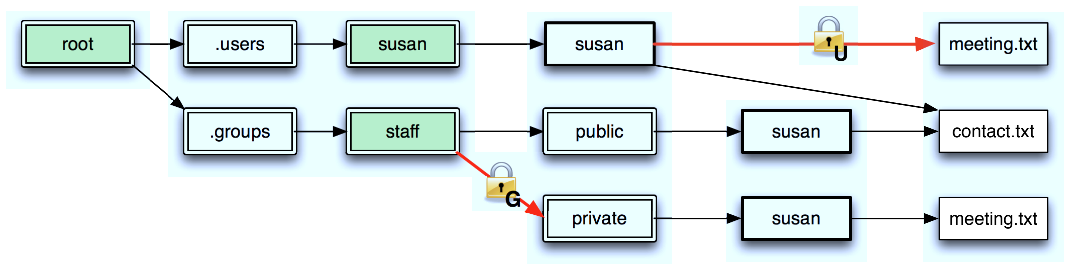

Figure 6.3: Example directory structure

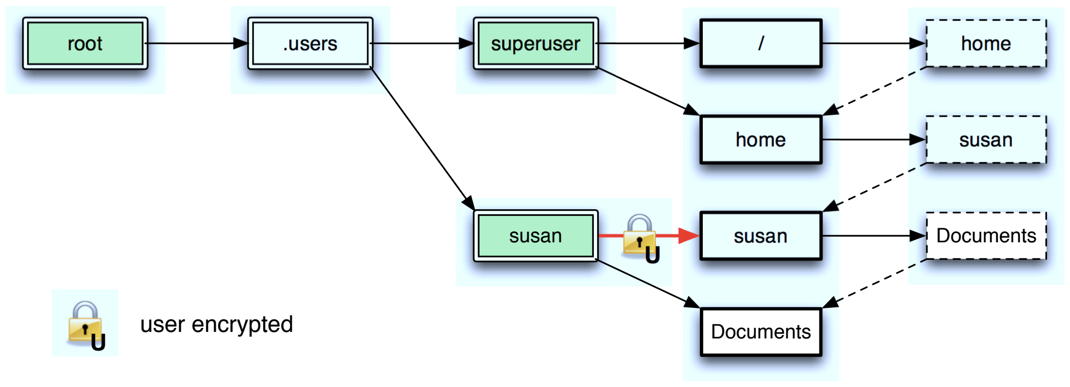

Figure 6.4: Evading access restrictions without forward keys

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.

6.3 Enforcing permissions for directories

Using the forward keys established in the previous section we can enforce permissions for directories.

For a directory, which is only readable by it's owner, the directory forward-key is encrypted with K_u. Hence, only the owner can read the directory contents.

For a directory that is readably by the owner and the group, a second redirect is placed in the private-part of the group-root. The owner can read the directory by following the redirect in her user-root and decrypting the forward-key. Group members can read the directory by following the redirect in the group-root.

For a world-readable directory, the directory forward-key remains unencrypted.

6.4 Files stored in foreign directories

If a group-member that is not the owner of the current directory creates a new file in a group-writable directory, the group-member cannot add the file to the user-root of the directory owner. In this case, the file has to be stored in a different manner: If the file is only readable by the writer herself, she creates a redirect to the actual file location, which resides within the user-root of the writer. The redirect is created in the public part of the group-root. The file pointers in the user-root of the writer are encrypted with K_u. Figure 6.5 depicts an example for this case.

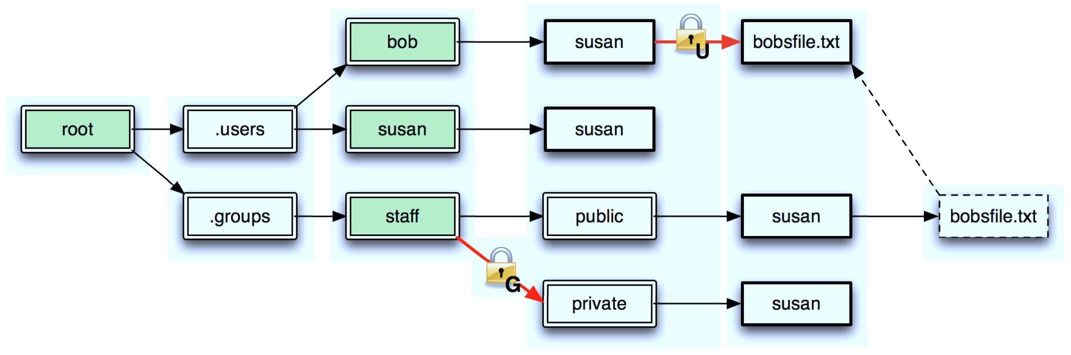

Figure 6.5: Private file in a foreign directory

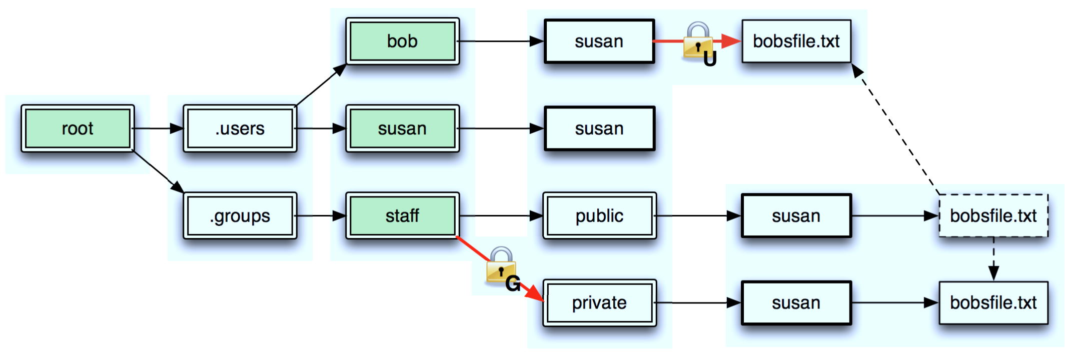

Figure 6.6: Group-readable file in a foreign directory

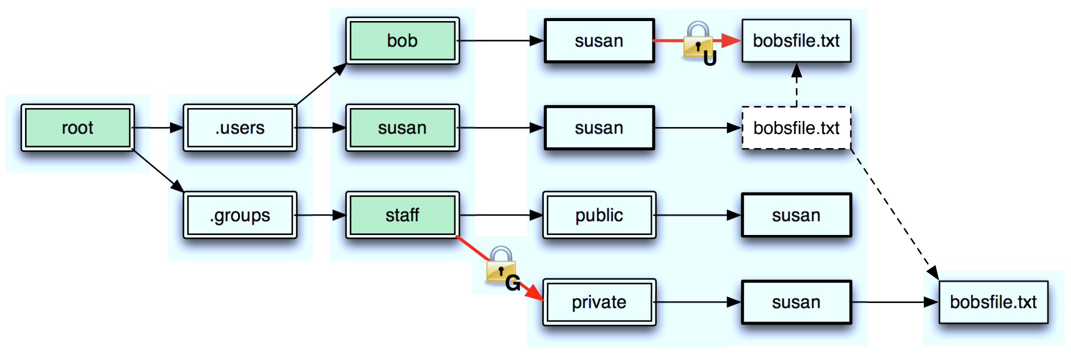

Figure 6.7: Group-readable file in a foreign directory, which is no longer writable

If the file is readable by the writer and the group, the writer stores the file pointers in the private part of the group-root in addition to her user-root. If the file is also group-writable, the pointers are only stored in the group-root and no copy is kept in the user-root. If the file is only group-readable, but not group-writable, a second entry is created in the writer's user-root, which is encrypted with K_u. The entry in the writer's user-root also contains a hash of the entry in the group-directory. The validity and freshness of the group-directory entry can then be verified by checking the entry against this hash. Otherwise, each group-member could mount a rollback attack against the file. In every case, the writer creates a redirect in the public-part of the group-root to let non-group-members see that there is a file in the directory, which they cannot read. Note that if the directory is group-writable, each group-member still can simply delete the file from the directory by removing it from the group-root. This is normal Unix behavior. Figure 6.6 depicts an example for this case.

If the file is readable by everyone and not writable by anyone but the writer, she creates a redirect to the real file location in her user-root in the public part of the group-root. The file pointers in the user-root are not encrypted.

If the file is readable by everyone and writable by the group members, the user stores the file pointers in the public-part of the group-root. All group members can change the content pointers without invalidating the hash.

If the file is readable by everyone and writable by everyone, the user stores the file-pointers in the public part of the group-root. The file pointers are not included in the directory hash and everyone may change them.

When the directory owner (or the file system superuser) later decide to remove the group-writable flag from a directory, files may no longer be added or removed from the directory. When the group-writable flag is removed, the file system creates a redirect for every object that does not belong to the directory owner in the directory owner's user-root. This redirect assures that these files can no longer be removed from the directory without invalidating the directory tree.

6.5 World-writable directories

In world-writable directories, everyone may create files. Existing files may be deleted or moved to another directory even if the user does not own them and may not access their contents.

For directories, the situation is different. Like for files, everyone may create and move directories. In classical Unix systems, everyone may remove empty directories, even if the writer has no access to the directory contents. Non-empty directories may not be deleted. However, a user still may move the directory to a place in the filesystem which only he/she can access. In practice, this is very much equal to the directory being deleted because the directory entry is gone and cannot be found by other users.

In our file system, directory entries are only redirects to the real directory locations. For world-writable directories, we simply do not protect these redirects. Subdirectories of a world-writable directory can thus be "deleted" by removing the redirect. In our opinion, this is no different than the situation in current Unix systems where everyone may move data to a hidden, inaccessible location.

6.6 Pointer locations

In the preceding sections, we have explained the way in which our scheme enforces file permissions. Tables 6.1 and 6.2 show all possible combinations of user access rights and where which pointers are stored. Table 6.1 shows where which type of pointer is created in all the different possible permission combinations when a directory is not world-writable. Table 6.2 shows the same for a world-writable directory.

As discussed in the previous sections, the locations where pointers are created depend on the following properties and conditions:

- the object owner and the object group.
- the object permissions
- the object type (file or directory)
- the owner and the group of the parent directory

In the tables we use the following abbreviations:

- PD means: parent directory
- GR means: group-root
- R means: file is readable
- W means: file is writable
- P means: file pointer
- E(P) means: encrypted file pointer
- +H means: pointers of entry are protected by hash-tree

Because of the complexity of the file permissions we need to provide, a few additional notes (see corresponding Table 6.1):

1. If the directory containing the file/redirect has another group, these pointers are stored in the group-root of the file/redirect group.
2. The pointers are stored in the user-root of the file owner, not in the user-root of the directory owner. They are looked up via the redirect in the public part of the group-root, which is visible to everyone accessing the directory.
3. To prevent a rollback attack launched by a group-member, the encrypted pointers in the user-root also contain the hash of the pointers saved in the private part of the group-root. The freshness of the pointers can be validated by checking the user-root entry.

Type	PD has	PD	Access by		Pointers in		
	same owner	writeable	Group	All	User-root	Public GR	Private GR
File owned by parent directory owner							
File	Yes	Yes	-	-	E(P)+H	-	-
File	Yes	Yes	R	-	E(P)+H	-	P+H[1]
File	Yes	Yes	W	-	E(P)+H	-	P+H[1]
File	Yes	Yes	R	R	P+H	-	-
File	Yes	Yes	W	R	P+H	P+H[1]	-
File	Yes	Yes	W	W	P	-	-
Directory owned by parent directory owner							
Dir	Yes	Yes	-	-	E(R)+H	-	-
Dir	Yes	Yes	R|W	-	E(R)+H	-	R+H[1]
Dir	Yes	Yes	R|W	R|W	R+H	-	-
File not owned by parent directory owner, write via group							
File	No	Yes	-	-	E(P)+H[2]	R+H[4]	-
File	No	Yes	R	-	E(P)+H[2]	R+H[4]	P+H[1,3]
File	No	Yes	W	-	-	R+H[4]	P+H[5]
File	No	Yes	R	R	P+H[3]	P+H[5]	-
File	No	Yes	W	R	-	P+H[5]	-
File	No	Yes	W	W	-	P	-
Directory not owned by parent directory owner, write via group							
Dir	No	Yes	-	-	E(R)+H[2]	R+H[4]	-
Dir	No	Yes	R|W	-	E(R)+H[2]	R+H[4]	R+H[3]
Dir	No	Yes	R|W	R|W	R+H[2]	R+H[4]	-
File not owned by parent directory owner. Parent-directory is not writable[6]							
File	No	No	-	-	E(P)+H[7]	-	-
File	No	No	R	-	E(P)+H[7]	-	P+H[1]
File	No	No	W	-	R[8]	-	P+H[1]
File	No	No	R	R	P+H[7]	-	-
File	No	No	W	R	R[8]	P+H[1]	-
File	No	No	W	W	P[7]	-	-
Directory not owned by parent directory owner. Parent-directory is not writable[6]							
Dir	No	No	-	-	E(R)+H[7]	-	-
Dir	No	No	R|W	-	E(R)+H[7]	-	R+H[1,3]
Dir	No	No	R|W	R|W	R+H[7]	-	-

Table 6.1: Pointer locations for non world-writable directories. See Section 6.6 for an explanation of the abbreviations

Type	Access by		Pointers in			
	Group	All	Directory	User-root	Public group-root	Private group-root
F	-	-	R	E(P)+H	-	-
F	R|W	-	R	E(P)+H	-	E(P)+H
F	R	R	R	P+H	-	-
F	W	R	R	P+H	P+H	-
F	W	W	R	P	-	-
Dir	-	-	R	E(R)+H	-	-
Dir	R|W	-	R	E(R)+H	-	E(R)+H
Dir	R	R	R	-	-	-
Dir	W	R	R	-	-	-
Dir	W	W	R	-	-	-

Table 6.2: Pointer locations for world-writable directories. See Section 6.6 for an explanation of the abbreviations

4. If the directory containing the file/redirect has another group, these pointers are stored in the group-root of the parent directory owner.

5. If the group of the file differs from the group of the parent directory, the group-root of the parent directory group still contains a redirect pointing to the entry in the entry in the group-root of the file group.

6. This situation can occur if a directory was group-writable or writable by everyone and later the permissions were changed by the directory owner or the file system superuser. When the mode is changed the file system automatically creates the redirects in the user-root of the directory owner.

7. The encrypted pointers are stored in the user-root of the file owner. In addition, the user-root of the directory owner contains a redirect to the user-root of the file-owner. This redirect is created by the directory owner when she changes the mode of the directory to no longer allow arbitrary file creation. The redirect can also be created by the superuser.

8. In this case, the redirect is only located in the user-root of the directory owner. The actual pointers are only stored in the private group root because each group member may change the file as she wishes, anyway. File deletion is prevented by the redirect entry.

In Table 6.2, the real directory always contains a redirect only. For files, this redirect always points to either only the user-root or the user- and group-roots where the (perhaps encrypted) copy of the file can be found. For directories, this redirect either points to the user-root or the user- and group-roots where the (encrypted) redirect to the actual directory can be found. If the access is not restricted, the redirect points to the destination directory.

7 Implementation

Due to the complexity of implementing the said approach directly in the IgorFs source tree, we have implemented the first proof of concept version of our file system as a Perl application. This approach has the advantage of allowing us to quickly try out changes in the functionality. Additionally, we do not have to deal with things like thread safety because unlike IgorFs, the Perl application is single-threaded. We also ignore concurrency issues like the concurrent access of different clients, etc. which do not have any association with the cryptographic problems but only make the implementation more difficult.

Figure 7.1 shows how our proposed cryptographic extensions can be added to a system like IgorFs.

The different modules of IgorFs were already briefly discussed in Section 2.3. In IgorFs, the file system calls which have been received from FUSE or NFSv3 are sent to the internal messaging system. Without our new additions, these calls are directly forwarded to the File and Folder Module. The File and Folder Module either answers the requests directly or sends them on to the Directory Handler. The Directory Handler is the part of the system that can return the block identifiers of a file system object when it is given a pathname.

When using our new cryptographic scheme, translations have to happen at several layers of the file system. File system calls are not sent directly to the File and Folder Module; instead they are passed through a translation layer that converts the requests from our external directory layout to requests in the internal directory layout. The directory handler is complemented with a cryptographic extension that can decrypt directory entries to which the node has access. It also verifies the validity of the directory structure. The required keys are stored in the key store.

7.1 The key store

The according data structure is called the key store. It is used by the directory handler for its operations. In Section 4.3 we already mentioned that the group keys are directly stored in the file system. Actually, besides the group keys we also store several more of the file system keys directly in the file system.

There are three different types of keys stored in the file system: root-keys, group-keys, and user-keys. Root-keys are only used by the file system superuser. Group-keys are used by group-members and user-keys are used for user-operations. In practice, these keys are stored in three different locations in our file system (compare Figure 5.2): In the internal directory layout root-keys are stored at `/.keys/rootkeys`, group-keys are stored at `/.keys/groupkeys` and user-keys are stored at `/.keys/keys`. These directories are not present in the external directory structure and thus invisible to the user.

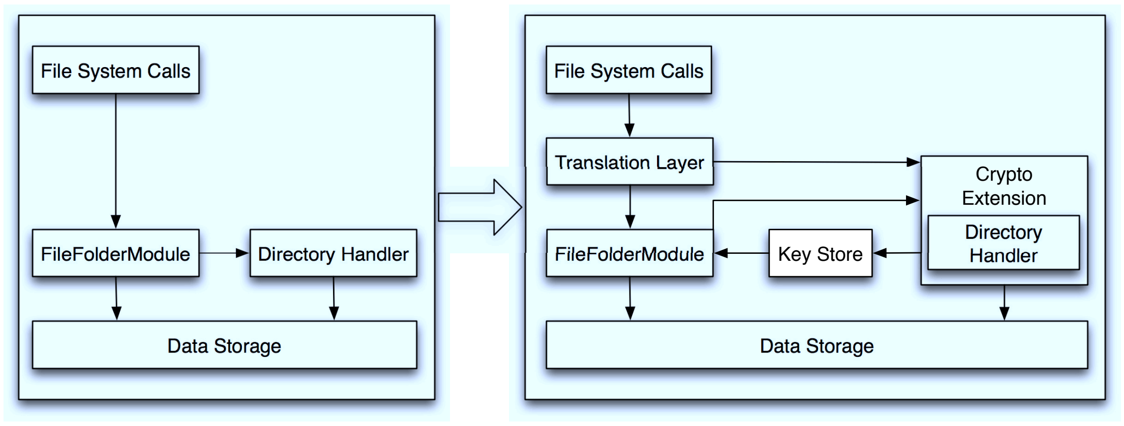

Figure 7.1: IgorFs with and without cryptographic extension

In Section 4.3 we already established, which keys are needed in our file system setup:

Each user possesses three keys K_u, S_u and D_u. These keys need to be given to the user via a secure out-of-band communication method. Additionally, K_u and S_u are stored in the `/.keys/rootkeys` directory encrypted with the private key of the file system superuser. This allows the file system superuser to read, change, and sign all files of the user. It also enables the superuser to give additional copies of the keys to the user if they have been lost accidentally. Since the superuser can regenerate D_u without any additional information, D_u does not have to be stored in the file system [see 4]. The public part of S_u is also stored unencrypted in the `/.keys/keys` folder to allow every file system user to verify user signatures.

Each group is associated with two keys: K_g and S_g. These keys are encrypted with the subset-difference encryption scheme and then stored in the `/.keys/groupkeys` folder. Additionally, the public part of S_g is stored unencrypted in the `/.keys/groupkeys` folder to allow every file system user to verify group signatures.

7.2 User management operations

This section describes how the user-management operations are executed with our added security structure.

There are six important operations that the file system superuser has to do:

1. Add new users to the file system (and generate their keys)
2. Evict users from the file system (and revoke their keys)
3. Add new groups to the file system (and generate their keys)
4. Remove groups from the file system

5. Add users to groups
6. Remove users from groups

7.2.1 Adding and evicting users

The superuser adds a new user to the file system by generating her subset-difference key D_u, a private key K_u, and a signature key pair S_u. All these keys have to be given to the user in a secure manner using an out-of-band channel. The file system superuser then generates the directory `/.users/[uid]` and gives ownership of this folder to the new user.

The superuser evicts users from the file system by removing them from all groups where they are a member. To do this, a new group-key is generated for each group. The key is encrypted in a way that no longer allows the evicted member to decrypt it using the subset difference algorithm. The new group-key is then used to encrypt the pointer to the private part of the group-root. Furthermore a new version of the root folder is created. The (Id, key)-tuple for the root is also encrypted in a way that does no longer allow the evicted user to read it with D_u using the subset difference algorithm. All members of the file system are then made aware of the new file system version.

The directory `/.users/[uid]` may however not be removed from the file system if it still contains files or directories because otherwise the file system could end up in an invalid state. An example for this is a directory containing directories of other users. Removing the user-root would render all subdirectories of all directories belonging to the user inaccessible. The superuser may however delete all files belonging to the user and all empty directories belonging to the user. The ownership of the remaining non-empty directories has to be changed – this moves them to another user-root. After that the user-root may be removed from the file system to free up space and unclutter the file system tree.

7.2.2 Adding and removing groups

The superuser adds a new group to the file system by generating K_g and S_g. These keys are then encrypted using the subset difference scheme and saved in the `/.keys` directory.

The directories `/.groups/[gid]/private` and `/.groups/[gid]/public` are created and given the appropriate permissions.

7.2.3 Adding and removing group members

When a user is added to a group, the file system superuser first regenerates K_g and S_g. The pointer to `/.groups/[gid]/private` is then encrypted with the new K_g. K_g and S_g are encrypted with the subset difference scheme in a way that allows the new user to decrypt them. The new encrypted keys are then stored within the `/.keys` directory.

User removal works exactly the same way, but the new versions of K_g and S_g are encrypted in a way that no longer allows the removed user to decrypt them.

7.3 File system operations

This section describes how the different file system operations are executed with our cryptographic extension. We explain the translation layer of Figure 7.1.

7.3.1 getattr

Arguments: *filename*.
The call returns the file attributes (size, owner, group, etc).

This call is passed through nearly unchanged by the translation layer. The translation layer only looks up the most recent version of *filename* and exchanges the location in the call.

7.3.2 readlink

Arguments: *filename*
The call returns the contents of a symbolic link.

This works just like getattr. The translation layer adjusts the path of *filename* and passes the call through to the file and folder module.

7.3.3 getdir

Arguments: *directory name*
The call returns the contents of a directory.

In our new scheme, each directory is present three times in the directory structure. The translation layer executes a getdir for each of the three directories and merges the contents of all three calls. The merged list is returned.

7.3.4 mknod

Arguments: *filename, numeric modes, numeric device*
The call returns an errno.

This call creates a new file in the file system. Files can be present at one or two locations in the directory structure, depending on the permission bits. The translation layer examines the numeric mode and determines the locations at which the file has to be created. The translation layer then forwards one or two mknod calls to the file and folder module.

7.3.5 mkdir

Arguments: *directory name, numeric modes*
The call returns an errno.

This call creates a new directory in the file system. Directories are created three times in our new directory structure. Hence the translation layer sends three mkdir calls to the file and folder module, one for the user-root, one for the public part of the group-root and one for the private part of the group-root.

7.3.6 unlink

Arguments: *filename*
The call returns an errno.

This call removes *filename*. The translation layer translates this to one or two unlink calls for the file and folder module.

If a directory is group-writable, and the file which shall be deleted belongs to the directory owner, it cannot be actually removed from the directory without invalidating the directory hash. In this case, the file is merely marked as deleted in the group-part of the directory structure. The next time the owner reads the group directory, the system removes the file from the directory hash.

Please note that this may introduce some unwanted backwards-confidentiality problems when new members are added to a group because they can read files that are apparently already deleted.

7.3.7 rmdir

Arguments: *pathname*
The call returns an errno.

This call removes the directory at *pathname*. The translation layer translates this to three rmdir operations, one for the user-root, one for the public part of the group-root, and one for the private-part of the group-root.

7.3.8 symlink

Arguments: *filename, symlink name.*
Returns an errno.

Works just like mknod.

7.3.9 rename

Arguments: *old filename, new filename*
Returns an errno.

This renames an object from *old filename* to *new filename*. Note that both, the old and the new file names are full path names. The rename call can not only change the name of the file, but also move it to an arbitrary new location within the file system.

The target of this call can either be a directory, a file or a symlink. For files and symbolic links, this call converts the rename operations to one or two rename operations depending on the number of redirects that are currently present in the file system.

For directories, this call converts the rename operations to one or two rename operations for the redirect that is pointing to the real directory locations. Note that only the redirect is moved to a new location, the actual hidden directory entries are not moved.

7.3.10 link

Arguments: *filename, hardlink target*
Returns an errno.

IgorFs does not support hard links because of it's directory structure. As a result, we do not support hard links too. Otherwise, the implementation would be the same as for a symlink operation.

7.3.11 chmod

Arguments: *pathname, numeric modes*
Returns an errno.

This call changes the permissions of a given object in the file system. It can only be called by the owner of a file or by the file system superuser.

When this call is executed, the locations of the pointers in the file system have to be changed to match the new permissions according to tables 6.1 and 6.2. Please note that this is a very complex operation. Consider e.g. a directory that has not been world-writable and is now made world-writable. In this case, all files stored directly in the directory have to be moved to the respective owners' directories After that, redirects to the new storage locations have to be created.

7.3.12 chown

Arguments: *pathname, user id, group id*
Returns an errno.

This call changes the user and/or the group of a given object. The owner of a file can call it to change the group of the file to another group to which she belongs. The file system superuser can call it to change the owner and the group of the file arbitrarily.

Together with chmod, this is one of the most complex operations the file system has to handle because to change the ownership of a file or directory, the file or directory has to be moved to a different user- or group-root.

When chown is called, the file or directory is moved to its new location according to tables 6.1 and 6.2.

Special care has to be taken for directories. The contents of the directory still belong to the original owners, because a chown call only changes the ownership of the directory, not of the directory contents. Thus the files and directories in the directory have to be moved back to the user-roots of their respective owners. Redirects have to be created in the directory whose ownership was changed to make them accessible – this is also done according to tables 6.1 and 6.2.

7.3.13 truncate

Arguments: *filename, offset*
Returns an errno.

This call truncates the named file at the given *offset*.

The translation layer looks up all locations at which the file pointers are stored in the hidden directory structure. The truncate call is then executed for all the pointer locations to which the current user has write permissions.

For example if a file is group-writable it is present one time in the user-root of the owner and one time in the group-root of the group (see tables 6.1 and 6.2). If the file is truncated by the owner, the pointers at both locations will be updated. If the file is truncated by a group-member other than the owner, only the pointers in the group-root will be updated.

The most current version can be determined using the version number of the file.

7.3.14 read

Arguments: *filename, size, offset*
Returns data or an errno.

This call attempts to read *size* bytes from *filename* starting at *offset*.

The translation layer looks up all locations at which the file pointers are stored in the hidden directory structure. The read call is then executed for the more recent pointers.

7.3.15 write

Arguments: *filename, buffer, offset*
Returns an errno.

This call tries to (over)write a portion of *filename*.

The translation layer looks up all locations at which the file pointers are stored in the hidden directory structure and executes the call just like for truncate.

8 Communication overhead and cryptographic cost analysis

In this chapter, we briefly describe the communication overhead and cryptographic cost that our proposal adds to a plain distributed file system.

8.1 Theoretical analysis

As mentioned in Chapter 1, 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 enhanced file system causes one or more operations in the underlying P2P network. In this section, we discuss the number of these operations as well as the associated cryptographic cost. We especially discuss the particular case of IgorFs, 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.

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 have 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 (see Figure 8.1 for $d = 2$). 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

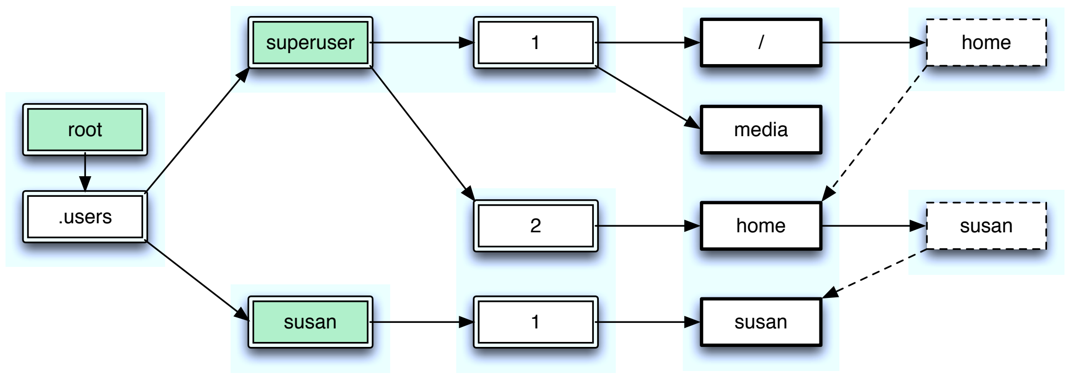

Figure 8.1: Internal directory structure with $d = 2$

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.

Encryption Layer	Redirection Layer	Run	Time [s]
Off	Off	1	1.557
Off	Off	2	1.379
On	Off	1	1.790
On	Off	2	1.431
On	On	1	2.303
On	On	2	1.854

Table 8.1: Encryption layer time measurements

8.2 Practical measurement

In this section, we present a practical measurement of the cost using our proof-of-concept implementation. Please note that the implementation was not optimized for speed in any way - the overhead of the cryptographic implementation probably can be reduced considerably.

For many operations, enabling the cryptographic access permissions has no effect on the speed of the operations. For example for write operations it is very difficult to establish the exact amount of overhead - all signature and encryption operation are not executed while the writes occur. Instead, they are deferred to a later time, i.e. until after the insertion of the data is complete.

Thus, we chose a scenario for our benchmark that is typical for real-world systems and where we can observe a speed difference. When opening a directory and accessing the files in the directory, our scheme has to verify the directory contents by hashing the entries, and it has to decrypt the pointers for the files if they are not world-readable. These operations have to be executed immediately because otherwise the user cannot read his/her files.

To simulate this scenario, we copied a typical Unix directory structure consisting mostly of small files into the file system. The structure consisted of 1980 files in 290 directories with a total size of just over 25 megabytes.

The test of the file system speed consisted of the following steps:

- copy data into the file system
- restart file system to clear all caches
- read data from the file system and measure time
- read data from the file system a second time and measure time

Table 8.1 shows the execution times for both, the first and the second read. The first measurement shows the time the file system takes in a ”cold state”, where all decryption and integrity verification operations have to be performed. The second measurement shows the time the file system takes when the decrypted keys and the directory integrity have already been cached.

When only using encryption and no redirection layer, the numbers for this case should be equivalent to the case when we are using no encryption at all. The overhead in our implementation is due to the higher complexity of the code that has to be executed. As already mentioned, this is an area where our prototype could be optimized.

When using encryption and the redirection layer, the overhead is also due to the higher number of lookups.

Table 8.1 shows our results for a measurement on a system with a 2.3 GHz CPU. As we can see, the use of our new scheme incurs a worst-case overhead of about 50% when reading data for the first time. But with caching, this overhead is already much lower at < 35 %. In our opinion this is not a bad result for a proof-of-concept system and shows the feasibility of our approach.

9 Further extensions

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

9.1 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 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 could 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 report, 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 these features. For files, this is due to the simple necessity that in a distributed setting a node must be able to read an executable file before being able to execute it.

For directories, it is possible to implement the execute-flag. One possible solution is to encrypt the file names stored in the directory with their filename. When trying to read a file, a client has to try to decrypt all directory entries - if it can decrypt an entry it has chosen a valid file name. It would be possible for a client to brute-force all file names in a directory by trying all possible file names - however this is also possible on a non-distributed Unix system.

We did not examine this approach in a greater detail because it was of no importance for our use-case.

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 (setuid), the set-group-Id-on-execution bit (setgid), 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 this technical report 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. In a traditional POSIX 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 disregarded this special case altogether in our approach. We consider

it of very low importance. Furthermore, 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.

9.2 ACL support

The approach presented in this report 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 [39].

The drawback of this way to support ACLs 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 [42] 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 4.3.

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 that we have previously discussed in Section 9.1. For example, 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.

9.3 Identity and attribute based encryption

Identity and attribute based encryption are two new cryptographic schemes which in the future could be used in the ACL scheme we proposed in the previous section instead of the asymmetric key pairs.

Especially attribute based encryption is still very new and there is ongoing work in the cryptographic community to make it usable. To our knowledge at the moment there are no feasibly working implementation of the schemes that would allow them to be used in our distributed file system - there are still problematic points like key revocation.

In future works it should be evaluated if new developments have made these schemes superior to the traditionally used cryptographic primitives..

Identity based encryption (IBE) encryption is a public key encryption system in which the public key can be an arbitrary identifier. The idea for IBE was first proposed by Shamir in 1984 [71], however the first fully functional implementation was not available until 2001 [14].

The original of motivation of Shamir was to devise a cryptosystem where email addresses can be used as the public key.

In an IBE system there is a central instance in possession of a master-key. This master-key is used to derive the publicly available system parameters. The master-key is also used to generate the private keys for all users active in the system.

To encrypt a message, a sender must know the public key of the receiver (i.e. his or her email address) and the publicly available system parameters. The receiver can then use the private key to decrypt this message.

Attribute based encryption (ABE) extends IBE to work with attributes. Each participant has a set of attributes (like e.g. “student”, “math”, and “computer sciences”). Rules are given to an object, e.g. ((“student” and “math”) or (“professor”)). Only participants whose attributes satisfy the object’s rules will be able to decrypt it.

ABE was introduced first by Sahai and Waters [70]. Goyal [33] proposed the first key-policy ABE that allows any monotone access structure. Later, ABE schemes were extended to work with non-monotone access structures [63].

Definition 1 (from [10, 33]) *Let the following be a set of parties: $\{P_1, \ldots, P_n\}$. A collection $\mathbb{A} \subseteq 2^{\{P_1,\ldots,P_n\}}$ is monotone iff $\forall B, C$: if $B \in \mathbb{A}$ and $B \subseteq C$ then $C \in \mathbb{A}$. An access structure (resp., monotone access structure) is a collection (resp., monotone collection) $\mathbb{A}$ of non-empty sub-sets of $\{P_1, \ldots, P_n\}$, i.e., $A \subseteq 2^{\{P_1,\ldots,P_n\}} \backslash \{\emptyset\}$. The sets in $\mathbb{A}$ are called the authorized sets, and the sets not in $\mathbb{A}$ are called the unauthorized sets.*

In principle, ABE and IBE schemes could be used to implement ACLs in distributed systems. To allow some arbitrary user to access a file, we would just have to encrypt the current file keys with the user-Id. Unix groups could be seen as attributes: each user has a set of group-attributes that they can read. The current file keys are then encrypted with the group-Id.

However, there are some serious problems when trying to use this approach with groups: key revocation in IBE and ABE schemes is a difficult subject; proposals for usable schemes have only been made very recently. Boldyreva et al. [13] proposed a revocation scheme with a non-prohibitive cost for IBE in 2008, which also can be adapted to ABEs. This scheme was further extended for ABE usage by Attrapadung et al. in 2009 [8].

These recent findings could make the usage of IBE and ABE viable for distributed file systems. However, to the best of our knowledge, there are no implementations available yet, that support the revocation schemes. But revocation is absolutely essential for usage in file systems; otherwise it would e.g. be impossible to revoke old group members for a group.

IBE and also ABE both rely on asymmetric cryptographic primitives and thus are slower than the respective symmetric operations used in our approach. IBE and ABE also only solve the cryptographic problems associated with user- and group-management. Our technical report also discusses how to guarantee the validity of the data in the file system how to protect the data e.g. against rollback attacks.

10 Conclusion and Outlook

In this report, 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 and 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.

The open-source implementation of our proposal which was used for the performance tests in Section 8.2 is available for download at the web site of our research group [73]. Appendix A shows how the implementation can be installed and used.

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.

A CryptFs usage

This appendix shows how to use our proof-of-concept file system implementation.

A.1 System requirements

The minimal version of Perl required for the system is Perl 5.10.0. The file system also requires a correctly configured Fuse installation.

The following CPAN Perl modules are required in addition to the Perl core modules:

- Moose
- MooseX::Runnable
- MooseX::Singleton
- MooseX::Getopt
- MooseX::Log::Log4perl
- Digest::SHA1
- Perl6::Slurp
- Crypt::CBC
- Crypt::OpenSSL::DSA
- Data::UUId

The module Crypt::Subset::Subscribe also has to be installed; this module was also developed by our working group and is not available on CPAN.

A.2 Options

The file system is started by running the Perl script named runner.pl. When started without options, the script will show a overview over the possible command line options:

```
usage: runner.pl [-?] [long options...]
    -? --usage --help  Prints this usage information.
    --mountpoint       Mountpoint for the file system
    --basedir          Storage directory for file system data chunks
```

```
--debug             Turn on fuse debugging. Default: 0
--uid               User-id of this user inside of the file system.
                    Default: 10001
--gids              Group-ids of this user inside of the file system.
                    Default: 10000 109 10000 10001 10004
--encryption        Use encryption? Default: 0
--redirect          Use redirect layer? Default: 0
--userkey           Path to user key file
--rootkey           Path to file system subset difference root key
```

The `-?` and other help options are self-explanatory, the other options are explained in more detail below.

- mountpoint
  This option sets the mountpoint, where the file system will be attached.
- basedir
  This option sets the storage directory for the file system data chunks. All data and meta-data of the file system will be stored in this directory. Note that this storage directory can also be placed into a distributed storage system, e.g. on a IgorFs volume to extend it with a permission system.
- debug
  This option enables fuse debugging. With fuse debugging enabled, each file system operation that is received from fuse will be output to the console.
- uid
  This option sets the user id of the current user inside of the file system. Please note, that the user-id in the file system does not have to be equal to the current effective user id. For example a non-root-user may be the superuser of a file system she created herself. In this case she will use the uid option to set her file system user id to 0.
- gid
  This option sets the group Ids of the current users. It essentially works the same as the `uid` option.
- encryption
  When this option is set the file system encryption algorithm is activated. Without the option the file system will not use encryption or signatures or integrity hashing. This can be used to measure the overhead introduced by the encryption layer.
- redirect
  When this option is set the file system redirect layer is activated. This option requires the encryption option to be activated. When encryption is turned on without this option, the hidden underlying directory structure will be exposed and

can be analyzed. This also can be used to measure the overhead introduced by the redirect layer.

- userkey
  The path to the current user key file. This key file is needed to access non-world-readable files in file systems with encryption enabled. When a new file system is created this option must not to be set.

- rootkey
  The path to the file system subset difference root key. This key file is required to enable the file system superuser to generate e.g. new groups. When a new file system is created this option must not be set. When the current user id is different from 0 i.e. for users that are not the superuser, this option must not be set.

A.3 Creating a new encrypted file system

This section will show the necessary steps to set up a basic encrypted file system.

The first step is to start the file system with encryption enabled, but without activating the new directory scheme:

```
$ perl runner.pl --basedir /export/store --mountpoint /export/mnt \
--encryption 1 --redirect 0 --uid 0 --gids 0
```

after that we can access the file system:

```
$ cd /export/mnt/
$ ls -l
drwxr-xr-x 1 root root 0 Feb 11 11:16 groupkeys
drwxr-xr-x 1 root root 0 Feb 11 11:16 groups
drwxr-xr-x 1 root root 0 Feb 11 11:16 keys
dr-xr-xr-x 1 root root 0 Feb 11 11:16 proc
drwx------ 1 root root 0 Feb 11 11:16 rootkeys
drwxr-xr-x 1 root root 0 Feb 11 11:16 users
```

We see that we have all the directories that we presented in our paper (`groups, users, keys`) plus a few additional ones that are used for easier management of our file system, but that are not strictly necessary (`groupkeys, rootkeys, proc`). `groupkeys` contains the group cryptographic keys. They are stored separately from the user cryptographic keys in `keys` to make it easier to distinguish between them. `rootkeys` is a directory only accessible by the file system superuser, which contains all keys in an unencrypted form to allow the superuser to access all files and directories. The `proc` directory contains special files which allow us to manage the file system.

We will now add a new user and group to the file system using the `proc` files:

```
$ cd proc
$ ls -l
-rwxrwxrwx 1 root root 30 Feb 11 11:16 addUserToGroup
-rwxrwxrwx 1 root root 30 Feb 11 11:16 createGroupById
-rwxrwxrwx 1 root root 30 Feb 11 11:16 createUserById
```

The special file `addUserToGroup` can be used to add existing users to existing groups. `createGroupById` can be used to create new groups and `createUserById` can be used to create new users.

We can create a new group with gid 1 and a new user with uid 1 and add the user to the group by executing the following commands:

```
$ echo 1 > createUserById
$ echo 1 > createGroupById
$ echo "1 to 1" > addUserToGroup
```

Now the `keys` directory will contain all the respective keys:

```
$ cd ../keys
$ ls -l
-rw------- 1 root root 15985 Feb 11 11:16 0
-rw------- 1 root root 15999 Feb 11 11:17 1
-rw------- 1 root root   991 Feb 11 11:16 group-0
-rw------- 1 root root  1068 Feb 11 11:19 group-1
-rw------- 1 root root   694 Feb 11 11:16 sdtree
```

We now have to copy the respective key files to a location outside of the file system to be able to access it later after we have unmounted it. The file `0` contains the private file system key of the superuser. The file `1` contains the private file system key of the user that we have just generated. The file `sdtree` contains the secret subset difference main key of the file system superuser.

Thus, we copy all the needed files to safety outside of our file system. After that we can stop the file system.

```
$ cp 0 1 sdtree ../..
$ cd ../..
$ fusermount -u mnt
```

Now we can start the file system and use the directory layer. To start it for the superuser we use the user Id 0 and

```
$ perl runner.pl --basedir /export/store --mountpoint /export/mnt \
--encryption 1 --redirect 0 --uid 0 --gids 0 --rootkey /export/sdtree \
--userkey /export/ba/0
```

to start it for the user with the user Id 1 we use

```
$ perl runner.pl --basedir /export/store --mountpoint /export/mnt \
--encryption 1 --redirect 0 --uid 1 --gids 1 ---userkey /export/1
```

List of Figures

List of Tables

Bibliography

[1] AACS LA. *Advanded Access Content System (AACS): Introduction and Common Cryptographic Elements*. Revision 0.91, February 2006.

[2] Accredited Standards Commitee X9. Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA). American National Standard X9.62:2005, ANSI, November 2005.

[3] 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 Operating Systems Review*, Volume 36 (SI): pp. 1–14, 2002.

[4] B. Amann. Secure Asynchronous Change Notifications for a Distributed File System. Diplomarbeit, Chair for Network Architectures, Technische Universität München, 2007.

[5] B. Amann; B. Elser; Y. Houri and T. Fuhrmann. IgorFs: A Distributed P2P File System. In *Proc. 8th IEEE International Conference on Peer-to-Peer Computing*, P2P'08, pp. 77 – 78. IEEE Computer Society, 2008.

[6] B. Amann and T. Fuhrmann. Cryptographically Enforced Permissions for Fully Decentralized File Systems. In *Proc. 10th IEEE International Conference on Peer-to-Peer Computing*, P2P'10, pp. 1 –10. 2010.

[7] 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.

[8] N. Attrapadung and H. Imai. Attribute-Based Encryption Supporting Direct/Indirect Revocation Modes. In *Cryptography and Coding '09: Proceedings of the 12th IMA International Conference on Cryptography and Coding*, pp. 278–300. Springer-Verlag, Berlin, Heidelberg, 2009.

[9] G. Badishi; G. Caronni; I. Keidar; R. Rom and G. Scott. Deleting files in the Celeste peer-to-peer storage system. *Journal of Parallel and Distributed Computing*, Volume 69 (7): pp. 613–622, 2009.

[10] A. Beimel. Secure Schemes for Secret Sharing and Key Distribution. Ph.D. thesis, Israel Institute of Technology, Technion, Haifa, Israel, 1996.

[11] R. Bhagwan; K. Tati; Y.-C. Cheng; S. Savage and G. M. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. 1st Symposium on Networked Systems Design and Implementation*, NSDI'04, pp. 25–25. USENIX Association, Berkeley, CA, USA, 2004.

[12] D. Bindel; Y. Chen; P. Eaton; D. Geels; R. Gummadi; S. Rhea; H. Weatherspoon; W. Weimer; C. Wells; B. Zhao and J. Kubiatowicz. OceanStore: An Extremely Wide-Area Storage System. Technical report, Berkeley, CA, USA, 2002.

[13] A. Boldyreva; V. Goyal and V. Kumar. Identity-based encryption with efficient revocation. In *Proc. 15th ACM Conference on Computer and Communications Security*, CCS '08', pp. 417–426. ACM, New York, NY, USA, 2008.

[14] D. Boneh and M. K. Franklin. Identity-Based Encryption from the Weil Pairing. In *Proc. 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pp. 213–229. Springer-Verlag, London, UK, 2001.

[15] 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*, Volume 3648/2005 of *Lecture Notes in Computer Science*, pp. 1173–1182. Springer Berlin / Heidelberg, 2005.

[16] B. Callaghan; B. Pawlowski and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, Internet Engineering Task Force, June 1995.

[17] G. Caronni; R. J. Rom and G. C. Scott. Celeste: An Automatic Storage System. White Paper, 2005.

[18] I. Clarke; O. Sandberg; B. Wiley and T. w. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proceedings of the International Workshop on Design Issues in Anonymity and Unobservability*, Volume 2009 of *LNCS*, pp. 46–64. Springer Berlin / Heidelberg, 2001.

[19] F. Dabek; E. Brunskill; M. F. Kaashoek; D. Karger; R. Morris; I. Stoica and H. Balakrishnan. Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service. In *Proc. 8th IEEE Workshop on Hot Topics in Operating Systems*, HotOS VIII, pp. 71–76. 2001.

[20] F. Dabek; M. F. Kaashoek; D. Karger; R. Morris and I. Stoica. Wide-Area Cooperative Storage with CFS. In *Proc. 18th ACM Symposium on Operating System Principles*, SOSP '01. Lake Louise, Banff, Canada, October 2001.

[21] 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 Systems*, IPTPS '03. Berkeley, CA, USA, 2003.

[22] P. Di; K. Kutzner and T. Fuhrmann. Providing KBR Service for Multiple Applications. In *7th International Workshop on Peer-to-Peer Systems*, IPTPS '08. St. Petersburg, U.S., 2008.

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

[24] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174, Internet Engineering Task Force, September 2001.

[25] C. Eckert. *IT-Sicherheit.* Oldenbourg, München, 4th edition, 2006.

[26] T. El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pp. 10–18. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[27] B. Elser; A. Förschler and T. Fuhrmann. Spring for Vivaldi – Orchestrating Hierarchical Network Coordinates. In *Proc. 10th IEEE Int. Conf. Peer-to-Peer Computing (P2P'10).* IEEE Computer Society, 2010.

[28] K. Fu; M. F. Kaashoek and D. Mazières. Fast and Secure Distributed Read-Only File System. In *OSDI'00: Proc. 4th conf. on Symposium on Operating System Design & Implementation.* USENIX Association, Berkeley, CA, USA, 2000.

[29] K. E. Fu. Group Sharing and Random Access in Cryptographic Storage File Systems. Master's thesis, MIT, Cambridge, June 1999.

[30] File System in Userspace. *http://fuse.sourceforge.net.*

[31] G. A. Gibson; D. F. Nagle; K. Amiri; J. Butler; F. W. Chang; H. Gobioff; C. Hardin; E. Riedel; D. Rochberg and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. *SIGOPS Operating Systems Review*, Volume 32 (5): pp. 92–103, 1998.

[32] 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*, pp. 131–145. 2003.

[33] V. Goyal; O. Pandey; A. Sahai and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *Proc. 13th ACM conference on Computer and communications security*, CCS '06, pp. 89–98. ACM, New York, NY, USA, 2006.

[34] D. Grolimund; L. Meisser; S. Schmid and R. Wattenhofer. Cryptree: A Folder Tree Structure for Cryptographic File Systems. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, pp. 189–198. IEEE Computer Society, Washington, DC, USA, 2006.

[35] K. Gummadi; R. Gummadi; S. Gribble; S. Ratnasamy; S. Shenker and I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols*

for Computer Communications, SIGOMM '03, pp. 381–394. ACM, New York, NY, USA, 2003.

[36] D. Halevy and A. Shamir. The LSD Broadcast Encryption Scheme. In *Proc. 22nd Annual International Cryptology Conference on Advances in Cryptology*, CRPYTO '02, pp. 47–60. Springer-Verlag, London, UK, 2002.

[37] R. Housley. Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP). RFC 3686, Internet Engineering Task Force, January 2004.

[38] J. H. Howard; M. L. Kazar; S. G. Menees; D. A. Nichols; M. Satyanarayanan; R. N. Sidebotham and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, Volume 6 (1): pp. 51–81, 1988.

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

[40] 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.

[41] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld (editor), *Information processing*, pp. 471–475. North Holland, Amsterdam, Stockholm, Sweden, August 1974.

[42] M. Kallahalla; E. Riedel; R. Swaminathan; Q. Wang and K. Fu. Plutus: Scalable Secure File Sharing on Untrusted Storage. In *Proc. 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pp. 29–42. USENIX Association, Berkeley, CA, USA, 2003.

[43] Y. P. Kising. Proximity Neighbor Selection and Proximity Route Selection for the Overlay-Network IGOR. Diplomarbeit, Lehrstuhl für Netzwerkarchitekturen, Technische Universität München, 2007.

[44] E. Koç. Access Control in Peer-to-Peer Storage Systems. Master's thesis, ETH Zurich, October 2006.

[45] E. Koç; M. Baur and G. Caronni. PACISSO: P2P Access Control Incorporating Scalability and Self-Organization for Storage Systems. Technical Report SMLI TR-2007-165, Mountain View, CA, USA, 2007.

[46] J. Kubiatowicz; D. Bindel; Y. Chen; S. Czerwinski; P. Eaton; D. Geels; R. Gummadi; S. Rhea; H. Weatherspoon; W. Weimer; C. Wells and B. Zhao. OceanStore: an Architecture for Global-Scale Persistent Storage. *ACM SIGPLAN Notices*, Volume 35 (11): pp. 190–201, 2000.

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

[48] K. Kutzner and T. Fuhrmann. The IGOR File System for Efficient Data Distribution in the GRID. 2006.

[49] J. Li; M. Krohn; D. Mazières and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proc. 6th Symposium on Opearting Systems Design & Implementation*, OSDI'04, pp. 121–136. USENIX Association, Berkeley, CA, USA, 2004.

[50] J. Lotspiech; M. Naor and D. Naor. Subset-Difference based Key Management for Secure Multicast. IRTF Draft, Internet Research Task Force, July 2001.

[51] E. K. Lua; J. Crowcroft; M. Pias; R. Sharma and S. Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys and Tutorials*, Volume 7: pp. 72–93, 2005.

[52] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pp. 53–65. Springer-Verlag, London, UK, 2002.

[53] D. Mazières and D. Shasha. Building Secure File Systems out of Byzantine Storage. In *Proc. 21st annual Symposium on Principles of Distributed Computing*, pp. 108–117. ACM, New York, NY, USA, 2002.

[54] R. C. Merkle. Secrecy, Authentication, and Public Key Systems. Ph.D. thesis, Stanford, CA, USA, 1979.

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

[56] 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*, pp. 34–40. 2002.

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

[58] D. Naor; M. Naor and J. Lotspiech. Revocation and Tracing Schemes for Stateless Receivers. In *Advances in Cryptology - CRYPTO 2001: 21st Annual International Cryptology Conf.*, Volume 2139 of *LNCS*, pp. 41–62. Springer Berlin / Heidelberg, 2001.

[59] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Commmunication Magazine*, Volume 32 (9): pp. 33–38, September 1994.

[60] NIST. Digital Signature Standard (DSS). Federal Information Processing Standards Publication 186–2, U.S. Department of Commerce/National Institute of Standards and Technology, January 2000.

[61] NIST. Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, U.S. Department of Commerce/National Institute of Standards and Technology, November 2001.

[62] NIST. Specifications for the Secure Hash Standard. Federal Information Processing Standards Publication 180–2, U.S. Department of Commerce/National Institute of Standards and Technology, August 2002.

[63] R. Ostrovsky; A. Sahai and B. Waters. Attribute-Based Encryption with Mon-Monotonic Access Structures. In *Proc. 14th ACM conference on Computer and communications security*, CCS '07, pp. 195–203. ACM, New York, NY, USA, 2007.

[64] D. Peric. DRFS: Distributed Reliable File System based on TomP2P. October 2008.

[65] D. Peric; T. Bocek; F. V. Hecht; D. Hausheer and B. Stiller. The Design and Evaluation of a Distributed Reliable File System. In *Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT '09, pp. 348–353. IEEE Computer Society, Washington, DC, USA, 2009.

[66] S. Rhea; P. Eaton; D. Geels; H. Weatherspoon; B. Zhao and J. Kubiatowicz. Pond: The OceanStore Prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pp. 1–14. USENIX Association, Berkeley, CA, USA, 2003.

[67] R. L. Rivest; A. Shamir and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, Volume 21 (2): pp. 120–126, 1978.

[68] R. L. Rivest; A. Shamir and L. M. Adleman. Cryptographic Communications System and Method. United States Patent 4,405,829. December 1977.

[69] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pp. 329–350. Springer-Verlag, London, UK, 2001.

[70] A. Sahai and B. Waters. Fuzzy Identity-Based Encryption. In *Adv. in Cryptology - EUROCRYPT 2005*, Volume 3494 of *LNCS*, pp. 457–473. 2004.

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

[72] S. Shepler; B. Callaghan; D. Robinson; R. Thurlow; C. Beame; M. Eisler and D. Noveck. NFS version 4 Protocol. RFC 3010, Internet Engineering Task Force, December 2000.

[73] Self-Organizing Systems Research Group. *http://www.so.in.tum.de.*

[74] 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*, pp. 149–160. 2001.

[75] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[76] TomP2P. *http://www.csg.uzh.ch/publications/software/TomP2P*.

[77] I. Voras and M. Zagar. Network Distributed File System in User Space. In *28th International Conference on Information Technology Interfaces*, pp. 669–674. Cavtat/Dubrovnik, 2006.

[78] K. Wehrle; S. Götz and S. Rieche. Distributed Hash Tables. In R. Steinmetz and K. Wehrle (editors), *Peer-to-Peer Systems and Applications*, Volume 3485 of *LNCS*, pp. 79–93. Springer Berlin / Heidelberg, 2005.

[79] M. J. Wiener. Performance Comparison of Public-Key Cryptosystems. *CryptoBytes*, Volume 4 (1): pp. 1–5, 1998.

[80] Wuala. *http://wua.la*.

[81] Worldwide Proteine Data Bank. *http://www.wwpdb.org*.