Tresorium: cryptographic file system for dynamic groups over untrusted cloud storage

István Lám, Szilveszter Szebeni, Levente Buttyán
Tresorium Kft, and Laboratory of Cryptography and Systems Security (CrySyS)
Budapest University of Technology and Economics
Budapest, Hungary
{*lam,szebeni*}@*tresorium.hu*, *buttyan@crysys.hu*

***Abstract*—In this paper, we present Tresorium, a cryptographic file system designed for cloud based data storage. In Tresorium, files are encrypted before they are uploaded to the cloud storage providers, therefore, not even the cloud storage providers can access the users' data. Yet, Tresorium allows the sharing files within a group of users by using an underlying group key agreement protocol. A key feature of Tresorium is that it handles changes in group membership and modification of files in an extremely efficient manner, thanks to the usage of so called key-lock-boxes and a lazy re-encryption approach. Finally, Tresorium supports an ACL-like abstraction, so it is easy to use. We describe Tresorium, and analyze its security and performance. We also present some simulation results that clearly show the efficiency of the proposed system.**

***Index Terms*—Untrusted Cloud, Dynamic Groups, Cryptographic File System, Collaborative work**

I. Introduction

The tendency of modern companies and organizations is that they outsource their storage systems into the cloud. Storing data in the cloud has various advantages: cloud based storage can be distributed an redundant, providing better dependability, and it is much cheaper for a company to outsource its data than to build and maintain its own data warehouse. In addition, a cloud storage provider solves the backup and the off-site backup. Finally, the costs of a cloud are easily calculable – which is crucial for every company.

The big question is truswothiness and privacy: can a company trust any storage cloud provider? Legal questions are also important: even if a bank trusted a storage cloud provider, could this bank upload customers' data to the cloud legally? The problem is that a cloud storage provider also has access to the stored data, because usually the authentication and authorization is also done by this provider – think about e.g. Google Storage, a user has to log in by providing a user name and a password in order to access his or her files. After the authentication, user authorization is based on the information in an Access Control List (ACL) – a simple list of users and their permissions. The problem with ACL based protection is that it needs an entity, which enforces the access control policy defined by the ACL by blocking the access to a non-permitted file. This is a simple and fast approach, however, problematic. Firstly, if the cloud storage provider is compromised, an attacker can access every file in contempt of the ACL. Secondly, the administrators of the cloud storage provider can override the ACL settings, so they have access to the users' private files. Although ACL based systems may be implemented in more complex and secure way, the basic idea is still the same. To overcome these problems, authorization should not be done on the storage provider side. This leads us to the idea of cryptographic network file systems.

In cryptographic file systems there is no problem with an outside attacker or the curiosity of the administrators, because every file is encrypted *before* being uploaded to the storage cloud. On the other hand, because of encryption, sharing files becomes problematic.

In this paper we present Tresorium, a cryptographic file system that builds on existing traditional storage clouds. We designed Tresorium in such a way that it is extremely flexible, supports dinamic groups and handles changes very efficiently. This means that it is possible to define traditional permissions on the file system in a simple way, so that it is easy to understand and implement.

Security is achieved by the encryption of files before uploading them to the cloud (this solves the problem of limited trust in the operator). Simplicity is achieved with an ACL-like abstraction towards users which is easy to adopt and use. Tresorium also supports groups and content sharing within a group, which makes collaborative work easier. Collaborative applications can leverage cloud services greatly, but they need to be secured, and that is what Tresorium offers.

For efficiency, Tresorium minimalizes the usage of asymmetric cryptography and file uploads/downloads. For flexibility, we rely on encrypting content with different keys so that a subset of the content can be shared at any time.

We first discuss the different players in the environment in section II. In section III, we present what requirements we set, afterwards we present the exact scheme in section IV, with an analysis in section V, with simulation and implementation results in section VI. The related work is presented in section VII. Finally, we conclude the paper in section VIII.

II. System model

In this section, we introduce the different entities that interact with the file system. We also describe what assumptions we have about these entities and how they are structured. Finally, we present exactly what adversaries there are in the system.

1530-2016/12 $26.00 © 2012 IEEE
DOI 10.1109/ICPPW.2012.44

A. Entities and Terminology

The endpoints of the system are the *users* and their software *clients* who would like to share files with each other. These users form *groups* with those users whom they would like to collaborate. A group consists of a set of users or even other groups. Every group has its content in a *tresor*. The tresor contains all the files of the group in an encrypted form. The tresor is protected by the *group key*; using this key the content of the tresor can be decrypted.

The tresor itself is stored on multiple *storage clouds* hosted by *service providers*. The storage clouds enable the users to upload *cloud objects*. A set of these cloud objects make up the group's tresor.

B. Assumptions

In this section, we state the assumptions we have about the entities. This is the context which our solution was designed for.

First of all, we assume that the users of the system would not always like to collaborate with the same people. This means that the aforementioned groups dynamically change in time. Users can be added or removed from an existing group, new groups may appear while others disappear.

During the lifetime of a group the content shared would also change. This means that the group's tresor dynamically changes. Files and directories can be added, removed or changed.

Currently, in storage clouds the service providers enforce the security of the cloud objects using ACL's, which are vulnerable to software bugs or override by an administrator or hacker. The service providers are assumed to be honest but curious, therefore, the service providers cannot be fully trusted. The storage clouds simply act as a passive storage to which users can upload or download cloud objects.

C. Structure

Every group has its own tresor on one or more cloud storages. Each and every user of the group can access the files in this tresor using the group key.

To provide high level privacy and security, all files are stored in an encrypted form on the cloud and only the clients of the group members can locally decrypt them using the group key.

D. Adversaries

The main objective of the adversaries in the system is to access the sensitive data in a chosen group's tresor. A potential adversary may be a former member of a group. The former member has knowledge of many previous keys of the group and could have access to the communication channels between group members and, in particular, could access the channel through which files are uploaded and downloaded to and from the cloud. Another adversary may be an operator at one of the cloud providers who may be able to read the uploaded contents of the group. The administrator has the ability to perform denial of service attacks like deleting tresors and blocking connections. However, we assume that the operator is honest but curious which means that the operator would not perform denial of service attacks, but would like to access the contents of the tresors. Last but not least, there are outsiders, possibly a competitor, who would like to access the group's files.

III. Requirements and design objectives

The requirements we set out to reach were so that we do not lose any advantage of what the cloud has to offer, but ensure the maximal amount of security. We first list the security requirements.

The first objective is *secrecy*: Any user not part of the group cannot access any content in the group's tresor. To further characterize the secrecy of the group we have to specify what previous users and future users of a group can access.

Any previous user of a group who has knowledge of group keys until his or her removal from the group should not be able to read any new content created in the group's tresor since his/her departure from the group. This is known as *Forward Secrecy*.

Any new user that joins the group only knows the group keys from the moment of entering into the group. The new user should be able to read all contents currently in the group's tresor but not deleted files. This we will refer to as *Weak Backward Secrecy*. Simply expecting Backward Secrecy would mean that the new user would only be able to access new content that is created during his/her participation in the group, but the reason the user was added to the group was to share the current and future states of the group's tresor.

As previously stated, we would like to preserve the advantages the cloud has to offer. Keeping this in mind we would like the system to be *reliable* meaning that any user that is authorized to access the contents of a group's tresor should always be able to decrypt the contents efficiently.

Imagine that a group of users have agreed in some group key and they encrypt every file in the groups tresor with that group key. This solution at first glance seems to satisfy everything that we would expect from such an environment, but very quickly we discover that there are many problems with this construction. The main problem is that if a user is removed from the group then the group key would have to change, this would mean that all the existing contents of the group's tresor would have to be re-encrypted. In order to solve the latter problem efficiently, in this paper we propose an efficient symmetric key derivation method that derives keys from the current group key to encrypt the files in the group's tresor.

IV. Proposed Scheme

In this section, we present our core solution. First, we introduce the different components or building blocks that we use, afterwards we present a high level overview of the solution and finally a detailed and formal static and dynamic overview of the system.

A. Building Blocks

We require the ability for the current users in a group to be able to calculate one common group key. The assumption is that only those users can calculate the common group key at time t who are part of the group at time t. For this we use a distributed group-key agreement protocol such as Invitation-oriented TGDH, which we describe in the companion paper [1]. We assume that the group-key agreement protocol supports the following operations on groups:

- Leave: Remove an existing user/group from a group.
- Join: Add a new user/group to a group.
- Calculate: Let a user/group calculate the common group key.
- Refresh: In case of compromise a user/group can refresh its key.

The encryption of the file system is done using symmetric cryptography. Our aolution uses AES in CFB mode, likewise to the OpenPGP standard [2].

B. High level Overview

In our solution, we propose to manage multiple key-lock-boxes in a tree structure. For simplicity, this tree is identical to the directory structure of the tresor. Every directory has a key-lock-box associated with it, and in this key-lock-box we can find the keys of the files within the directory. The key that opens or decrypts a key-lock-box can be found in the key-lock-box above it in the tree structure. The key-lock-box at the top of the tree structure is called the master key-lock-box which can be opened by the group key.

C. Detailed description - static view

$\mathcal{N}$	Number of users in a group
$\mathcal{K}$	set of keys, where $\mathcal{K} \subseteq \{0,1\}^*$
$\mathcal{T}$	set of encryption algorithms
$\mathcal{K}^t$	set of keys for some encryption algorithm $t \in \mathcal{T}$ ($\mathcal{K}^t \subset \mathcal{K}$)
$\mathcal{F}$	set of files
$\mathcal{M}$	set of users in a group
$\mathcal{B}$	set of key-lock-boxes where $\mathcal{B} \subset \mathcal{F}$, which means that the key-lock-boxes themselves are files
$a \triangleright b$	a is the key of b, or opened ascendant key-lock-box of b
$E_k^t(\cdot)$	encryption algorithm, where $t \in \mathcal{T}, k \in \mathcal{K}^t$
$D_k^t(\cdot)$	decryption algorithm, where $t \in \mathcal{T}, k \in \mathcal{K}^t$
GK	Group Key agreement protocol
gk	Group key generated by a Group Key agreement protocol
k_i^t	key, where $k_i^t \in \mathcal{K}^t$

Table I
TRESORIUM NOTATION

Definition 1 (Group Key agreement protocol): Let GK be a group key agreement protocol that assigns every current member of a group $\mathcal{M}_i$ a common group key gk_i unique to that group:

$$GK : \mathcal{M} \longmapsto gk$$

Definition 2 (Key to a file): $k \in \mathcal{K}^t, t \in \mathcal{T}$ is the key to some file $f \in \mathcal{F}$ if the encrypted file f^* can be decrypted so that $D_k^t(f^*) = f$ Notation:

$$k \triangleright f$$

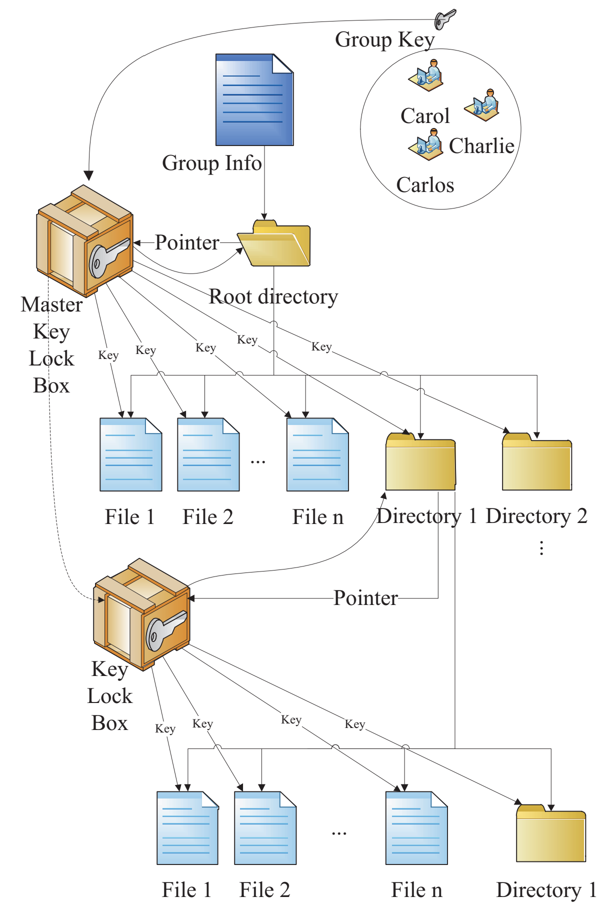

Figure 1. Example Tree

Definition 3 (Key-lock-box): $b \in \mathcal{B} \subset \mathcal{F}$ key-lock-box is an encrypted file in which file–key pairs are stored. This means that b is a discrete function $b(f) : \mathcal{F} \longmapsto \mathcal{K}$ that maps encryption keys to files in the following manner:

$$\forall f \in Dom(b) \quad \exists k \in Ran(b) : b(f) = k \wedge k \triangleright f$$

$Dom(b) \subset \mathcal{F}$ is the set of files mapped in key-lock-box b, $Ran(b) \subset \mathcal{K}$ is the set of keys mapped in key-lock-box b.

As noted in section IV-B, every file's and directory's key is in the key-lock-box that is associated with its parent directory.

Definition 4 (Key-lock-box of a file, parent key lock box): A $b \in \mathcal{B}$ key-lock-box is the key-lock-box of an $f \in \mathcal{F}$ file, if $b(f) \triangleright f$

Definition 5 (Ancestor, descendant of a file): $b \in \mathcal{B}$ is an ancestor of file $f \in \mathcal{F}$ and f is a descendant of b. – with notation $b \triangleright f$ – if: $\exists b_1, \ldots, b_n \in \mathcal{B} : b(b_1) \triangleright b_1 \wedge \ldots \wedge b_i(b_{i+1}) \triangleright b_{i+1} \wedge \ldots \wedge b_n(f) \triangleright f$

Claim 1: From definition 5 it immediately follows that if a user has knowledge of the contents of some key-lock-box $b \in \mathcal{B}$ then the user can decrypt the contents of all files $f \in \mathcal{F}$ where $b \triangleright f$. Intuitively, b "opens" f.

Definition 6 (Master key-lock-box): $mb \in \mathcal{B}$ key-lock-box is master key-lock-box if it is at the top of the key lock box hierarchy and encrypted with a group key gk_i:

$$\nexists b_j \in \mathcal{B} : b_j \triangleright mb_i \quad \wedge \quad \exists gk_l : gk_l \triangleright mb_i$$

D. Detailed overview - dynamic view

In this section, we present the different operations like the changing of a group and the modification of files. We first introduce a few definitions that help determine if an existing key can be still used and afterwards present the algorithms of the operations.

1) Definitions:

Definition 7 (group change): A group M changes when a user is added or removed from the group or a user's key is changed (e.g. in case of compromise). The amount of group changes is counted from 0. The amount of changes in a group is $GC(M) \in \mathbb{N}$

Definition 8 (Key version, key version of file): A key $k \in \mathcal{K}$ has a version of n if only those users of a group know or have the ability to determine the key that were part of the group when $GC(M_i) \geq n$. Notation:

$$KV(k) \in \mathbb{N}$$

A file $f \in \mathcal{F}$ has a key version n, if $k \in \mathcal{K} : k \triangleright f \wedge KV(k) = n$. Notation:

$$KV(f) \in \mathbb{N}$$

Claim 2 (Key version of subtree): The key version of any key-lock-box is at least as big as any descendants' key version.

$$\forall b \in \mathcal{B}, \forall f \in \mathcal{F}, b \triangleright f : KV(f) \leq KV(b)$$

see proof: IX-A

The next definition is the concept of dirty keys, which states that a key cannot be used anymore as a removed user has knowledge of that key. The dirtiness of a key can be decided using Algorithm 1

Definition 9 (Dirty keys): A key $k \in Ran(b), b \in \mathcal{B}$ is dirty – noted: $\tilde{k}$ – , if

$$(\exists b_j \in \mathcal{B} \quad b_j \triangleright b \wedge KV(k) < KV(b_j))$$

Algorithm 1 The determination of the dirtiness of a key: the $is\_dirty(k)$ function

```
1: $mb, b \in \mathcal{B} : mb \triangleright b \vee mb = b$
2: $k \in Ran(b)$
3: **if** $KV(k) < KV(mb)$ **then**
4:     **return** true
5: **else**
6:     **return** false
7: **end if**
```

2) Group Change: Let us assume that group M has changed in some way. Let the changed group be M^*. Due to this, the gk_i group key changes to some gk_i^*. In this case the user that conducted the change – called the sponsor – re-encrypts the master key-lock-box mb with the new group key and in the same transaction records the group change (uploads the file that contains the plublic group information) to keep consistency. These steps can be seen in Algorithm 2.

Algorithm 2 Handling of group change

```
1: $begin\_transaction()$
2: $gk^* \leftarrow calc\_key(user\_private\_key, new\_group\_info)$
3: $write\_file(new\_group\_info)$
4: $mb \leftarrow D_{gk}(read\_file(master\_key\_lock\_box))$ {Read master key-lock-box with old group key}
5: $KV(mb) + +$ {With this step all descending key-lock-boxes become dirty}
6: $write\_file(E_{gk^*}(mb))$ {Write out master key-lock-box with new group key}
7: $commit\_transaction()$
```

3) File Change: Let us imagine that Carol and Carlos remove Charlie from their group. As we stated before, this means that master key-lock-box is re-encrypted with the new group key. However, we would like to avoid having to immediately re-encrypt all contents of the group. This leads to the idea of lazy re-encryption where only those files need to be encrypted with a new key that are changed after Charlie has been removed from the group. At this point, dirtiness becomes useful: only dirty keys have to be replaced with a new key.

After Charlie's removal, keys are only changed when a new file is created or an existing one is modified. In other words, keys are only changed when writing the tresor. This is the idea of lazy re-encryption when only newly created or changed files' key-lock-box hierarchy is re-encrypted.

For example, in Figure 1, Carlos has just been removed from the group, so every key except the new group key is dirty. Carol would like to change $Root\ directory/Directory1/File1$. Before uploading this file, the master key-lock-box is uploaded containing a newly generated key for the key-lock-box. Afterwards the key-lock-box, encrypted with this new key, is uploaded containing a newly generated key for $Root\ directory/Directory1/File1$. Finally, the file can be uploaded with a new key. After these steps Carlos would not be able to read the contents of the uploaded content anymore.

The $get\_key(b \in \mathcal{B}, f \in \mathcal{F})$ key-lock-box function performs the above described procedure (see Algorithm 3). It is important that the calling of $get\_key(\cdot, \cdot)$ and the upload of the new encrypted file has to happen in the same transaction.

4) Cross group share: Up until now we have shown how a variable group of users can share a set of variable files. In practice, the ability to share a whole directory with another group is also desirable. Let us consider the following example, illustrated in Figure 2. Alice and Bob form a group, called „IT

Algorithm 3 Lazy re-encryption in function $get\_key(b, f)$

```
 1: $b \in \mathcal{B}, f \in \mathcal{F} : b \triangleright f$ {were $b$ is opened}
 2: $mb \in \mathcal{B}, mb \triangleright b$ {A master key-lock-box}
 3: if $use\_of\_key = read$ then
 4:   return $b(f)$
 5: else
 6:   if $is\_dirty(b(f))$ then
 7:     $k_f^* \leftarrow generate\_new\_random\_key(t)$
 8:     $add\_key(b, f, KV(mb), k_f^*)$
 9:     if $b \neq mb$ then
10:       $pb \in \mathcal{B} : pb(b) \triangleright b$ {find parent}
11:       $k_b^* \leftarrow get\_key(pb, b)$ {recursive call}
12:     else
13:       $k_b^* \leftarrow calc\_group\_key()$
14:     end if
15:     $write\_file(E_{k_b^*}(b))$
16:     return $k_f^*$
17:   else
18:     return $b(f)$
19:   end if
20: end if
```

department", while Carol, Carlos and Charlie form another group, called „Financial department". Those two departments might have a lot of documents to share, which has to be accessible only for the current employees of „IT department" and „Financial department", regardless who are the actual members of those groups. This is a common situation in a corporate environment, when permissions are defined for a group of users, and a change of group membership implies change of permissions, too.

In order to support this, in Tresorium a new, hidden group is created whose users are the *groups* that the directory is shared with (see Figure 2).

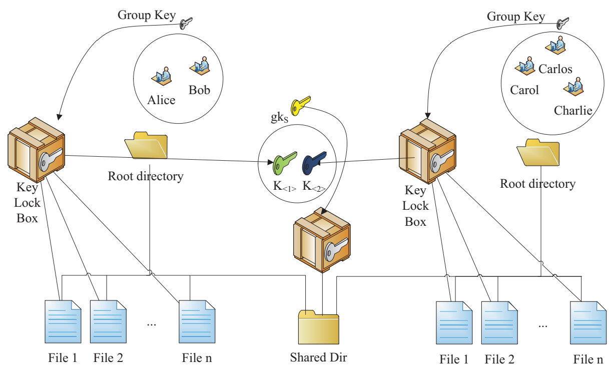

Figure 2. Cross group share example

Let $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ be different groups that may overlap, which form another group, $\hat{\mathcal{M}} = \bigcup_{i=1}^{n} \mathcal{M}_i$. Let the shared files and directories be $\mathcal{S} = \{f_1, f_2, \ldots, f_m\} \subset \mathcal{F}$.

Let $mb_{\mathcal{S}} \in \mathcal{B}$ be the master key-lock-box of files $\mathcal{S}$.

Let $b_1, b_2, \ldots, b_n \in \mathcal{B}$ be key-lock-boxes of groups $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_n$ where every key-lock-box is the *mounting point* of the shared files. In Figure 2, the mount points are the root directories. From these mount points the shared folder can be directly accessed.

Let every group $\mathcal{M}_i$ store a $K_{<i>}$ private key, in an encrypted key-lock-box, that can be used by some chosen key agreement protocol, such that $gk_{\mathcal{S}} = GK(\hat{\mathcal{M}})$ where $\mathcal{M}_i \subset \hat{\mathcal{M}}$ and $gk_{\mathcal{S}} \triangleright mb_{\mathcal{S}} \triangleright \mathcal{S}$. The calculated shared group key $gk_{\mathcal{S}}$ will be the group key of the shared directory. The private keys are stored in the mount points of the groups:

$$K_{<1>} \in Ran(b_1), \ldots, K_{<n>} \in Ran(b_n)$$

Imagine that in Figure 2 Carlos is removed from the group on the right. In this case the master key of the group on the right changes, but due to the fact that Carlos has knowledge of key $K_{<2>}$ it should also be changed to some $K_{<2>}^*$. This can be done by refreshing the key used in the distributed group key agreement protocol of the shared group according to section IV-A.

Similarly, in case a user's key is compromised, the effected user should refresh his or her key in each group that he or she belongs to.

V. Analysis

In Tresorium, the key-lock-boxes are managed in a tree hierarchy, which has multiple advantageous properties, like the ability to share a subdirectory with another group, the small size of a key-lock-box and the ability to use any encryption algorithm. In this section we give some informal arguments on why the scheme fulfills the security requirements set in section III and analyze the performance of the algorithms of section IV.

A. Security

In section III, we stated that the scheme should satisfy Forward and Weak Backward Secrecy. In this section we intuitively show that both these properties hold.

Claim 3: (Forward Secrecy and Weak Backward Secrecy properties in Tresorium key-lock-box trees): Using algorithms 2, 1, 3 the Forward and Weak Backward Secrecy properties hold.

1) Forward Secrecy: If GK is Forward Secure, Algorithm 2 guarantees that only those users can read the contents of mb_i who can calculate the group key gk. This means that a user u will not be able to read any master key-lock-box after the time of his/her departure from the group. Thanks to the recursive call in algorithm 3, $KV(\cdot)$ is maintained correctly. All modified files will have a key version of $KV(mb)$, which means that the Forward Secrecy constraint will hold and no former member will be able to decrypt any of the new files.

2) Weak Backward Secrecy: If a user u has been added to a group M, and if GK is Backward Secure, that user will only know the master key-lock-boxes since his or her entrance to the group. This means that user u will only be able to access those keys that are in the key-lock-box hierarchy during his or her stay in the group, satisfying Weak Backward Secrecy.

Note that it may not be important to support Weak Backward Secrecy. In sucha a case, in algorithm 2 $KV(mb)$ does not have to be increased if a new user is added to the group. This means that in algorithm 3 keys do not need to be changed as frequently, saving time and bandwidth.

B. Performance

In this section, we analyze the complexity of algorithms 2, 1, 3. By complexity we mean the amount of files that have to be uploaded/downloaded. The uploading and downloading of files is by far the most time consuming operation so it has to be minimized.

1) Algorithm 2, group change: Algorithm 2 is a pretty straightforward $\mathcal{O}(1)$ time algorithm since only two files have to be uploaded, namely the group info and the master key-lock-box. The real advantage of this algorithm is that all files are marked dirty in $\mathcal{O}(1)$ time.

2) Algorithm 1, is_dirty: If we can locate the master key-lock-box of a key-lock-box in constant time using a link then the decision about the dirtiness of the key-lock-box key can be done in $\mathcal{O}(1)$ time.

3) Algorithm 3, get_key: Assuming that the user navigates through the file system, the key-lock-box of every file in every step would be refreshed. This would mean that acquiring a read key $get\_key(\cdot, \cdot)$ would have complexity $\mathcal{O}(1)$, since $use\_of\_key = read$.

Claim 4 (the complexity of function $get\_key(\cdot,\cdot)$): Aquiring a key to modify a file (write key) has complexity

$$\mathcal{O}(get\_key(\cdot,\cdot)) \leq \mathcal{O}(|\{b_l \in \mathcal{B} | b_l \triangleright b_i\}|)$$

where b_i is the initial key-lock-box. After a group change and all the files are changed then

$$\mathcal{O}(F get\_key(\cdot,\cdot)) \leq \mathcal{O}(F\alpha(N))$$

where F is the amount of files and N the amount of key-lock-boxes. In the case of obtaining a write key the average complexity of $get\_key(\cdot,\cdot)$ is

$$\mathcal{O}(\alpha(N)))$$

where $\alpha(N)$ is the inverse of the Ackerman $\phi(\cdot)$ function. $\alpha(N)$ can be basically considered constant since $\alpha(m) \leq 4$ if $m \leq 2^{65536}$.

In the worst case scenario (just after a group change) to change a new file $\mathcal{O}(log(N))$ key-lock-boxes have to be uploaded (every ascendant key-lock-box of the file). Later many key-lock-boxes and files will have the same key version as the master key-lock-box meaning that only one or maybe no key-lock-box has to be changed. On average $\mathcal{O}(\alpha(N)))$ re-encryptions are needed.

The proof in section IX-B, can be traced back to the disjoint-set data structures proof [3], where the find method has an amortized time of $\mathcal{O}(\alpha(N))$. This can also be seen in the simulations in Figures 4 and 5.

VI. Simulation and implementation

A. Implementation

We implemented the above described hierarchical key-lock-box structure in C++, with additional novel algorithms, including the group-key agreement protocol which we describe in the companion paper [1], and other novel, however unpublished solutions, like a novel distributed key authentication algorithm. Our implementation was led by the inspiration of flexibility: tresors can be stored on different kinds of storages. Now Google Docs and storages with an SMB (Windows File Server) interface are supported, and this list of storage types is continuously growing.

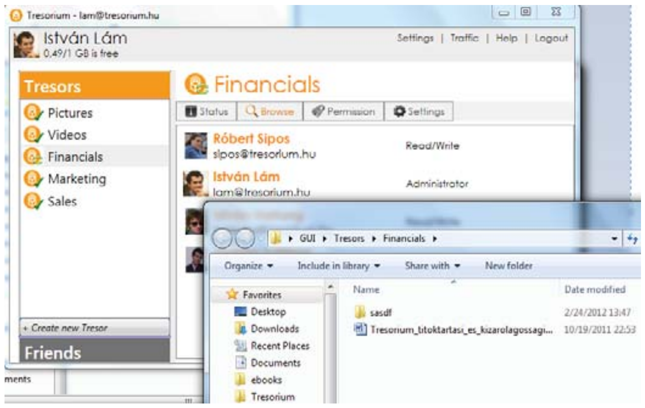

Figure 3. A C# proof-of-concept frontend of the Tresorium C++ Core library

The implementation works as follows: each user can have multiple tresors. Each tresor can be shared with users' friends, as in the example presented in Figure 3. Each tresor is synchronized with a specific local folder, and in those folders files are stored in plain text. Encryption is performed on-the-fly while the user client is uploading or downloading a changed file. A tresor can be shared with any other user, using the invitation feature. Invitations are sent in e-mail, like in Dropbox, but Tresorium invitation contains additional cryptographic information for group-key agreement. The further steps of key-agreement are performed through an application server, which is not needed to be trusted.

According to preliminary results in the same network environment, the Tresorium client uploads almost twice as fast as Google's GDrive and 1.3 times faster as Dropbox.

B. Efficiency measurement

The simulation environment was the following: The file system had a maximal directory depth of d with a Tresorium key-lock-box tree.

The filesystem was generated randomly: Every directory had a random amount $10 \leq r_f \leq 60$ of files and a random amount of $2 \leq r_d \leq 7$ directories. For every $1 \leq d \leq 9$ the result is the average of 50 tests each with 10000 random file change operations.

After building the random file system the simulation chose a file to upload, and if needed conducted the lazy re-encryption.

For every operation, the amount of re-encryptions needed to upload the specific file was noted. This has been presented as a trend line with 33 points, where one point is the average of 300 file uploads.

We investigated two scenarios: in the first in Figure 4, a group change happens before the upload of the files, meaning that at the beginning all keys are dirty (more than 10 million keys when $d = 9$). In the second scenario, in Figure 5, the group changes 5 times, thus the peaks on the curve.

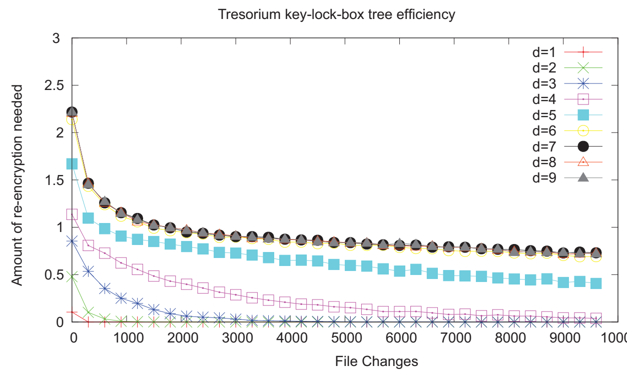

Figure 4. Amount of re-encryptions needed using the Tresorium key-lock-box tree with a one-time group key change

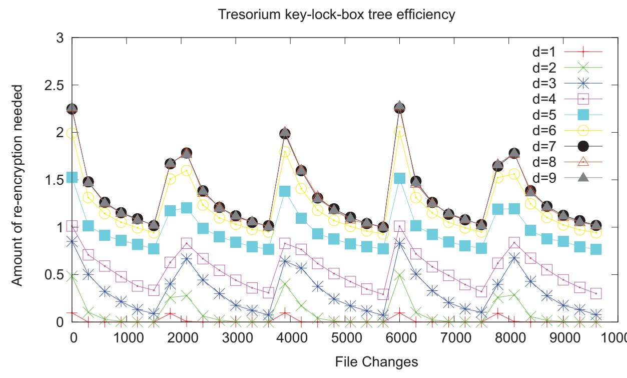

Figure 5. Amount of re-encryptions needed using the Tresorium key-lock-box tree with recurring group key changes

We note that in Figure 4 the average number of re-encryptions needed is 1 even at a directory depth of 9. We also note that after $d = 7$ the number of re-encryptions needed grows very slowly.

VII. Related work

Cryprograhic file systems can be categorized as *local* and *network* cryptographic file systems. Local cryptographic file systems, like CFS [4], CryptFS [5], TCFS [6] or NCryptFS [7] are designed for protecting stored data from attacker with physical access to the client hardware.

Network crytographic file systems usually assume untrusted, network attached storage. The idea of key-lock-boxes was first used in Cepheus [8]. Cepheus encrypts the files before they are uploaded to the network storage, and it supports groups. However the required group database is hard to be maintained as it requires a centralized server. The cryptographic file-system of Harrington [9] is similar to the mentioned one, however does not include any key management and key distribution.

Plutus [10], SiRiUS [11] and SNAD [12] use public key cryptography for key distribution. In Plutus, a novel key-rotation scheme was proposed, which guarantees that only the owner of the file can generate new keys for writing. Compared to Tresorium, Plutus requires much more public key crypography operations. Plutus, SNAD and SAFIUS [13] assume a trusted third party server. For key revocation in SiRiUS, a cryptographic key recovation scheme is used [14], which assumes an active, always on-line owner of the file. Kong et. al [15] proposed an access control for groups which "merge" the keys of the members, and that merged key is used for encryption. This solution is much less efficient than [1], which is used in Tresorium. CRUST [16] uses a symmetric key cryptography based protocol, namely Leighton-Micali protocol [17], which assumes an initial key shared with a trusted third party. On the market, we found that only two solutions provide real cryptographic protection: Wuala [18], which uses [19] and TeamDrive [20] which uses Diffie-Hellman for group key agreement.

VIII. Conclusion

In this paper, we presented Tresorium, a cryptographic file system designed for cloud based data storage. In Tresorium, files are encrypted before they are uploaded to the cloud storage providers, therefore, not even the cloud storage providers can access the users' data. Tresorium allows the sharing of files within a group of users by using an underlying group key agreement protocol. A key feature of Tresorium is that it handles changes in group membership and modification of files in an extremely efficient manner, thanks to the usage of so called key-lock-boxes and a lazy re-encryption approach. Another important feature of Tresorium is the ability to share content between groups, a feature which has not yet been implemented in any of the cryptographic file systems that we analysed.

Besides presenting the operation of Tresorium, we analysed its security and performance. In terms of security, Tresorium achieves Forward Secrecy and Weak Backward Secrecy. Regarding performance, we proved the efficiency of Tresorium by analytical means and simulations.

We are currently implementing Tresorium and plan to commercialize it in the near future.

IX. Appendix

A. Proof of Claim 2

Assume that $\exists b \in \mathcal{B}, \exists f \in \mathcal{F}, b \triangleright f : KV(f) > KV(b)$. Suppose a user u was part of the group M while $GC(M) \leq KV(b)$. From definition 8 it follows that u knows key-lock-box b. From claim 1 it follows that u can decipher file f which contradicts definition 8 since user u was never part of group M when $GC(M) \geq KV(f)$

B. Average Re-encryption Complexity

Asuming that the group has recently changed, all files are dirty except the master key-lock-box. We show that the writing of new files is very similar to the disjoint-set data structure's find function (algorithm 4) which has the average complexity of $\mathcal{O}(\alpha(N)))$.

Algorithm 4 Find(x) - Disjoint-set

```
1: $x$ is a node in tree
2: if $x.parent != x$ then
3:     $x.parent != Find(x.parent)$
4: end if
5: return $x.parent$
```

When finding a node x, all ancestor nodes including x are linked to the root node. Similarly, when writing a file f, all ancestor key-lock-boxes and f become clean, meaning that the key they are encrypted with has the same key version $KV(f)$ as the master key-lock-box.

Just like when a node x becomes linked to the root node and a search can stop after 1 step after reaching x, clean key-lock-boxes and files need not be cleaned, so algorithm 3 stops its recursion once it gets to a clean key-lock-box.

Acknowledgement

We would like to express our gratitude to all those who helped us to implement our ideas for real use. We would like to thank András Eisenberger for his contribution during early stages of our project, and for the team of Tresorium, Balázs Kovács, Tamás Koczka, György Szilágyi, István Hartung, Róbert Sípos and Péter Schaffer.

References

[1] I. Lam, S. Szebeni, and L. Buttyan, "Invitation-oriented tgdh: Key management for dynamic groups in an asynchronous communication model," in *Submitted to 4th International Workshop on Security in Cloud Computing*, 2012.

[2] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer, "OpenPGP Message Format," RFC 4880 (Proposed Standard), Internet Engineering Task Force, November 2007, updated by RFC 5581. [Online]. Available: http://www.ietf.org/rfc/rfc4880.txt

[3] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, pp. 215–225, April 1975. [Online]. Available: http://doi.acm.org/10.1145/321879.321884

[4] M. Blaze, "A cryptographic file system for UNIX," *Proceedings of the 1st ACM conference on Computer and communications security - CCS '93*, pp. 9–16, 1993. [Online]. Available: http://portal.acm.org/citation.cfm?doid=168588.168590

[5] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A Stackable Vnode Level Encryption File System," 1998.

[6] G. Cattaneo, L. Cauogno, A. D. Sorbo, and P. Persiano, "The Design and Implementation of a Transparent Cryptographic File System for UNIX," in *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001, pp. 199–212.

[7] C. P. Wright, M. C. Martino, and E. Zadok, "NCryptfs: A secure and convenient cryptographic file system," in *USEIX Annual Technical Conferences ATC General Track*, 2003, pp. 197–210. [Online]. Available: http://www.usenix.org/event/usenix03/tech/full_papers/wright/wright_html/

[8] K. E. Fu, "Group sharing and random access in cryptographic storage file systems," Master's thesis, MIT, 1999.

[9] A. Harrington and C. Jensen, "Cryptographic access control in a distributed file system," *Proceedings of the eighth ACM symposium on Access control models and technologies - SACMAT '03*, p. 158, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?doid=775412.775432

[10] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu, "Plutus: Scalable secure file sharing on untrusted storage," in *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, vol. 42. USENIX Association, 2003, pp. 29–42. [Online]. Available: http://portal.acm.org/citation.cfm?id=1090698

[11] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh, "SiRiUS : Securing Remote Untrusted Storage," in *Proceedings of NDSS*, no. 0121481, ISOC. Geneva: The Internet Society, 2003, pp. 131–145. [Online]. Available: http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/9.pdf

[12] E. L. Miller, D. D. E. Long, W. E. Freeman, and B. C. Reed, "Strong Security for Network-Attached Storage," p. 1, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=1083323.1083325

[13] V. Sriram, G. Narayan, and K. Gopinath, "SAFIUS - A secure and accountable filesystem over untrusted storage," in *Security in Storage Workshop, 2007. SISW '07. Fourth International IEEE*, 2007, pp. 34–45. [Online]. Available: http://arxiv.org/abs/0803.2365http://dx.doi.org/10.1109/SISW.2007.7

[14] D. Naor, M. Naor, and J. Lotspiech, "Revocation and Tracing Schemes for Stateless Receivers," *Advances in Cryptology CRYPTO 2001*, vol. 2139, no. June, pp. 1–36, 2001. [Online]. Available: http://link.springer.de//link/service/series/0558/papers/2139/21390041.pdf

[15] Y. Kong, J. Seberry, J. R. Getta, and P. Yu, "A Cryptographic Solution for General Access Control," in *Science*, informatio ed., F. Zhou, Jianying and Lopez, Javier and Deng, Robert and Bao, Ed. Springer Berlin / Heidelberg, 2005, no. September 2005, ch. A Cryptogr, pp. 461–473. [Online]. Available: http://dx.doi.org/10.1007/11556992_33

[16] E. Geron and A. Wool, "CRUST: cryptographic remote untrusted storage without public keys," *International Journal of Information Security*, vol. 8, no. 5, pp. 357–377, 2009. [Online]. Available: http://www.springerlink.com/index/10.1007/s10207-009-0081-6

[17] T. Leighton and S. Micali, "Secret-key agreement without public-key cryptography," in *Advances in Cryptology Crypto 93*, D. R. Stinson, Ed. Springer-Verlag, 1993, pp. 456–479.

[18] L. AG, "Wuala - secure online storage," http://www.wuala.com, 2010.

[19] 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*, 2006, pp. 189–198.

[20] T. S. GmbH, "Teamdrive - sync your data fast and secure," http://www.teamdrive.com/.