Cryptree: A Folder Tree Structure for Cryptographic File Systems

Dominik Grolimund, Luzius Meisser, Stefan Schmid, Roger Wattenhofer
{grolimund@inf., meisserl@, schmiste@tik.ee., wattenhofer@tik.ee.}ethz.ch
Computer Engineering and Networks Laboratory (TIK), ETH Zurich, CH-8092 Zurich

Abstract

We present Cryptree, a cryptographic tree structure which facilitates access control in file systems operating on untrusted storage. Cryptree leverages the file system's folder hierarchy to achieve efficient and intuitive, yet simple, access control. The highlights are its ability to recursively grant access to a folder and all its subfolders in constant time, the dynamic inheritance of access rights which inherently prevents scattering of access rights, and the possibility to grant someone access to a file or folder without revealing the identities of other accessors. To reason about and to visualize Cryptree, we introduce the notion of cryptographic links. We describe the Cryptrees we have used to enforce read and write access in our own file system. Finally, we measure the performance of the Cryptree and compare it to other approaches.

1 Introduction

In traditional file systems, access control is enforced by the storage device. Access control lists are consulted in order to decide whether the requests of a user should be followed or not. Since this approach requires trusting the storage device to enforce the access rights properly, it cannot be used in systems that operate on *untrusted storage*, as it is often the case in distributed file systems. Instead, all content is encrypted such that only legitimate accessors can decrypt it. This relieves the storage device from the burden of access control, but requires a clever key management scheme. Although a number of such key management schemes have already been proposed, none of them met the requirements of our own distributed file system, *Kangoo*[1].

The *Cryptree* which we will present in the following has been implemented as part of the development of Kangoo and has a number of favorable properties in comparison to previous approaches. In particular, we wanted the Cryptree to fulfill the following three important criteria:

[1]To be released (http://www.caleido.com/kangoo).

1. *Semantics*: The more users interact in a file system, the more important it is to have useful and intuitive access control semantics. In particular, we require our file system to support *confidentiality* and *dynamic inheritance* of access rights.

2. *Efficiency*: The number of keys to be managed should not grow proportionally to both, the number of files and the number of involved users. Furthermore, the usage of expensive asymmetric cryptography should be minimized.

3. *Simplicity*: The access control scheme should be simple to understand and implement.

The Cryptree achieves these properties by leveraging the file system's folder hierarchy. Our tests indicate that Cryptree manages keys about four times as fast as previously proposed approaches. Especially the performance of operations that recursively affect access rights is dramatically improved.

We first discuss the context of this paper in Section 2. Then, in Section 3, our desired access control semantics are specified in more detail. Section 4 introduces cryptographic links—the building blocks of Cryptree. Section 5 specifies the Cryptree by describing the concrete mechanisms used to manage read and write access keys in our file system. The performance of Cryptree is evaluated in Section 6. Finally, in Section 7, we discuss the cryptographic links and compare the Cryptree to previously proposed structures. The paper is concluded in Section 8.

2 Context and Related Work

This section puts the Cryptree into perspective by outlining previous work in the area of key management on untrusted storage. Of particular interest are the concept of *lazy revocation* which we adopt, and the idea of *key regression* to which we compare cryptographic links in the discussion section (Section 7). Section 7 also presents the advantages of Cryptree in comparison to the key management schemes described in the following. Note that we will skip key management schemes that are based on *trusted* storage (e.g., Kerberos [1]).

2.1 Key Management in File Systems

While early cryptographic file systems [2, 3] did not address key management, it has already found attention in distributed systems like OceanStore [4], PAST [5] or FarSite [6]. More recent systems increasingly employ efficient access control, Cepheus [7] being the one to introduce the idea of lazy revocation (cf Section 2.2). Some of these systems entirely focus on key management and access control [8, 9, 10, 11, 12].

The most widespread and trivial approach to key management is to maintain a key list along with every file. These lists contain an entry for each user who has access rights to the corresponding files. Such an entry consists of a key representing an access right, encrypted with the public key of a user. This enables the user to decrypt the key using her private key and to assert the access rights associated with it. In the rest of this paper, we will refer to this straight-forward approach as the *classic access control list approach*, or short: *CACL approach*.

Plutus [8] improves on this approach by introducing *filegroups*. All files in such a group are encrypted with the same key. Thereby, access can be granted to entire groups of files at once by only revealing one key. In order to avoid unnecessary overhead when access rights are revoked, Plutus applies lazy revocation and introduces a new technique called *key rotation* which was later refined to *key regression* [13], sometimes also referred to more generally as *key-updating scheme* [14].

Interestingly, key management schemes for general hierarchies have already been investigated in depth, e.g. [15, 16, 17, 18], the last one representing the state of the art in this area. In contrast to our work, however, these schemes have not been written in the context of file systems but from a purely cryptographic point of view. Often, they do not support granting structural write access. Their focus usually is on finding clever key generation algorithms that lead to meaningful dependencies between the generated keys, thereby sharing some similarities with the technique of key regression.

2.2 Lazy Revocation

Cepheus [7] has introduced the idea of *lazy revocation*, a technique which reduces the overhead of revoking access rights at the price of slightly lowered security. It is commonly considered a good idea to allow lazy revocation [14, 20, 21]. In the following, this concept—which is also adopted by Cryptree—is explained.

When someone's read access rights for a file or folder (or generally: *item*) are revoked, it is necessary to encrypt the item with a new key in order to prevent that person from accessing the corresponding item ever again. The policy of lazy revocation allows to postpone this encryption until the next change of this item. This affects security insofar as an adversary now can continue reading the file by keeping a copy of the encryption key. Without lazy revocation, he would have been bound to keep the entire file.

Valid encryption keys that might still be known by some former accessor—who is not supposed to know them anymore—are considered to be *dirty*. By *cleaning* dirty keys, we refer to replacing them by newly generated keys and reencrypting the associated contents with them. In the following, *reencrypting* data with a key denotes decrypting the data with the old key and encrypting it again with the specified key.

2.3 Key Regression

Key regression is designed to solve a particular problem that arises when encrypting multiple files with the same key. For instance, Plutus applies it for its filegroups.

Given a filegroup containing n files, encrypted with a dirty key K_1, let us assume we want to modify a file f from this group. Since key K_1 is dirty, it needs to be replaced by a new key K_2. Unfortunately, f is not the only file encrypted with K_1. Hence, when updating the file and reencrypting it with K_2, all other files in the filegroup must be reencrypted with K_2 as well in order to preserve the filegroup benefit of only having one key to access all contained files.

This expensive overhead can be prevented using the key regression technique. Key regression specifies how to generate a key K_2 such that everybody knowing K_2 can compute K_1, but not the other way round. Thus, if key regression is used to generate K_2, the benefit of filegroups can be preserved without reencrypting all files of the group: the knowledge of key K_2 cannot only be used to decrypt f, but also to decrypt all other files by first deriving K_1 from K_2. Therefore, thanks to key regression, the overhead of reencrypting all files of the group can be avoided.

3 Access Control Semantics

This section specifies the desired access control semantics. In most file systems, access control information is stored on a per file basis. When granting read access to a folder and all its contents, there is a flag being set for each of its files, plus recursively for each of its subfolders. Unfortunately, however, this approach entails a number of problems and may make the system behave in ways unexpected to the user. We argue that coupling access control to the folder structure results in much more natural access control semantics. In particular, the file system should adhere to the following rules:

- **Downward Inheritance of Access Rights**: Explicitly granted access to a folder implies inherited access to its descendants. When granting Alice access to a folder *documents/* and later adding a new document to that folder, Alice should also be able to access the newly added one. And when moving items from *documents/* into another folder, Alice should lose access to them. This is different in most other file systems. There, the access rights stick to the items and hence might get scattered all over the file hierarchy.
- **Upward Inheritance of Access Rights**: Explicitly granted access to a folder implies inherited access to the *names* of the folder's ancestor folders. For example, when granting Alice recursive access to a folder *documents/* that resides in */bob/projects/*, Alice should automatically be allowed to see the names of its ancestor folders */bob/* and */bob/projects/*. Without being able to see the parent folders' names, it would not be possible for her to reconstruct the correct path of *documents/*. When *documents/* is moved somewhere else, Alice should still have access to it and gain the ability to see the names of its new ancestors while losing the ability to see the names of its old ancestors.
- **Confidentiality of File and Folder Names**: When Alice grants Bob access to her folder *holiday/cancun_with_bob/*, Bob should not be able to see that there is also a folder named *holiday/greece_with_charles/*. Most other file system handle this issue differently. There, being able to read a folder name typically implies being able to read the names of its siblings as well.
- **Confidentiality of Access Rights**: It should be possible to grant read and write access without revealing who else has access to the files in question.

The Cryptree is designed to efficiently satisfy these semantics.

4 Cryptographic Links

In order to understand how the Cryptree is constructed, it is essential to know the simple, yet powerful, concept of cryptographic links. These links resemble the edges of the graphs proposed in works like [16, 17, 18]. A cryptographic link from a key K_1 to another key K_2 enables everyone in possession of K_1 to derive K_2. Concatenating multiple links allows to build cryptographic data structures. Note that when K_1 becomes dirty (cf Section 2.2), K_2 becomes dirty as well. Two types of cryptographic links are distinguished: symmetric links and asymmetric links.

4.1 Symmetric Link

Given two secret keys K_1 and K_2 and a symmetric encryption algorithm (our implementation uses AES-128), we denote the result of encrypting K_2 with K_1 by $K_1 \rightarrow K_2$, where $\rightarrow$ is called a *symmetric cryptographic link*. Anyone knowing K_1 and this link can derive K_2.

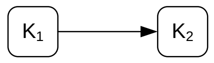

Figure 1. The symmetric link $K_1 \rightarrow K_2$.

4.2 Asymmetric Link

Asymmetric links are basically the same as symmetric links. Their special property is that an asymmetric link $K_1 \rightharpoonup K_2$ can be updated to $K_1 \rightharpoonup K_2'$ without the knowledge of K_1, which is impossible with symmetric links. The price for this property is a loss of performance since we need to make use of asymmetric cryptography (our implementation uses RSA-1024).

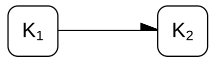

Figure 2. The asymmetric link $K_1 \rightharpoonup K_2$.

Given a key pair $(K_{public}, K_{private})$ as well as a secret key K_2, the asymmetric link is constructed by encrypting K_2 with K_{public}. The public key K_{public} is always stored together with the link while the private key $K_{private}$ is kept secret. Defining $K_1 = K_{private}$, the result is the asymmetric link $K_1 \rightharpoonup K_2$.

In order to update $K_1 \rightharpoonup K_2$ to $K_1 \rightharpoonup K_2'$ for an arbitrary K_2', K_2' is encrypted with K_{public}. The result is $K_1 \rightharpoonup K_2'$.

5 The Cryptree

This is the core section of this paper. First, the Cryptree in general is specified. Then, the two Cryptrees used for access control in our file system are described in detail. It follows a section about how file system operations need to take the Cryptrees into account.

5.1 General Cryptree

A Cryptree is a cryptographic data structure consisting of keys and cryptographic links. It is inspired by [18] and can be seen as a directed graph with keys as vertices and cryptographic links as edges. Given a Cryptree with at least two keys K_1 and K_2, K_2 can be derived from K_1 if and only if there is a directed path from K_1 to K_2. This is a consequence of the nature of cryptographic links. Furthermore, as the name indicates, Cryptrees often have a tree structure.

Due to this tree structure, Cryptrees can be used to efficiently manage the keys of nested folders in cryptographic file systems, typically by making the links of the cryptographic tree publicly available. From this alone, no information about the keys can be derived. However, it suffices to know one single key in order to recursively derive all descendants of that key. This is a powerful property when designing access control in cryptographic file systems, as access to entire subtrees can be granted with $O(1)$ operations. It also leads to intuitive and natural access control semantics.

The designs of two specific Cryptrees to enforce read and write access control are described in turn.

5.2 Read Access Cryptree

In this section, we specify the Cryptree used for read access control in our file system. We first describe the keys used to represent folders and show how they are linked, and then give a short description of the file keys to complete the picture. The *read access Cryptree* efficiently satisfies our desired access control semantics as specified in Section 3, without the need for any additional constructs.

5.2.1 Folders

The central element of our read access Cryptree is introduced next: the folder item with its five secret keys. This folder item can be combined with other folder items in order to construct a folder tree. We start with a complete specification of the cryptographic links and keys used to represent the folder item. Then, the benefits of a tree of such folder items are explained by means of an illustrated example.

Each folder f has five secret keys:

- A *data key* DK_f to encrypt all data needed to represent the folder. This includes its name, creation date and whatever other information the file system stores about the folder.
- A *backlink key* BK_f to find out information about parent folders.
- A *subfolder key* SK_f to read subfolders.
- A *files key* FK_f to read the files this folder contains.
- An optional *clearance key* CK_f that can be revealed to other users in order to grant access to f and its descendants.

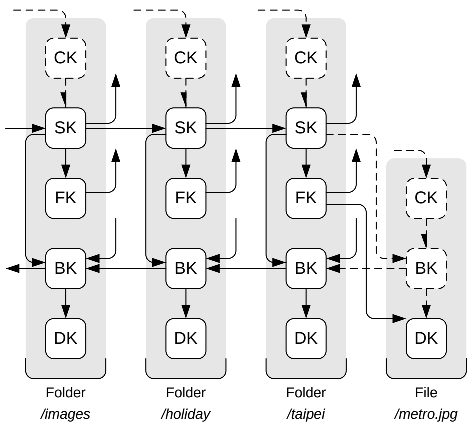

Figure 3. An instance of a read access Cryptree as described in Section 5.2. The dotted parts represent optional keys and links. The links that start or end nowhere indicate arbitrary numbers of connections to elements not shown in the figure, namely links to additional child folders and files. The incoming dotted links pointing to the clearance keys denoted with CK represent read access rights.

Instead of storing the keys themselves, five cryptographic links are stored along with each folder f:

$$BK_f \to DK_f \quad BK_f \to BK_{p(f)} \quad SK_f \to BK_f$$

$$SK_{p(f)} \to SK_f \quad SK_f \to FK_f$$

where $p(f)$ denotes the parent folder of f. Optionally, there might be a sixth, asymmetric link: $CK_f \rightharpoonup SK_f$. Figure 3 illustrates these links in an example with three nested folders.

Due to the way the involved keys are linked, it suffices to know the subfolder key $SK_{holiday}$ of the folder *holiday/* in order to derive all the keys necessary to not only decrypt the folder itself, but also to completely decrypt all its descendants as well as the metadata of its ancestors. Note that in spite of being able to read the names of *holiday/*'s parent *images/*, no information about the other children of *images/* can be decrypted since there is no directed path from $SK_{holiday}$ to SK_{images}.

Unlike in the case of the write access Cryptree discussed in the subsequent section, the asymmetric nature of $CK_f \rightharpoonup SK_f$ is not strictly needed: It could be avoided by linking together the two trees ($WSK_f \to CK_f$). However, it facilitates cleaning the key structure and improves simplicity since the two trees can be kept independent from each other.

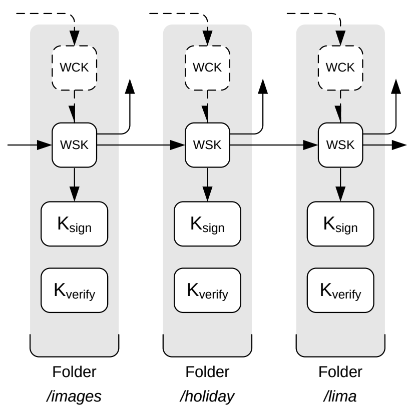

Figure 4. The write access Cryptree as described in Section 5.3.

5.2.2 Files

Every folder may contain one or more files. Each file f is encrypted with its own data key DK_f which can be retrieved using the link $FK_{p(f)} \rightarrow DK_f$ which is stored along with the file. Again, $p(f)$ denotes the parent folder of f. In case we want to be able to grant read access to individual files, we need to add a backlink key BK_f and a clearance key CK_f, and link them as shown in Figure 3.

5.3 Write Access Cryptree

In systems operating on untrusted storage, write access control is usually done by creating a sign/verify key pair for every file and folder. The signature key is used to sign all changes of these items, and the verification key is used to verify the legitimacy of changes. While the verification key is distributed to all readers, the signature key is only revealed to writers of a file or folder. The verification key is also made available to the server such that it can deny unauthorized write operations. As [19] points out, this is not necessary for purely log-structured file systems. In this case, the server does not have to verify write operations.

The verification key can be made publicly available and can be stored in plain text. The signature key however must only be revealed to authorized writers. To do so, we can again leverage the Cryptree structure. Figure 4 illustrates the keys involved in Kangoo's write access Cryptree. These keys are:

- A *verification key* K_{verify} stored in plain text along with the folder.
- A signature key K_{sign} which can be used to sign a write operation.
- A *write subfolder key* WSK which allows gaining write access to subfolders. It has the same role as the subfolder key SK in read access control.
- An optional *write clearance key* WCK which is revealed to grant someone write access. It has the same role as the clearance key CK in read access control.

As before, access can simply be granted by revealing the clearance key. The main difference to read access control is that dirty keys must be cleaned immediately since lazy revocation is not applicable for write access.

Note that the asymmetric nature of $WCK \rightharpoonup WSK$ allows us to guarantee confidentiality of access rights. If it was not asymmetric, writers would need to know WCK when cleaning WSK. Then, WCK would get dirty whenever a writer loses access to it as a result of a move operation. Hence, WCK has to be cleaned again by the writers, which requires knowledge of all incoming links to WCK. Eventually, if the writers do not know each other, an asymmetric link is unavoidable.

5.4 Operations

We now show how the different file system operations are performed. We assume that granting and revoking access is always done by someone with full access rights to all involved items. Currently, we do not support the delegation of administration rights.

When reasoning about the efficiency of operations, we assume the folder hierarchy to resemble a random binary tree and therefore to have a depth of $O(\log n)$, where n denotes the total number of files.

5.4.1 Cleaning

Before executing any operation, it must be ensured that all involved items are clean. Most keys are cleaned lazily, so we will have to check them for dirtiness when doing an operation as described in the following. An exception are subfolder keys and clearance keys: Both are always kept clean, as explained later in this section.

If a file f we would like to change is dirty, we must reencrypt its contents with a new key DK'_f. If the file key $FK_{p(f)}$ of its parent is also dirty, we have to replace $FK_{p(f)}$ as well and update all the links $FK_{p(f)}$ is involved in. If $FK_{p(f)}$ is already clean, it suffices to update the link $FK_{p(f)} \rightarrow DK_f$. In addition, if someone is granted read access to this individual file, there will be a backlink key. In that case, the backlink key is also cleaned and the involved links are updated.

If a folder f we would like to change is dirty, there is slightly more work to do than in case of files. Besides replacing f's data key and reencrypting the metadata as before, we also have to check the backlink structure. If the backlink key is dirty, it has to be replaced. So do all the dirty backlink keys of its descendants. In the worst case, we will end up replacing $O(n)$ backlink keys, where n is the number of files. Fortunately, there is no operation that leaves more than $O(\log n)$ backlink keys dirty (cf Section 5.4.4). Therefore, the average number of backlink keys becoming dirty must be in $O(\log n)$. Since we cannot clean more items than there are dirty ones, the amortized average cost of cleaning is also in $O(\log n)$.

Subfolder keys are always kept clean. The two operations during which they might get dirty (revoke read access and move) clean them immediately. The reason to do so is that we want to ensure that writers with limited write access rights will always be able to clean everything when performing an operation.

To give an example of how this would go wrong if subfolder keys were not cleared immediately, consider the nested folders *images/2047/moon/*. Say Bob has read access to *2047/* and Claire has read and write access to *moon/*. If we revoked Bob's access without cleaning any keys and if subsequently, Claire wanted to add some file to the folder *moon/*, it would be necessary to clean up a number of keys in order to prevent Bob from being able to access the new file. One of the keys to be cleaned would be the subfolder key of *2047/* which Claire must not know and therefore cannot clean. To prevent Claire from getting into this kind of trouble, we keep subfolder keys clean.

Clearance keys are also be kept clean. These keys are only changed by the administrator to grant and revoke access. Since they cannot get dirty during structural changes, writers do not need to know about them.

5.4.2 Granting Read Access

Before granting someone read access to a file or folder f, it must be ensured that f has a clearance key CK_f. If not, such a key is generated and the necessary links described in Section 5.2 are established. Subsequently, the key CK_f is revealed to the new accessor, thereby providing access to f. In our file system, keys are revealed using additional items we do not describe in this paper, but it could generally also be performed with any other suitable key exchange scheme.

5.4.3 Granting Write Access

Granting write access is done by revealing the writer's clearance key WC in the same way as when granting read access.

5.4.4 Revoking Read Access

Revoking access is more complex than granting access. When access to a folder f is revoked, all its descendants must be marked dirty. Revoking read access requires the clearance key CK_f to be cleaned and the new version revealed to the remaining accessors. Furthermore, all subfolder keys of the descendants are cleaned immediately for the reasons mentioned in Section 5.4.1. Together with the subfolder keys, the backlink keys of the descendants are cleaned as well. This is acceptable since these elements are touched anyway and it enables us to guarantee that no operation leaves more than $O(\log n)$ backlink keys dirty.

Revoking read access is—besides revoking write access—one of the most expensive operations in the Cryptree approach.

5.4.5 Revoking Write Access

Revoking write access cannot be done lazily. We need to replace the clearance key WCK and all $O(n)$ involved write keys and write verification key pairs immediately. The asymmetric cryptography involved in generating these key pairs makes revoking write access the most expensive operation. Fortunately, it does not happen too often.

5.4.6 Moving Folders

The move operation is the most complex operation in Cryptree. When moving a subtree to a new parent, the whole subtree must be marked dirty. This is necessary since there might be users losing access to the subtree as a result of the move operation. By moving the subtree, they are no longer allowed to read the old ancestors. This privilege was granted using backlink keys so we need to mark them dirty as well. Therefore, we do not only need to mark the moved subtree itself as dirty, but also the old ancestors of the subtree. As in the revoke read access operation, the subfolder keys and backlink keys of the moved subtree are cleaned immediately.

5.4.7 Other operations

We skip the other operations like deleting, renaming, creating or otherwise changing a file or folder as they are straight forward to implement.

6 Performance Evaluation

This section compares the processing time spent on cryptographic operations in different key management approaches. First, the methodology and setup of our tests are explained, then the results are presented and discussed.

6.1 Methodology

6.1.1 Evaluated Systems

We consider four encryption approaches in file systems:

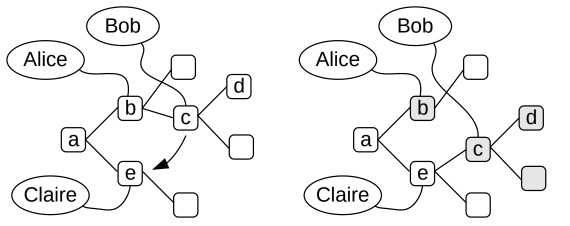

Figure 5. A folder tree before and after a move operation, Alice having read access to folder b, Bob to c and Claire to e. Dirty items are filled gray. Folder c is being moved from its current parent b to e. Hereby, the whole subtree starting with c gets dirty because Alice loses access to it. Folder b becomes dirty because Bob loses access to it. Folder a does not becomes dirty since it still is an ancestor of c. Note that this operation can be performed without the need of being aware of the existence of any of the involved users and without explicitly resetting access rights as it is necessary in other systems.

- CACL: The CACL approach as described in Section 2.1.
- Lazy CACL: The CACL approach enhanced by allowing lazy revocation.
- Cryptree: The Cryptree as presented in this paper.
- Encrypt-on-wire: This term refers to file systems storing their data in plain text on the server and encrypting the data for transmission (e.g. AFS, NFS). In encrypt-on-wire systems, the user is typically authenticated at the beginning of a session and a secure connection is established. Every piece of data transmitted over this connection is automatically encrypted on the sender's side and decrypted on the receiver's side. Although this approach is not feasible when operating on untrusted storage, it is still interesting to compare it to the other approaches.

6.1.2 Simulation

In order to make the different approaches directly comparable, we created toy implementations for each of them in Java. These toy implementations contain all the relevant characteristics regarding cryptography of the respective file systems. We let these systems execute the same set of operations on a given set of files and measure the time spent in cryptographic functions.

We took the execution times of all relevant cryptographic library functions in advance. Then, during the actual test, we accumulated these previously measured values instead of measuring the real time spent in the function. This leads to an increased scalability since we do not need to perform the actual calculations anymore. Furthermore, the results are fairer since undesired external influences (e.g., garbage collection cycles) that do not affect our benchmark are eliminated. Observing that subsequently accessed files often reside in the same folder, we assume that in half of the cases, the parent of an accessed item can be read from a cache and does not need to be decrypted again. Furthermore, for simplicity, we ignore the cost of authentication for the encrypt-on-wire approach: It is only necessary once at the beginning of a session; afterwards, an arbitrary number of operation can be executed without authenticating again. For encrypt-on-wire systems which frequently connect and disconnect, the performance might be significantly worse than measured in our evaluation. Finally, we allow CACL and the Cryptree to postpone the generation of write sign/verify keys until they are needed, assuming that the owner's authentication key pair is used instead as long as nobody but the owner has write access to the corresponding files.

6.1.3 Test Case

We have performed tests with the set of files of one of the authors of this paper. The files are roughly what we would expect by a user of Kangoo. One third of the files are documents, one third images, and one third media files; there are no system files. Concretely, our test case comprises approximately 2,600 folders and 29,700 files with a total size of 76 GB and an average size of 2.5 MB. The average file resides at a depth of 6, and the maximum depth in the tree is 16. Moreover, we have considered one million operations with ten possible types, each type having its own likelihood: access (60%), create (20%), delete (10%), move (4%), modify (2%), copy (2%), recursively granting read access (0.6%), recursively revoking read access (0.2%), recursively granting write (0.06%) access, and recursively revoking write access (0.02%).

These parameters represent the expected usage pattern of the file system Cryptree is designed for. Of course, however, these numbers are somehow arbitrary, and we will provide a more general discussion of the Cryptree's performance later in this section.

The tests were all run on a Windows PC with a 3.6 GHz Pentium IV processor and 2 GB RAM, of which the Java Virtual Machine (HotSpot 1.5) is allowed to allocate 64 MB. For asymmetric cryptography, RSA-1024 is used, while for symmetric cryptography, we are applying AES-128. Both algorithms are provided by Sun's own implementation of the Java Cryptography Extension (JCE).

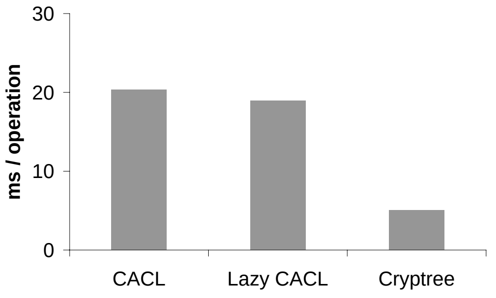

Figure 6. The average processing time spent on key management per operation for CACL, CACL with lazy revocation, and the Cryptree approach.

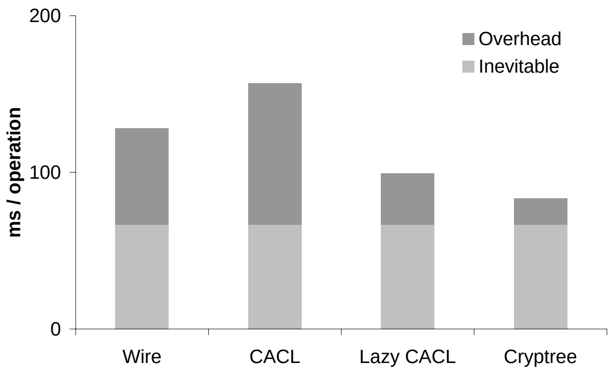

Figure 7. The average processing time spent on cryptography per operation with different approaches.

6.2 Results

In our first simulation, we simply count the number of touched items. With *touching* an item we mean accessing or modifying the content (or metadata) of a file or folder. While performing the one million operations, CACL touched 3.0 million items, lazy CACL touched 2.9 million items, and the Cryptree touched 2.1 million items. While for most operations, the touch count is similar in all approaches, the Cryptree is significantly better for operations that affect entire branches of the folder tree.

In our second simulation, we focus on the cost of key management. Its results are shown in Figure 6. In our test environment, the Cryptree approach could handle about four times as many operations as the CACL approaches. The Cryptree achieves its good performance due to its lower touch count and the use of symmetric cryptography, while the CACL approach needs to perform expensive asymmetric cryptographic operations.

The third simulation shows how the performance gains of the Cryptree compare to the total costs of cryptography (Figure 7). A large share of these costs are inevitable, namely the costs of encrypting and decrypting all data during the communication with the server. Ignoring authentication, the costs of the encrypt-on-wire approach are exactly twice as high as this lower bound because not only the client but also the server encrypts and decrypts all data. The gain of the Cryptree relative to lazy CACL remains the same as before in absolute terms.

Of course, all these results depend on the chosen file and operation set. Generally, the encrypt-on-wire approach is less favorable when frequently accessing large files, and approaches with lazy revocation benefit from frequent revoke operations. Besides these observations, the results do not differ significantly when varying the test cases.

Apart from granting access, all operations are faster with the Cryptree approach than with the CACL approach. Granting read or write access, however, might take longer with Cryptree if the affected folder does not have enough descendants (around 35 for granting read access in our configuration) to amortize the expensive generation of the clearance key that is necessary when granting access to a folder for the first time.

6.3 Findings

As expected, key management with the Cryptree is more efficient than with CACL based approaches. The Cryptree achieves this by leveraging the folder tree structure which allows the use of symmetric cryptography while the CACL approach depends on asymmetric cryptography. Additionally, the number of touched items is significantly reduced with the Cryptree.

Of course, in many cases, the user will not directly feel the performance gains of the Cryptree, since on a fast machine, it is in the order of a couple of milliseconds. However, this is different for operations that affect whole trees of folders, for example moving a folder or granting access to it. For instance, recursively granting read access to a folder with 500 descendants takes 5 seconds with CACL, but only around 0.01 seconds with the Cryptree (or 0.4 seconds if the clearance key has not yet been generated for that folder). This difference is even more dramatic when taking network latency into account. While the costs of this operation remains constant for the Cryptree, CACL needs to touch every single descendant, which can result in an arbitrarily bad performance.

Alternatively to RSA, *elliptic curve cryptography* could also be used. As a consequence, key pair generation (as for example required when creating a clearance key) would be much cheaper. Besides that, the cost of creating signatures would decrease and the cost of verifying signatures slightly increase (when comparing RSA-1024 to ECDSA-163 [22]). With increasing key sizes, RSA loses attractiveness compared to elliptic curve cryptography.

In distributed systems, the actual time spent in cryptographic functions might not be relevant to the user, since it is often dwarfed by the latency of the network. Here, the number of touched items is much more important. Since this number is significantly lower for the Cryptree than for CACL, Cryptree performs much better in distributed file systems than CACL—even if the cryptographic performance gains are of minor importance.

In contrast to the encrypt-on-wire approach, CACL and Cryptree have the advantage of performing most cryptographic operations on the client side. This decreases the processing power requirements of the storage device much.

7 Discussion

We have introduced a novel approach to key management for file systems on untrusted storage. This section first demonstrates the power of cryptographic links in comparison with key regression. Then, the Cryptree is compared to previously proposed structures.

Symmetric Links vs. Key Regression

Despite their simplicity, symmetric links can be very powerful. As an example, we show how such links can make the much more complex technique of key regression obsolete.

The Plutus file system [8] reduces the number of keys a user needs to manage by grouping files and encrypting all files in this group with the same key K_1. A problem arises when K_1 is dirty and a file f changes. In that case, f is bound to get reencrypted with a new key K_2. And with it, all other files in the file group are reencrypted as well. This is a large overhead which can be avoided. Plutus proposes key regression to do so, as described in detail in Section 2.3.

This problem can also be solved using symmetric links. In contrast to the key regression solution, an arbitrary new key K_2 is generated. Then, f is reencrypted with it and the symmetric link $K_2 \rightarrow K_1$ is created. This link is stored along with the metadata of the filegroup. Thanks to this link, it is not necessary to reencrypt the other files either. One key K_2 still suffices to access all files in the group since K_1 can be derived easily if needed using the link.

Let us now compare the two solutions. The advantage of key regression is that no link needs to be stored. This, however, is a minor issue considering that a symmetric link only occupies 16 bytes on disk (with AES-128). The disadvantage of key regression is that it requires more complex algorithms and comes with some limitations. Key regression schemes either make use of expensive asymmetric cryptography, or they allow only a limited number of key updates. A further, maybe more severe limitation, is that key regression, unlike cryptographic links, does not allow the use of an arbitrary K_2. This makes it hard, if not impossible, to move a file to a new filegroup without reencrypting it. As a conclusion, we recommend using simple cryptographic links instead of key regression for most applications.

Cryptree vs. CACL-Approach

The advantages of the Cryptree are now compared to the CACL-approach mentioned in Section 2.

The CACL-approach simply stores a list of access keys along with each file, each entry of this list being encrypted with the public key of a user. The major advantages of this scheme are its simplicity and its compatibility to conventional file systems whose access semantics are based on access control lists. The major drawbacks of CACL in comparison to the Cryptree are its excessive use of asymmetric cryptography, its large number of involved keys and the inherent semantical deficits mentioned in Section 3.

In scenarios with not too many users and in which compatibility with existing access control schemes is important, the CACL approach is definitely the one to choose. However, in scenarios with many users, or when performance and useful semantics are crucial, it seems that Cryptree should be preferred.

Cryptree vs. Elaborate Cryptography

Finally, we compare the Cryptree to the key management scheme presented in [18]. In [18], directed graphs similar to the Cryptree are constructed. The highlights of these graphs in comparison to the Cryptree are that they allow more efficient (by a constant factor) derivation of keys since only hash functions are used (no encryption). Furthermore, shortcut edges are introduced into the graph which allows to derive keys that are n levels further down a hierarchy by only traversing $O(1)$ edges.

However, [18] does not consider the possibility of granting write access. With its current design, structural changes can only be performed by someone who has full access to the whole graph. In contrast, the Cryptree supports granular granting of write access and overcomes the associated challenges by introducing asymmetric links. Furthermore, [18] does not honor lazy revocation and only discusses a minimal set of operations.

In conclusion, we would definitely recommend considering the graph introduced in [18] when looking for a general solution to key management in hierarchies, as it is widely applicable and offers slightly better performance than the Cryptree. The advantage of the Cryptree, however, lies in its simplicity and its focus on access control in file systems.

The Cryptree allows for granular write access, honors lazy revocation, and has proved to be useful in a concrete file system.

8 Conclusions

To the best of our knowledge, the Cryptree is the first cryptographic data structure that combines the latest research in the area of general cryptographic key hierarchies and in the area of access control in file systems. The result is a key management scheme that is more efficient than those proposed in the context of file systems. In contrast to previous solutions coming from a cryptographic context, it does not require any profound knowledge of cryptography and is much easier to apply as it comes with a detailed description about how the Cryptree approach affects file system operations and takes into account vital concepts such as write access or lazy revocation.

The strengths of the Cryptree are its simplicity, its efficiency and the intuitive access control semantics. Its simplicity and the notion of cryptographic links make the Cryptree very flexible and easily adaptable for various scenarios. Its efficiency is unprecedented in the context of file systems and the inherent access control semantics lead to a more intuitive file sharing than in conventional file systems.

In future work, additional techniques are investigated to further improve the efficiency of the Cryptree, in particular the shortcuts or the usage of hash functions for key derivation as discussed in [18]. Furthermore, the proposed Cryptrees could be extended to manage not only the keys of folder hierarchies, but also to manage those of *user hierarchies*.

Acknowledgements

We thank the anonymous reviewers for their useful comments. This research has been supported by the Swiss National Science Foundation.

References

[1] J. Kohl and C. Neuman, *The Kerberos Network Authentication Service (V5)*, RFC 1510, Internet Engineering Task Force, 1993.

[2] M. Blaze, *A Cryptographic File System For Unix*, ACM Conf. on Comp. and Comm. Sec. (CCS), pp. 916, 1993.

[3] G. Cattaneo, L. Catuogno, A. D. Sorbo, P. Persiano, *The Design and Implementation of a Transparent Cryptographic File System for Unix*, USENIX Ann. Tech. Conf., 2001.

[4] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, B. Zhao, *An Architecture for Global-Scale Persistent Storage*, Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2000.

[5] P. Druschel, A. Rowstron, *PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility*, Hot Topics in Operating Systems Conference (HotOS), 2001.

[6] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, R. P. Wattenhofer, *FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment*, OSDI, 2002.

[7] Kevin Fu, *Group Sharing and Random Access in Cryptographic Storage File Systems*, Master's Thesis, Massachusetts Institute of Technology, 1999.

[8] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, K. Fu, *Plutus—Scalable Secure File Sharing on Untrusted Storage*, USENIX FAST, 2003.

[9] E. Goh, H. Shacham, N. Modadugu, D. Boneh, *SiRiUS: Securing Remote Untrusted Storage*, NDSS, 2003.

[10] D. Mazieres, D. Shasha, *Don't Trust Your File Server*, HotOS, 2001.

[11] G. Gibson, R. Van Meter, *Network Attached Storage Architecture*, Comm. ACM, Vol.43, No.11, 2000.

[12] K. Fu, M. F. Kaashoek, and D. Mazieres, *Fast and Secure Distributed Read-only File System*, OSDI, 2000.

[13] K. Fu, S. Kamaram, Y. Kohno, *Key Regression: Enabling Efficient Key Distribution for Secure Distributed Storage*, NDSS, 2006.

[14] M. Backes, Ch. Cachin, and A. Oprea, *Secure Key-Updating for Lazy Revocation*, 11th European Symposium On Research In Computer Security (ESORICS), 2006.

[15] R. S. Sandhu, *Cryptographic Implementation of a Tree Hierarchy for Access Control*, Information Processing Letters Vol. 27, no. 2, pp. 95-98, 1988.

[16] T. Chen, Y. Chung, . Tian. *A Novel Key Management Scheme for Dynamic Access Control in a User Hierarchy*, IEEE COMPSAC, 2004.

[17] H. Chien, J. Jan, *New Hierarchical Assignment Without Public Key Cryptography*, Computers & Security, 2003.

[18] M. Atallah, K. Frikken, M. Blanton, *Dynamic and Efficient Key Management for Access Hierarchies*, ACM CCS, 2005.

[19] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, J. J. Wylie, *Lazy Verification in Fault-tolerant Distributed Storage Systems*, SRDS, 2005.

[20] E. Riedel, M. Kallahalla, and R. Swaminathan, *A Framework for Evaluating Storage System Security*, FAST, 2002.

[21] P. Stanton, W. Yurcik, L. Brumbaugh, *Protecting Multimedia Data in Storage: A Survey of Techniques Emphasizing Encryption*, SPIE, Volume 5682, pp. 18-29, 2004.

[22] J. Lopez and R. Dahab, *Performance of Elliptic Curve Cryptosystems*, Technical Report, IC-00-08, May 2000.