Int. J. Inf. Secur. (2009) 8:357–377
DOI 10.1007/s10207-009-0081-6

REGULAR CONTRIBUTION

CRUST: cryptographic remote untrusted storage without public keys

Erel Geron · Avishai Wool

Published online: 5 June 2009

Abstract This paper presents CRUST, a stackable file system layer designed to provide secure file sharing over remote untrusted storage systems. CRUST is intended to be layered over insecure network file systems without changing the existing systems. In our approach, data at rest is kept encrypted, and data integrity and access control are provided by cryptographic means. Our design completely avoids public-key cryptography operations and uses more efficient symmetric-key alternatives to achieve improved performance. As a generic and self-contained system, CRUST includes its own in-band key distribution mechanism and does not rely on any special capabilities of the server or the clients. We have implemented CRUST as a Linux file system and shown that it performs comparably with typical underlying file systems, while providing significantly stronger security.

Keywords Cryptographic file systems

1 Introduction

1.1 Motivation

Network-based storage systems necessarily reduce the trust that can be placed in the storage infrastructure. For instance, the storage server may be outsourced or shared with other individuals. Moreover, the server may be vulnerable to network-based and physical attacks. Despite this, many existing solutions rely on the remote file server for data integrity and access control. The data in these solutions is often stored unencrypted, and the users rely on the server's access control. This means that users effectively trust the file server's administrators, and data may be exposed from backup copies or stolen hard-disks.

Eliminating the trust in the storage server introduces several security problems. End-to-end security is required, including data secrecy, data integrity, authenticity and access control. Existing solutions that take on these challenges, such as SiRiUS [12] and Plutus [16], rely heavily on the use of public-key cryptography. It is widely known that public-key cryptography algorithms are orders of magnitude slower than their symmetric-key counterparts. This fact encourages the design of a new file system that avoids using public-key cryptography and uses symmetric-key alternatives instead.

1.2 Related work

The popularity of networked storage systems is constantly growing. However, the nature of these systems exposes them to a vast variety of security threats. A framework for evaluating the security of storage systems is presented in [29]. Additional surveys of security services provided by storage systems are given in [17,31]. These works examine and compare existing systems, architectures and techniques.

CFS [5] is the first widely-known file system that performs file encryption. It is a virtual file system (VFS) that encrypts each protected directory with a secret key. CFS was primarily designed for securing local file systems, and thus only relatively simple confidentiality issues were addressed. File sharing in CFS is limited; for instance, sharing a protected file with another user involves disclosing the encryption key

An extended abstract of this paper has appeared in the 4th IEEE Security in Storage workshop, 2007.

E. Geron · A. Wool (✉)
School of Electrical Engineering, Tel Aviv University,
Ramat Aviv, 69978 Tel Aviv, Israel
e-mail: yash@acm.org; yash@eng.tau.ac.il

E. Geron
e-mail: erel.geron@gmail.com

to that user. CryptFS [36] and TCFS [8] are variants of CFS. NCryptfs [35] provides kernel-level encryption services. It allows convenient file sharing between users located on the same machine. However, support for file sharing in more general situations is limited.

More advanced file systems were designed for securing remote storage systems and for allowing more flexible file sharing between users. Most of these systems trust the file servers and concentrate on protecting against malicious users accessing the network. Such systems usually include mechanisms that ensure proper authentication of the users. SFS [23] also authenticates the server, so that adversaries are prevented from masquerading as the server. Communication with the server is protected by a session key. However, trusting the server with the data enables attacks where adversaries collude with the server.

SGFS [18] offers secure, efficient and flexible global file sharing. NASD [11] provides security to network attached storage, where the file server is removed from the data path. Improved performance is gained by the direct interaction of users with the storage device. In this architecture, the storage devices themselves participate in cryptographic protocols. However, they are trusted with the data, and the data is stored in the clear. Thus, the attacks mentioned above are still relevant.

The strictest trust model used by related work avoids trusting the entire storage infrastructure. In such systems, the data is encrypted by writers before it is sent to the server, and decrypted by readers after it is received from the server. Access control is achieved by cryptographic means, and does not rely on the file server's mechanisms. Plutus [16], SiRiUS [12] and SNAD [25] are cryptographic file systems that enable secure file sharing over untrusted servers. However, eliminating the trust in the server comes at the expense of performance. Relaxed assumptions, such as those used by some variants of SNAD, allow improved performance in cases where the server is not completely untrusted.

SiRiUS is designed to be layered on top of any existing file system (but was implemented only over NFS). It implements in-band key distribution, while relying on secure public-key servers or IBE [6] master-key servers. Plutus provides efficient random access, file name encryption and lazy revocation, but does not relate to key distribution mechanisms. Plutus, SiRiUS and SNAD provide end-to-end encryption of data, and thus prevent adversaries from accessing files, despite having access to the physical storage device. Plutus and SNAD, unlike SiRiUS, place some trust on the server's access control mechanisms by requiring it to perform checks before committing users' requests.

The designs of Plutus, SiRiUS and SNAD rely on public-key cryptography. SNAD suggests using a symmetric HMAC as an alternative to signatures, as long as the server can be trusted for differentiating readers from writers, thus deviating from the untrusted storage model. Our work follows the research and suggestions of [26] for securing untrusted storage without public-key operations. In our work, we incorporate some of the suggested alternatives into a complete file system design.

There are many other secure file systems whose foci differ from ours. SUNDR [22,24] addresses important consistency issues of untrusted servers. OceanStore [20] and FARSITE [1] are secure large-scale distributed file systems. Availability and fault-tolerance issues are handled in these systems by replicating data and by using cryptographic techniques that allow discarding bad information while preserving useful information.

1.3 Contributions

This paper introduces CRUST, a new stackable file system layer designed to provide secure file sharing over untrusted servers. CRUST is intended to be layered over any existing file system, even a system that does not offer file sharing at all. The underlying file system is not modified, but is rather extended through the CRUST layer. Our design completely avoids the use of public-key cryptography in order to achieve better performance than existing systems, without sacrificing security properties.

Our basic design follows the direction of SiRiUS [12], but uses several methods suggested by Naor et al. [26]. The latter work identifies symmetric-key alternatives for expensive public-key operations. We incorporate some of those techniques and introduce a complete, practical file system design. We also introduce a novel key regression mechanism that allows efficient user revocation while avoiding the use of long hash chains.

CRUST is designed and implemented in a portable way, requiring a minimal installation process and supporting any underlying file system. The user is required to keep a small number of keys, which can be easily managed. The keys are provided by the user only when he mounts the file system. This allows existing applications to operate normally on the mounted file system without the user's intervention.

CRUST stores the data in encrypted and signed form, so it needs a key distribution mechanism. We chose not to use a Kerberos-like online trusted key distribution center (cf. [32]). Instead we use the Leighton–Micali key predistribution scheme [21], which requires the involvement of a trusted agent only during system set up. This approach enables file sharing without the need for a secure server, and does not require online communication between the users.

Like SiRiUS [12], CRUST offers flexible sharing policies by maintaining per user access privileges for each file, and differentiates between file ownership, read-only and read-write privileges. However, instead of relying on the asymmetry between the secret signing key and public verification key, we used a MAC-based signature scheme [26]. Furthermore, CRUST performs random access (both read and write) to different parts in a file and maintains files of varying sizes. Finally, CRUST includes a novel key regression mechanism that allows efficient user revocation, which may be of independent interest.

CRUST is implemented over the FUSE framework [33] on Linux. Our work includes an extensive performance evaluation. The results show that CRUST performs very well with only 2% overhead for reading large files and 8% for writing.

The rest of this paper is organized as follows. In Sect. 2 we define our goals and assumptions. Section 3 presents the mechanisms and design of CRUST. Section 4 introduces a novel key regression method. Section 5 describes the data organization used in CRUST. Section 6 presents the implementation of CRUST, and Sect. 7 evaluates its performance. Section 8 presents several extra features and security enhancements that can be added to CRUST. We summarize our results in Sect. 9. The details of basic CRUST operations appear in Appendix A.

2 Design requirements

2.1 System considerations

Stackable file system

CRUST must function as an add-on, providing secure file sharing over existing, unmodified file systems. It should support any system with the basic capability of storing files. This requirement is crucial when the user has no control over the file system, as in the case of using storage services on the web. Moreover, it also enables utilizing CRUST for a wider variety of uses, such as securing removable storage devices.

File sharing

CRUST provides flexible per file sharing, based on the taxonomy of [29], where every user can act as one of the following players:

- ***Owner***—The owner, who creates the file, can read, modify and delete it. The owner provides read and write permissions to other users, and may also revoke users' privileges.
- ***Reader***—The readers are permitted to read the file.
- ***Writer***—The writers are permitted to read and modify the file's data.

This is similar to the approach taken by SiRiUS [12], which relied heavily on public-key cryptography in order to distinguish between readers and writers. Our goal is to provide the same functionality using only symmetric cryptography.

Performance

CRUST should perform comparably to its underlying file system, minimizing access time and storage space overheads. In particular, we require random access, which is the ability to access arbitrary parts of a file without processing the entire file. User revocation is another operation that needs special care in order to be done efficiently in cryptographic file systems.

Key management

CRUST users should be able to access the file system securely using just a small number of keys—and still allow per file, per user access control with a read/write/own granularity. Key exchange should not require any online message exchange.

2.2 Security considerations

Data confidentiality, integrity and authenticity

File data must be unreadable to unauthorized users, despite having access to the physical storage device. No entity should be authorized unless explicitly granted permission by the file owner. In particular, the storage server's administrator is not trusted. Note that an attacker with access to the physical storage device can, of course, erase or modify the encrypted data. However, unauthorized modifications should be detectable by the CRUST clients.

Meta-data confidentiality, integrity and authenticity

File meta-data must also be protected against unauthorized modifications. Unprivileged modifications to meta-data should be detectable, as in the case of file data. However, we do not require complete confidentiality of meta-data, assuming that the non-confidential part does not reveal any sensitive information. Specifically, file names, directory structure and file sizes are not regarded as secret in the current version of CRUST. Access lists and file names can be encrypted using the methods described in Sect. 8.1. Implementing such features in CRUST is left for future work.

Cryptographic access control

CRUST must enforce per file access control. Users should not rely on any access control performed by the server. Access control is enforced by cryptographic means, and involves the three security requirements mentioned above: confidentiality, integrity and authenticity of file data.

Key distribution

CRUST is based on secret key cryptography that requires key distribution. However, we chose not to rely on out-of-band mechanisms (unlike Plutus [16]), and not to use a Kerberos-like key distribution center [32]. Instead, each user shares a long-term key with every other user. These keys are distributed efficiently using the Leighton–Micali key pre-distribution scheme [21], which uses public meta-data stored on the server. File-specific keys are provided by the file owner to the file readers and writers via the file's meta-data (encrypted separately using the keys shared by the owner and each other user). Thus, a secure online channel between the users is not required. Moreover, this approach allows privilege modifications such as user revocation to be done when the target users are offline.

Untrusted storage server

In CRUST, users do not rely on the server to provide any level of security. We assume that an adversary may be able to read, change or destroy arbitrary data stored on the server. Confidentiality, integrity and authenticity are achieved using cryptographic means performed by the CRUST clients. Availability of stored data, however, cannot be assured in our threat model. Our work does not present new methods for improving availability; existing methods are described in Sect. 2.3.

Trusted client machine

Users trust their local machines to securely handle their data and keys. However, we do not rely on secure communication between different machines.

A trusted agent during setup

Since CRUST provides its own key distribution mechanism that does not involve direct communication between the users, a trusted agent must set up the system. The agent is trusted to securely keep his keys and to not deceive the users with incorrect keys. Once the system is set up, the trusted agent is not needed any more. In fact, he can forget his keys as long as new users do not need to be added. The agent's possession of the keys allows him to access any information shared between the users; however, we reduce the trust in the agent by preventing him from accessing users' private information. Thus, the agent may only access files that are shared between the users.

We summarize our trust model in Fig. 1.

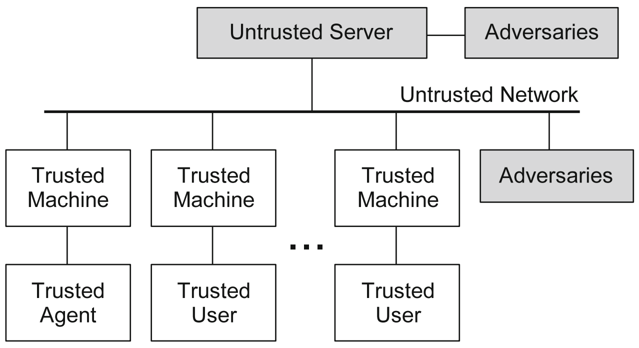

Fig. 1 The CRUST trust model. Untrusted entities are *grayed*

2.3 Adversary model

Since the server is untrusted in our threat model, it may be fully controlled by an attacker. This control includes reading, writing and destroying arbitrary data stored on the server. The possible attackers include malicious users and other network users, the server administrators and any other people that may have physical access to the server. Since all these entities have the same abilities in our model, even when they collude in a joint attack, we refer to them in the common name—adversaries.

Full control of the server by an adversary allows rollback attacks [24]. In these attacks, an adversary replaces all data and meta-data of a file with earlier, valid versions of them. Following this attack, the file users would read stale data and the file access privileges would be restored to their earlier state. Restoring the access privileges prevents recently privileged users from accessing the file, but also revives the privileges of revoked users. We have not implemented counter-measures against rollback attacks—see Sect. 8.2 for some possible solutions.

Recall that data availability is an open problem in our threat model, because adversaries may destroy the stored data at any time. Preservation of data may be improved, for instance, by replicating it on multiple servers. Currently, CRUST does not address availability issues. Nevertheless, since CRUST is a stackable file system layer, it may easily be layered over existing systems that include availability techniques, such as FARSITE [1] or OceanStore [20].

3 System design

3.1 Overview

Our basic design follows the direction of SiRiUS [12], but uses several symmetric-key alternatives for expensive public-key operations. The data structures in our design can be divided to global structures and per file structures. The global data structures maintain the list of users in the system and allow key agreement between the users. These structures are instatiated by the trusted agent as part of the system initialization, but are not secret.

The per file structures are maintained in two parts. Each part is stored in a different file on the underlying file system: the *data file* and the *meta-data file*. The data file keeps the encrypted file data. The meta-data file contains additional information required for key management and for authentication purposes. The details of all these structures are explained below.

3.2 The user table

CRUST users refer to each other by their *user names*. Such references occur when, for instance, a file owner decides to grant access privileges to another user. However, it is a common practice in file systems to efficiently represent users in meta-data by using specialized, unique numbers called *user IDs*. The users are normally associated with their IDs upon their registration, by the administrator or by the file server. Translations between user names and IDs often rely on the server.

However, CRUST users do not trust the file server for any purpose, including user ID translations and listing the registered users. Furthermore, we wanted to support file systems with no user management capabilities. Therefore, we decided to supply our own secure method for listing the users and translating between user names and IDs. We call this mechanism the *user table*.

The user table structure consists of a list of entries. Each entry contains a user name and its associated ID. This information is publicly stored on the server, in plain text form, by the trusted agent. We believe that this meta-data does not have to be confidential. However, it must be authenticated in order to prevent attackers from masquerading as arbitrary users.

The trusted agent shares a separate secret key, $K_i^{\text{User table MAC}}$, with each user i. This key is assumed to be securely exchanged between the trusted agent and each user during system initialization. The key is derived from a master key, denoted by $K_{\text{Master}}^{\text{User table MAC}}$, that is generated and maintained by the trusted agent. The shared key is derived using a one-way function based on the user's ID i, i.e., $K_i^{\text{User table MAC}} = h(K_{\text{Master}}^{\text{User table MAC}}, i)$. The user table is authenticated by the trusted agent for every user, by performing a MAC on the table contents with their shared key. The array of MACs is appended to the stored table. Each user checks the authenticity of the table by verifying his own MAC. Note that the symmetry of MACs requires that each user shares a separate key with the administrator, so that one user cannot be deceived by other users who can modify the table and calculate the correct MACs.

Only the trusted agent may add or remove users from the system—which can be done at any time provided the trusted agent still has its keys. However, we require that the same ID is never reused for different users, even if users are revoked, in order to prevent conflicts between different versions of the user table. This requirement is also crucial for the security of the users' key derivation procedure that is described in the next section. CRUST assures this requirement by allocating monotonically increasing IDs to new users. For this purpose, the trusted agent keeps track of the last allocated ID and increases it every time a new user is added.

3.3 Key distribution

Sharing encrypted files between several users requires some form of key distribution. Whereas Plutus [16] depends on an external out-of-band mechanism for key distribution, we argue that such a requirement is not practical in many cases. We prefer the alternative concept of in-band key distribution, which we adopt from SiRiUS [12] and modify to use only symmetric-key primitives.

This solution uses a file's meta-data for distribution of all the keys relevant to that file. Each user with some access rights to the file is allocated meta-data space, which we call a *lockbox*. The lockbox contains encrypted information shared with the file's owner. In SiRiUS, the information is encrypted using the target user's public key, and relies on a public-key infrastructure (PKI) for exchanging users' public keys. CRUST uses an alternative symmetric-key method, in which every pair of users shares a *common secret key*. For exchanging these keys, we use one of the methods of Leighton and Micali [21]. A brief description of the protocol follows (see also [26,30]):

The trusted agent generates two *master secret keys*, K and K'. Using these keys and a pseudo-random function $h(\cdot)$, the trusted agent computes and provides each user i with his *exchange key* K_i and his *individual authentication key* K_i', where:

$$K_i = h(K, i),\ K_i' = h(K', i).$$

In practice, we use the HMAC algorithm as the pseudo-random function. The keys K_i and K_i' are given to each user during system setup in a secure, out-of-band method. Additionally, a public database is published by the trusted agent. It contains two matrices, P and A, including *pair keys* and *authentication keys*, respectively, such that:

$$P_{i,j} = h(K_i, j) \oplus h(K_j, i),\ A_{i,j} = h(K_i', h(K_j, i)).$$

Each user i that wants to encrypt information for user j computes the common secret key $K_{i,j}$ according to:

$$K_{i,j} = P_{i,j} \oplus h(K_i, j)\ \ \big(= h(K_j, i)\big).$$

User i can verify the key's authenticity by ensuring that:

$$h(K'_i, K_{i,j}) = A_{i,j}.$$

The decrypting user j can calculate $K_{i,j} = h(K_j, i)$ without reading the matrices.

In our scenario, when a file owner, user i, shares a file with user j, he creates a lockbox for user j in the meta-data file, encrypted with the key $K_{i,j}^{\text{Enc}}$ and authenticated by applying a MAC with $K_{i,j}^{\text{MAC}}$, where:

$$K_{i,j}^{\text{Enc}} = h(K_{i,j}, \text{“Enc”}), \; K_{i,j}^{\text{MAC}} = h(K_{i,j}, \text{“MAC”}).$$

User j derives these keys from $K_{i,j}$ in the same manner. The lockbox contains file-specific meta-data needed by user j for accessing the file, as we further explain in Sect. 5.2.

A file owner also needs to store private information that is not shared even with privileged writers. For example, Sect. 4 shows that the owner stores some information allowing derivation of future encryption keys to be used when a reader or writer is revoked. Therefore, the owner maintains a private lockbox as well. Encryption and MAC keys are used for encrypting and authenticating the owner's lockbox, just as any other lockbox. However, although an owner i could use the keys $K_{i,i}^{\text{Enc}}$ and $K_{i,i}^{\text{MAC}}$, which are a part of the key distribution scheme, we prefer that every user generates his own keys for this purpose. These keys, denoted by $K_i^{\text{Self enc}}$ and $K_i^{\text{Self MAC}}$, respectively, are generated by the user upon initialization and are kept securely. They allow storing secure private information that is unavailable even to the trusted agent.

3.4 Access control

Data confidentiality in CRUST is maintained by encryption of data at rest performed by the client. Each file is associated with its own encryption key. This key is generated by the file owner upon the file's creation and is securely distributed to the file readers and writers through their encrypted lockboxes. Unprivileged users cannot obtain the key, since they cannot decrypt other users' lockboxes.

Data integrity and authenticity in earlier cryptographic file systems, such as SiRiUS [12] and Plutus [16], are provided by public-key digital signatures. A signature on the hash of a file proves that the file was written by an authorized writer having the private key. Therefore, readers and writers are distinguished by the asymmetry between the signature's private and public keys. In CRUST, we instead use an efficient MAC-based symmetric-key signature scheme, as suggested in [26]. Note that although public-key signatures achieve the additional property of *non-repudiation*, we believe that it is not a common requirement in file systems.

Since MACs are symmetric by themselves, we need a special construction to provide writer-reader differentiation. In CRUST, the encrypted file data is signed by the file writers and is verified by both the file readers and writers. This is done by utilizing multiple MACs: one MAC is stored in the meta-data for the writers, and an additional MACs is stored for each reader. A CRUST signature operation involves calculating and storing all the MACs, whereas a verification operation involves calculating and comparing a single MAC. The file owner randomly generates a *file master MAC key*, denoted as $K^{\text{File MAC}}$, during file creation. This key is distributed to the writers through their lockboxes. Each reader is handed a separate *file reader MAC key*, denoted as $K_i^{\text{File MAC}}$, where i is the reader's user ID. The reader keys are derived from the master key by applying a one-way function, i.e., $K_i^{\text{File MAC}} = h(K^{\text{File MAC}}, i)$. This scheme simplifies key management, because writers can compute the MAC keys for all readers, based on the master key and the list of readers.

Signature time and space grow with the number of readers, while verification time is always as short as one MAC computation. The space consumption for the signature is acceptable in our case, because we maintain a per reader lockbox anyway.

3.5 Random access

Efficient random access is an important feature in file systems, because it improves the performance of the system in some applications by orders of magnitude. In order to support random access, both encryption and authentication need to operate in smaller chunks than the entire file. For this purpose, CRUST divides the file data into blocks of a predetermined size (we used 4,096 bytes). Encryption is performed on each data block independently. More details about the encryption techniques are given in Sect. 5.1.

Authentication of file data by signing the hash of the entire file prevents efficient random access since it requires the file to be entirely obtained in order to verify or update a single block of data. Instead, we use a more efficient scheme by hashing the data in a *hash tree* construction; each leaf of the tree stores the hash of a single block of data, and each internal node stores the hash value computed on the concatenation of the values in its children. The hash tree is stored as a part of the file's meta-data and is not secret because the data blocks being hashed are in encrypted form. The file data is authenticated by signing the tree's root instead of the hash of the entire file. Thus, verifying a block of data only requires: (a) calculating its hash, (b) comparing it to the hash in the relevant leaf of the hash tree, (c) checking the consistency of the hashes on the branch connecting the leaf to the root, and (d) verifying the signature of the root. The hash tree structure is demonstrated in Fig. 2.

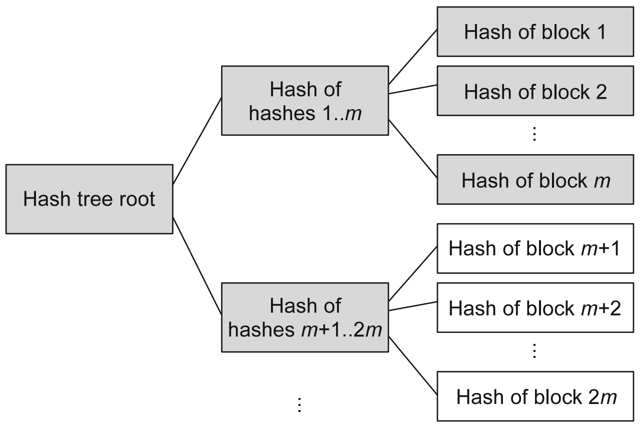

Fig. 2 The hash tree structure, where each internal node has *m* children. The *grayed* nodes need to be accessed for verifying any block in the range 1...*m*

3.6 Owner-identifying file names

The owner of a file must be known in order to access the file properly. For instance, a user must know the identity of the owner so he can select the keys used for decrypting and authenticating his lockbox.

Naive methods of owner identification are vulnerable to the following attack, which was mentioned in SiRiUS [12]: a malicious user can replace an existing file with a file that the malicious user owns. Unless the true owner is known in advance, users can not detect this situation. Furthermore, any data written by a non-suspecting file writer, following the replacement, would be readable by the malicious user. One solution suggested in SiRiUS requires publishing a small amount of information on a secure server, i.e., deviating from the untrusted storage model. Another solution requires every owner to occasionally verify the ownership of his files. The latter solution is more practical, but is still vulnerable to attacks. For example, a malicious file server may hide the replacement from the true file owner by pretending to store the original file's contents only when being accessed by the owner. Thus, the replacement may not be detected by the file owner.

CRUST uses the following solution, which we call *owner-identifying file names*. The CRUST root directory contains a sub-directory per each user, so that all the files owned by a user reside under a specific directory named after that user. For example, if a Linux user *alice* mounts the CRUST file system at her `~/crust` and she is logged into CRUST as Alice, then she can access a typical file owned by her using the file name `~/crust/Alice/foo.c`. Another Linux user *bob*, logged into CRUST as Bob, can mount CRUST at his own `~/crust` and then he can access Alice's file using the same name, `~/crust/Alice/foo.c`. The per user directories are created when users are added (i.e., during system setup).

This way, when a file name is known, the owner is automatically determined. Once the true owner is known, a file user can verify the authenticity of the file's meta-data using the authentication key that is secretly shared between the file user and the owner. Note that *creating* a file in another user's directory is forbidden by CRUST since the ownership of such a file would be ill-defined.

3.7 Key management

A CRUST user i only needs to store five long-term keys: K_i, K'_i, $K_i^{\text{User table MAC}}$, $K_i^{\text{Self enc}}$ and $K_i^{\text{Self MAC}}$. The first three keys (the Leighton–Micali keys and the user table authentication key) are securely provided to the user by the trusted agent, whereas the two other keys (the personal encryption and authentication keys) are generated by the client and are kept secret. No further keys are required once the file system is mounted. This way, applications can access the files transparently, without the user's intervention.

During system initialization the trusted agent is required to keep track of three long-term keys. The trusted agent's keys are K, K' and $K_{\text{Master}}^{\text{User table MAC}}$ (the Leighton–Micali master keys and the master key for user table authentication), which are generated and kept secret by the agent. The agent can discard (or lose) the keys at any time, but this will prevent adding new users to the system after that moment.

Since each CRUST file maintains its own short-term keys and access control information, a file system backup can be performed by unprivileged users. A backup program can even detect only the modified files and perform the backup over the underlying file system, without mounting CRUST and without knowing any keys.

3.8 Limitations

In order to improve overall performance, CRUST uses several symmetric-key alternatives for expensive public-key operations. However, the use of such mechanisms does introduce some limitations and trade-offs with respect to their public-key counterparts, as follows.

The key distribution alternative introduced in Sect. 3.3 requires global storage space that grows quadratically with the number of users in the system (for the Leighton–Micali data structures). We believe that such space overhead is acceptable on most practical uses, since the number of users is typically dwarfed by the number of files and their combined size.

The access control mechanism described in Sect. 3.4 requires creating and storing a MAC for each reader of the file. Thus, modifying a file requires time that grows with the number of its readers. This may not be efficient for files with many readers. However, signature time is not affected by the number of writers.

Finally, our scheme does not distinguish well between multiple writers of same file since all the writers share a single MAC key. Thus, if there is more than a single user with write premission to a file, it is impossible to determine which of these authorized writers did in fact update the file.

4 Revocation without re-encryption

4.1 Lazy revocation

The event of revoking a user's access from a file modifies that file's access permissions. Thus, keys currently used for encrypting the file must be replaced. However, re-encryption of the entire file with the new keys is an expensive process. The concept of *lazy revocation* states that re-encryption may be delayed until the next update to the file. Moreover, re-encryption may be performed partially, just for the file blocks that are being updated. This concept makes sense because the revoked users may have already read all the data stored prior to their revocation.

In CRUST, the lifetime of a file is divided into *epochs*, where each revocation begins a new epoch. New file encryption and signature keys are generated at the beginning of each epoch. Updating the file is always done by writing blocks that are encrypted with keys of the latest epoch, whereas reading may require older keys, depending on the epochs when the relevant blocks were written. Therefore, older keys must be available to the users as long as they are being used. However, associating each epoch with its own independent key implies that the space required for maintaining the keys would grow linearly with the number of revocations. A more efficient scheme would use a relation between the keys, so that the most recent key can be used to derive the key of any earlier epoch.

Plutus [16] introduced the *key rotation* technique, where the owner uses a dedicated private key to generate a new encryption key from the current one. The most recent encryption key is handed to the users of the file. The users can derive all previous keys from the current one by using the dedicated public key. Plutus uses key rotation for a different purpose than ours, where the epoch (or version) was a property of each file in a certain group of files.

The approach we take in CRUST is based on the more generalized paradigm of *key regression* [10]: instead of providing the users with the latest encryption key directly, we provide them with a *state*, which is a small structure that allows computation of the current key and all earlier keys. Full details of our key regression method appear in the next section.

The infrastructure for using a key regression mechanism in CRUST involves sharing the latest key regression state with each of the users that are privileged to access the file. The owner provides the key regression state in the users' encrypted lockboxes. Moreover, in order to keep track of each block's epoch, an array of epoch identifiers is maintained as a part of the file's meta-data. This array is not secret, but it requires authentication; see Sect. 5.2. Existing blocks' encryption keys are derived from the current state. Newly written blocks are encrypted with the key of the most recent epoch, hence a revoked user cannot decrypt them.

4.2 The hash matrix

4.2.1 Overview

In this section, we present an efficient key regression scheme using only symmetric-key cryptography. In comparison, the key rotation mechanism presented in Plutus [16] (which can be extended to a key regression mechanism [10]) relies on public-key cryptography, since it uses a dedicated private key to generate a new encryption key from the current one. A straight-forward symmetric-key alternative mentioned in Plutus, is the *hash chain*.

The hash chain method uses a one-way hash function for deriving an older encryption key from the current one. In order for the owner to compute the first key, the entire chain of n keys must be calculated upon initialization. Thus, the initialization begins by generating K_{n-1}, a random master key. The other keys in the chain are then computed according to the formula $K_i = h(K_{i+1})$, where $h(\cdot)$ is a cryptographic hash function. K_i is the file encryption key to be used after the ith revocation. The number i is associated with every block, and represents the block's epoch. The last key of the chain, K_0, is the first one to be used.

In the hash chain method, when a revocation occurs, the owner calculates the next key and provides it to the users. The most recent key can be used by any user to derive all earlier keys, by applying the hash function for an appropriate number of times. The owner keeps private information that allows calculating the new keys after revocations occur. He can either store the entire chain or as little as the master key alone. The amount of information stored affects the key derivation time. More specifically, there is a trade-off between the two, where the memory-times-computational complexity is $O(n)$ per key derivation. Jakobsson [15] presented an improved technique for efficiently deriving the keys without storing the entire chain; however, all hash chain-based techniques suffer from an inefficient initialization (i.e., file creation) process, because it requires calculating the entire chain. Since the length of the chain limits the number of lazy revocations, which is desired to be as large as possible—and we need a separate hash chain *per file*—the hash chain technique is inefficient for our purposes.

Instead of the naive hash chain, we present a new key regression mechanism called the *hash matrix*. It enables both initialization and key derivation to be done efficiently, in aspects of memory and computational complexity. The hash matrix can be seen as a generalization of the hash chain, where multiple keys are derived from every single key, by using multiple one-way functions. This multiplicity of one-way functions shortens the computation paths dramatically, in comparison to a hash chain supporting the same number of total available keys.

Our mechanism is reminiscent of the binary-tree key-updating scheme (TreeKU) of [2], where the keys are derived in a tree-like manner. The general goals of the two mechanisms are similar, though the mechanism to be used in our system has a stricter space limit. In TreeKU, a key can be derived into two other keys by using it as a seed for a pseudo-random generator, and taking two parts of its output as the derived keys. The keys are conceptually arranged in a complete binary tree of depth d, where a master key is used as the tree's root. The edges indicate which part of the pseudo-random generator's output is taken for deriving each key from its parent. The maximal number of pseudo-random function applications required in order to compute any key in the tree is d. The total number of available keys is $n = 2^{d+1} - 1$. TreeKU requires storing a table of up to $d + 1$ keys for maintaining the key-updating state; each key can be computed from an adequate state by applying the pseudo-random function at most d times. TreeKU security is an immediate consequence of the well-known construction of pseudo-random functions [13]. Both space and time bounds are logarithmic in n, but we seek for a technique that allows storing less than $\log_2 n$ keys, without harming the computational complexity significantly. The scheme we suggest in the following section achieves these goals.

4.2.2 Scheme details

In our technique, the keys are arranged in a d-dimensional matrix of uniform length m. The total number of keys in our key regression system is $n = m^d$. Each key K_i is identified by a base-m number i indicating its position in the matrix (i can also be seen as the epoch identifier). We use d different one-way functions $f_k(\cdot)$ for $k = 0, 1, \ldots, d - 1$ in order to define the keys. The master key K_{n-1} is generated randomly when the key regression system is initialized (i.e., at file creation time). Each key K_i can be derived from the master key by a unique *computation sequence*, in which each step includes an application of a one-way function on the previous step's result. Let $b_k(i)$ for $k = 0, 1, \ldots, d - 1$ denote the kth base-m digit of i, where $k = 0$ refers to the least significant digit. The computation sequence of K_i from K_{n-1} consists of several consecutive applications of each one-way function, starting with $f_{d-1}(\cdot)$ and ending with $f_0(\cdot)$. For each k, $f_k(\cdot)$ is applied $t_k(i)$ consecutive times during the

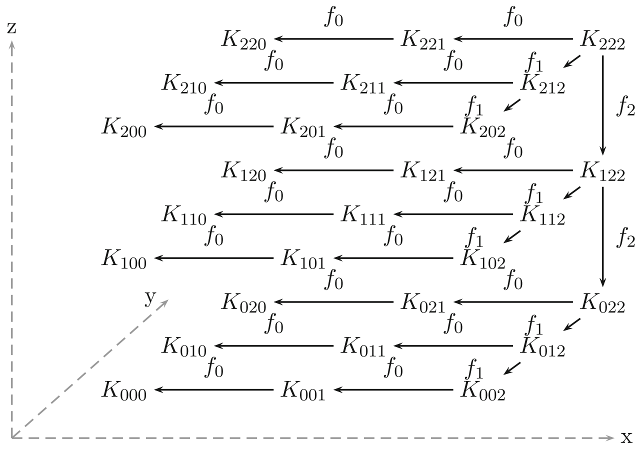
z
y
x
K_{222} K_{221} K_{220} K_{212} K_{211} K_{210} K_{202} K_{201} K_{200}
K_{122} K_{121} K_{120} K_{112} K_{111} K_{110} K_{102} K_{101} K_{100}
K_{022} K_{021} K_{020} K_{012} K_{011} K_{010} K_{002} K_{001} K_{000}
f_0
f_1
f_2

Fig. 3 The hash matrix for $m = 3$, $d = 3$. *Arrows* indicate the key computation sequences

sequence, where:

$$t_k(i) = (m - 1) - b_k(i).$$

Figure 3 demonstrates the hash matrix for $d = 3$ dimensions, with 3 one-way functions, each allowing up to $m-1 = 2$ activations. In the example, we obtain that $K_{012} = f_1(f_2(f_2(K_{222})))$ and $K_{122} = f_2(K_{222})$. Note that some keys have a computation sequence that is a prefix of other keys' computation sequences. This allows to derive some keys from others.

After the ith revocation, our scheme provides the users with a state, S_i, which is a structure allowing the computation of all the keys $K_0, K_1, \ldots, K_i$, but no others. The state S_i consists of a small group of keys, including K_i and some earlier keys, that enable the computation of $K_0, K_1, \ldots, K_i$. More specifically, the state S_i consists of d keys:

$$S_i = \langle K_i, K_{\mathrm{sub}(i,1)}, K_{\mathrm{sub}(i,2)}, \ldots, K_{\mathrm{sub}(i,d-1)} \rangle,$$

where $\mathrm{sub}(i, k)$ is the base-m number obtained by subtracting one from the kth digit of i, and maximizing all less significant digits. If the digit to be subtracted is zero, $\mathrm{sub}(i, k)$ is defined as zero, and $K_{\mathrm{sub}(i,k)}$ is not necessary for the state. In any case, $\mathrm{sub}(i, k) \leq i$.

The hash matrix mechanism includes three algorithms: an *initialization* algorithm, a *state derivation* algorithm and a *key extraction* algorithm. The initialization routine constructs S_{n-1} by simply choosing the master key K_{n-1} randomly, and computing $K_{\mathrm{sub}(n-1,k)} = f_k(K_{n-1})$ for $k = 1, 2, \ldots, d - 1$. The state derivation and key extraction algorithms compute S_i and K_i accordingly, given a more recent state, S_j, where $j >= i$. For $j = i$, the algorithms are trivial, since $S_i = S_j$, and K_i is included in S_j. We now explain the algorithms for the other case, $j > i$; please refer to Fig. 3.

First, we show the key extraction algorithm, which is later extended to the state derivation algorithm. Let k denote the position of the most significant digit differing i and j. Obviously, $b_k(i) < b_k(j)$. If $k = 0$ then K_i and K_j differ only in the number of applications of $f_0(\cdot)$, hence $K_i = f_0^{j-i}(K_j)$ (i.e., $j-i$ applications of $f_0(\cdot)$). Otherwise, observe $K_{\text{sub}(j,k)}$, which is included in the given state S_j. We claim that this key's computation sequence is a prefix of K_i's computation sequence.

This follows from the fact that the $d - (k + 1)$ most significant digits of i, j and $\text{sub}(j, k)$ are identical, the kth digit of i is smaller or equal to that of $\text{sub}(j, k)$, and each of the less significant digits of $\text{sub}(j, k)$ is maximal (i.e., equals to $m - 1$). The maximality of the less significant digits means that $f_{k-1}(\cdot), f_{k-2}(\cdot), \ldots, f_0(\cdot)$ were not applied at all for computing $K_{\text{sub}(j,k)}$. Therefore, what is left to do for completing the computation sequence of $K_{\text{sub}(j,k)}$ to that of K_i, is applying $f_k(\cdot), f_{k-1}(\cdot), \ldots, f_0(\cdot)$ for the appropriate number of times; first, $f_k(\cdot)$ should be applied $b_k(\text{sub}(j, k)) - b_k(i)$ times; then, each of the functions $f_{k-1}(\cdot), f_{k-2}(\cdot), \ldots, f_0(\cdot)$ should be activated $t_{k-1}(i), t_{k-2}(i), \ldots, t_0(i)$ times accordingly.

The key extraction algorithm is extended to the state derivation algorithm by computing all the keys contained in S_i, as follows. For $\hat{k} = k+1, k+2, \ldots, d-1$, which represent the most significant digits that are equal in i and j, $\text{sub}(i, \hat{k}) = \text{sub}(j, \hat{k})$, so $K_{\text{sub}(i,\hat{k})} = K_{\text{sub}(j,\hat{k})}$. For $\hat{k} = 0, 1, \ldots, k$, we relate to a specific intermediate key computed during the original algorithm, that was the result of applying $f_{\hat{k}}(\cdot)$ several times as indicated in the algorithm. We apply $f_{\hat{k}}(\cdot)$ on that key one additional time. This additional transformation is equivalent to the subtracted digit in $\text{sub}(i, \hat{k})$, and thus calculates $K_{\text{sub}(i,\hat{k})}$. Note that each of these calculations should be avoided if $b_{\hat{k}}(i) = 0$.

Finally, we remark that an additional hash function should be applied on each of the extracted keys K_i in order to derive the actual encryption keys to be used in our system: $K_i^{\text{Enc key}} = h(K_i)$. This final step of the key extraction algorithm ensures the pseudo-randomness of the encryption keys, which is a required property of key regression schemes [10].

We prove the security of the hash matrix by proving that it is computationally infeasible to determine K_i given $K_0, K_1, \ldots, K_{i-1}$ (note that this also implies that it is infeasible to determine K_i given $S_0, S_1, \ldots, S_{i-1}$ as well). This is done by generalizing the method of [13] to a d-ary tree; an informal proof follows. For every $j < i$, let k be the maximal number such that $b_k(i) \neq b_k(j)$. Obviously, $b_k(i) > b_k(j)$ and all the more significant digits of i and j are equal. Therefore, $t_k(i) < t_k(j)$. These facts imply that the unique calculation sequences of K_i and K_j share a common prefix. At the first step where the sequences differ, the calculation sequence of K_j contains an additional application of $f_k(\cdot)$ that does not appear in K_i's. This one-way transformation makes the computation of K_i given K_j infeasible. Since such a difference exists for every $j < i$, it is infeasible to determine K_i given $K_0, K_1, \ldots, K_{i-1}$.

4.2.3 Complexity analysis

The maximal space needed for storing a hash matrix state is equivalent to storing d keys. The time for computing keys and states is dominated by the one-way transformations. Initialization requires $d-1$ transformations. Extracting K_i involves applying the d one-way functions, no more than $m - 1$ times each. Thus, it involves at most $d(m-1)$ one-way transformations. Deriving S_i involves at most $d - 1$ additional one-way transformations, but is also bounded by $d(m - 1)$ total one-way transformations.

For practical reasons it is convenient to have n (the maximal number of user revocations per file) to be a power of 2, and also to have $m = 2^p$ for a small constant p, thus for a given d we get $n = 2^{dp}$. This choice of parameters implies that the maximal size of a state is equivalent to $\frac{1}{p}\log_2 n$ keys, and deriving a state or a key involves no more than $\frac{1}{p}(2^p - 1)\log_2 n$ one-way transformations.

In CRUST we use $p = 4$, i.e., $m = 16$, so that each base-m (hex) digit can be stored in a 4-bit nibble. We use $d = 7$, giving a maximum of $n = m^d = 2^{28}$ keys and $n - 1$ revocations without re-encryption. We use HMAC based on SHA-1 as the keyed one-way function. Thus the state S_i contains $d = 7$ hash values (20 bytes each), totaling as little as 140 bytes. Initialization, as well as key or state derivation, requires at most $d(m - 1) = 105$ HMAC operations. The 20-byte values extracted from the key regression mechanism are truncated to 16 bytes in order to be used as AES-128 encryption keys.

5 Data organization

5.1 Data file structure

Recall that a CRUST file is maintained using two files in the underlying file system: the data file and the meta-data file. The data file contains the encrypted file data. To allow random access, the file is divided into blocks, where each block is encrypted independently. The encryption keys are specific to each file. To allow efficient user revocation without file re-encryption, we use block-specific encryption keys (unlike SiRiUS [12], which encrypts the entire file with the same key). However, unlike Plutus [16], we do not use *independent* per-block keys and we do not use a stream cipher. Instead, all the block keys are derived from a single key regression state, based on the block's epoch. The key regression state and the array of block epochs are provided in the meta-data file. Our

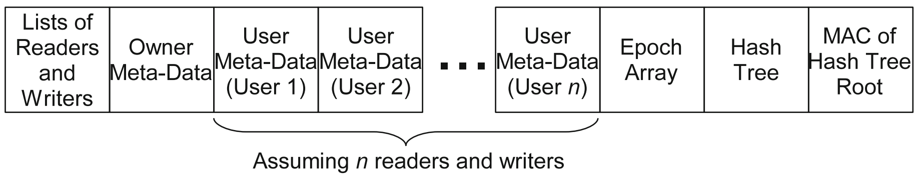

Fig. 4 Meta-data file structure. The hash tree root's MAC is calculated using $K^{\text{File MAC}}$

approach yields a revocation time that is independent of the file size.

Any block cipher can be used for encrypting the data file. However, since the same key may be used for encrypting multiple cipher blocks, and encrypting the same plain text under the same key produces the same output, not every *mode of operation* assures confidentiality. Our chosen mode of operation follows NIST's recommendation [9] to use CBC, performed on every file block separately, where the *initialization vector* for encrypting each block is the encryption of the block's offset in the file, with the same encryption key. This mode also allows efficient random access of the file blocks. However, encrypting multiple versions of data in the same file block is not recommended when using this mode. Security for such cases can be improved in future work by saving a per block initialization vector that is regenerated randomly on each update to the block.

CBC mode requires the plain text size to be a multiple of the cipher block size (16 bytes for AES [27]), so CRUST uses the following padding method. Let s denote the plain text size, in bytes, and let $k = s \bmod 16$. We first pad the plain text with $16 - k$ binary '0' bytes, and encrypt the result in CBC mode. To allow determining the original plain text size, the cipher text is then padded with k additional (unencrypted) '0' bytes. The only exception is when $k = 0$; in this case, no padding is used. For example, a 2-byte file is stored as an 18-byte CRUST data file, in which the last 2 bytes are nulls. This approach allows fast calculation of the plain text size, since it only depends on the padded data file's size, which is stored in its *inode*. Once the plain text size is determined, unpadding is done by removing the extra padding bytes from it. The unpadding routine does not ensure that the removed bytes are actually '0' bytes. As a result, all possible cipher texts of valid length are considered valid. This property of our padding scheme makes it resistant to valid-padding side-channel attacks [4].

5.2 Meta-data file structure

All the meta-data needed to maintain a CRUST file is stored in its associated meta-data file. The meta-data files are omitted by the CRUST client when files are being listed, so that the directory structure appears to the user as containing only the data files.

In order to support file data encryption using key regression, the master key regression state, denoted by KRS_{Master}, is privately kept in the owner's lockbox. The current key regression state, denoted by KRS_{Current} is distributed through the file users' lockboxes. The state includes the current epoch identifier. Additionally, the epoch array, containing the epoch identifier of each data block, is stored publicly in the meta-data file. The epoch array is not secret, but its integrity and authenticity must be verified. Therefore, it is signed along with the file data, as described below.

File signatures that allow efficient random access are achieved by using the hash tree. The tree is stored publicly in the meta-data file. The hash tree root is signed by storing the writers' MAC and an additional MAC for each reader, according to the scheme described in Sect. 3.4. The MAC keys are also distributed according to the scheme, through the lockboxes.

The user privileges for accessing the file are maintained by storing two lists of user IDs, describing the readers and writers, respectively. The lists are stored publicly in the meta-data file, since we assume that they are not secret (but if needed, they can be protected as described in Sect. 8.1). The access lists, denoted by $\text{ACL}_{\text{Readers}}$ and $\text{ACL}_{\text{Writers}}$, are signed using a MAC. The lists are also used for ordering the lockboxes in the meta-data file, so that each lockbox can be associated with its user.

File names in CRUST are maintained by the underlying file system, without using encryption. However, the file names must be verified in order to detect file-swapping attacks. Each file's (full) name is signed using a MAC, along with the access lists and each user's lockbox. This implies that CRUST files cannot be renamed or moved except by their owner.

We summarize the meta-data file structure in Fig. 4. A part of it includes user-specific information, stored separately for the owner and for each reader and writer. This user-specific meta-data is depicted in Fig. 5. Note that a user's meta-data includes his lockbox, which contains secret information. Each lockbox is encrypted and signed with secret keys shared by the owner and the addressed user. The rest of the meta-data in the file is stored once and is not secret.

Note that the only parts of the meta-data that a (non-owner) writer is privileged to modify, and are used by other users, are the epoch array, the hash tree and the MACs of the hash tree root. Readers, on the other hand, cannot feasibly modify any meta-data that is used by another user, without the modification being detected by that user. The owner can detect all

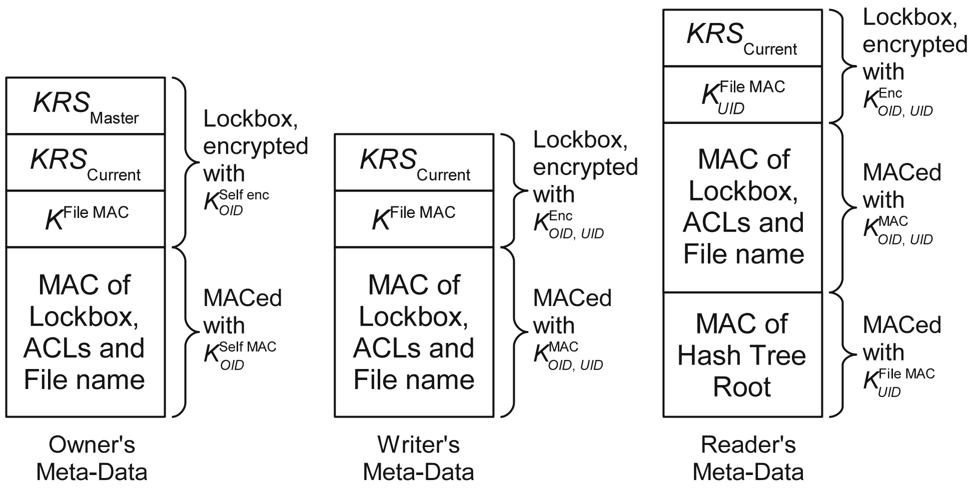
KRS_{Master}
$KRS_{Current}$
$K^{File\ MAC}$
MAC of Lockbox, ACLs and File name
Lockbox, encrypted with $K^{Self\ enc}_{OID}$
MACed with $K^{Self\ MAC}_{OID}$
Owner's Meta-Data
$KRS_{Current}$
$K^{File\ MAC}$
MAC of Lockbox, ACLs and File name
Lockbox, encrypted with $K^{Enc}_{OID,\ UID}$
MACed with $K^{MAC}_{OID,\ UID}$
Writer's Meta-Data
$KRS_{Current}$
$K^{File\ MAC}_{UID}$
MAC of Lockbox, ACLs and File name
MAC of Hash Tree Root
Lockbox, encrypted with $K^{Enc}_{OID,\ UID}$
MACed with $K^{MAC}_{OID,\ UID}$
MACed with $K^{File\ MAC}_{UID}$
Reader's Meta-Data

Fig. 5 User-specific meta-data. OID denotes the owner's ID, and UID denotes the addressed user's ID

modifications to the lockboxes, because they can be derived from information in the owner's lockbox.

5.3 Global meta-data structures

CRUST stores two public data structures in a constant location on the file server, which are accessed on demand by the users. These structures are the user table and the Leighton–Micali structure (recall Sects. 3.2 and 3.3). The user table structure consists of user name and ID pairs, and an array containing a MAC of the table for each user. The structure for operating the Leighton–Micali protocol consists of two matrices, containing the pair key and authentication key for every pair of users.

Note that storing and maintaining the global Leighton–Micali and user management structures are not necessary if dynamic user management is not required. The system will keep functioning properly even if the structures are not available, as long as new users are not being introduced to the system. In a static user management situation, the structures can be distributed to the users during system initialization, instead of storing them on the server.

6 Implementation details

6.1 Overview

We implemented CRUST on Linux using the FUSE (Filesystem in Userspace) framework [33] and the OpenSSL cryptographic library [34]. FUSE provides an interface for user-level programs to export a VFS to the Linux kernel. It also enables non-root users to mount their own file systems. FUSE consists of a kernel module, a user-level library and a mount utility. The user-level file system communicates with the FUSE kernel module through a dedicated device file.

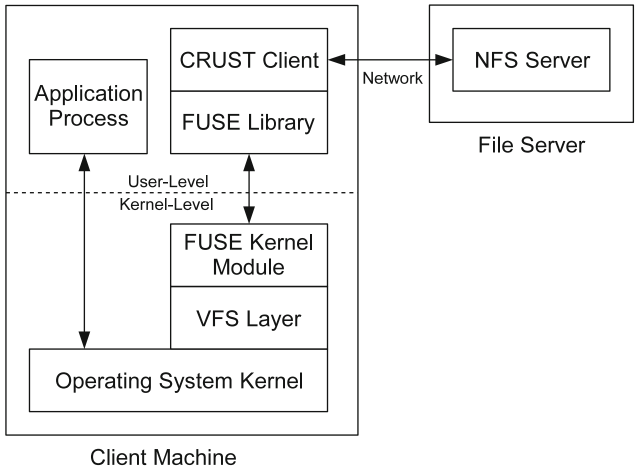

Fig. 6 Architecture of CRUST layered over NFS. Note that NFS is just an example. CRUST can be layered over any file system

CRUST is implemented as a user-level client providing the data and meta-data of the file system through the FUSE interface. Installing the CRUST client requires the FUSE kernel module to be installed by the root user. However, since Linux kernel versions 2.6.14 (released in 2005) and later include FUSE support out of the box, CRUST can also be installed by a non-root user on recent enough Linux operating systems.

We designed CRUST as a stackable file system layer [14]. It appears to the user as an ordinary file system, while functioning as a layer over another file system. Requests are processed by the user-level client by communicating with the underlying file system. As an additional layer, CRUST provides strong security to any insecure file system. The underlying system is not known in advance and it is not modified. It is only assumed to have basic functionalities. In fact, it may be any local or network file system, such as ext2 or NFS. Figure 6 shows the general architecture of CRUST in a typical scenario.

The OpenSSL library supports many cryptographic primitives that can be used for the various mechanisms of CRUST. The default options used by CRUST are the following widely-used primitives: SHA-1 [28] as the cryptographic hash function, HMAC [3] based on SHA-1 as the MAC algorithm and AES-128 [27] as the block cipher.

6.2 Software components

CRUST was implemented according to the design presented in Sect. 3. The implementation includes a mount utility (`crust-mount`), a file sharing utility (`crust-chmod`) and an administration tool `crust-admin` for the trusted agent. A user logs into CRUST by executing `crust-mount` and providing the CRUST mount point (e.g., `~/crust`) and the underlying file system's mount point (e.g., `/nfs`). The mount utility runs a CRUST client daemon in the background, which intercepts and handles Linux file requests on the CRUST mount point. FUSE guarantees that the mount point is private for the mounting Linux user; additional users on the same machine must mount CRUST on other mount points by running the mount utility (and daemon) independently. Note that the mounting Linux user can identify himself as any CRUST user, as long as he possesses the required keys.

Per file access permissions are managed using `crust-chmod`. This tool resembles the Linux `chmod` command, but is more expressive, because the file access privileges of each user are controlled individually in CRUST. Command-line options of this tool include `add-writer`, `add-reader` and `revoke-user`.

The system is initialized by the trusted agent using the `crust-admin` tool. The same tool also provides user management functionalities and publishes the public structures of the key distribution protocol when a user is added or removed from the system. Command-line options of this tool include `init`, `add-user` and `remove-user`.

The client daemon (which is started by `crust-mount`) consists of two main components: user interaction and file management. The user interaction component is responsible for securely listing the system's users and providing translations between user names and IDs, as well as computing the common secret keys shared with each user. This functionality is handled by reading and authenticating the public meta-data structures stored by the trusted agent. The user interaction component also manages the user's master keys, as described in Sect. 3.7.

The file management component of the client implements all the file-specific operations. Standard operations such as file opening, reading and writing are directed to FUSE through the VFS interface [19]. File sharing operations in CRUST, however, are more expressive than the standard interface and are therefore directed in a special manner; `crust-chmod` expresses sharing operations as specially-formatted Set Extended Attribute (SETXATTR) operations, which are available in the VFS interface; `crust-chmod` encodes the various parameters of a file sharing request, as given in the command-line, into one string. This string is used as an "attribute" being set for the relevant file by initiating a SETXATTR system call. When the CRUST client intercepts this call, it decodes the file sharing command parameters from the attribute string and finally executes the command.

6.3 Optimizations

6.3.1 Caching

Caching of data and meta-data has a great impact on the performance of file systems in general and cryptographic file systems in particular. It can save precious time by preventing redundant cryptographic and file operations. For instance, consider an application reading a 4 KB CRUST file, one byte at a time. Since cryptographic operations are performed on 4 KB blocks, reading each byte separately involves reading, decrypting and authenticating the entire block every time. A simple data cache would perform this lengthy process only for the first byte in the block, and keep the obtained data for instantly handling the next requests.

CRUST maintains several caches. Note that all caches are maintained by the client software, so each user utilizes his own caches. We employ a data cache that keeps recently read blocks of file data, as demonstrated in the above example. It is a *write-back* cache, in which modified data is kept in memory and is not written or even encrypted until absolutely necessary. This increases the probability of writing full blocks of data, and thus reduces redundant operations.

The hash tree is equipped with a specially designed cache. It caches entire blocks of hashes in every level of the tree, while verifying the consistency of every cached block's parent. Recall that the consistency of each node in the tree depends on the matching of its value to the hash of its children's values. The integrity of each data block involves the consistency of the corresponding leaf and all of its ancestors. Since ancestors are common to many nodes, caching them prevents many redundant verifications.

The key regression mechanism keeps track of recently derived keys, and thus improves the performance for reading blocks of the same epoch. The user interaction component caches the common keys, so that each key is only calculated once. It also caches the entire user table. This makes sense because the table must be entirely read anyway in order to verify its integrity and authenticity.

6.3.2 Multi-threading

Another important feature of our file system implementation is allowing multiple processes to efficiently access numerous files at the same time. Handling multiple requests concurrently is achieved by multi-threaded programming of the CRUST client. CRUST handles each request by a separate thread. Multi-threading allows for better utilization of the system's resources, because a thread that is waiting for a file operation to complete can be switched with a thread having intensive computations. Furthermore, modern computer systems that have multiple CPUs, or CPUs with multiple cores, allow the threads truly concurrent execution.

The basis for multi-threading support is implemented by the FUSE library, which allows separate requests to run in separate threads. CRUST synchronizes the different threads by enforcing mutual exclusion on the data structures that are shared between them (using POSIX *pthread* mutexes). Coherency between threads accessing the same file is assured in CRUST by sharing all the in-memory data structures that are relevant for the file. By sharing the caches, threads accessing the same file gain a performance improvement, since redundant operations are omitted. Note that the sharing of caches is legitimate because the different application processes addressing the client represent the same CRUST user, thus they share the same privileges.

6.3.3 Meta-data organization

Further optimizations were achieved by improving the meta-data organization. Note that the epoch array and the hash tree leaves are always accessed correspondingly, because both are required for reading and writing any block of file data. Therefore we decided to store them interleaved, such that a single block of meta-data contains both epoch information and hash tree information relevant for several data blocks. This scheme improves the delay of our system because fewer underlying file operations are necessary in order to read a single byte. It may also improve the system's throughput due to the increase in space locality of the meta-data access sequences.

The meta-data file contains several dynamically sized structures, such as the array of lockboxes and the hash tree. Storing such structures adjacently in the same file introduces an organization problem: the location of each structure in the file must be managed. Moreover, changing the size of a structure requires a reorganization, which may involve moving a large amount of data from one position to another. When frequent changes occur, such as the case of constantly appending data at the end of a CRUST file, reorganization of meta-data can become very inefficient.

Our solution uses a small index of locations and sizes for organizing the various structures. This index is saved at the beginning of the meta-data file. To avoid frequent reorganizations, the structures are allocated with more space than immediately needed. A doubling technique multiplies the allocated space by two when additional space is needed and by half when only a quarter of the space is utilized. Note that the organization problem can also be solved by storing each structure in a separate meta-data file. However, this simplistic solution adds the overhead of maintaining multiple file handles, and does not scale for systems containing large numbers of files.

6.3.4 Optimized constants

The information in the data file is always read in chunks of a single block, because the whole block is necessary for decrypting and authenticating each byte in the block. A similar situation occurs for writing. Having a large block size increases the delay required for handling a single read or write request, whereas having a small block size increases the space overhead of our meta-data structures. CRUST uses a block size of 4 KB, to match the default block size in most Linux file systems; this way, we reach a reasonable balance between the delay and the space overhead of our system.

Similarly, the hash tree nodes (except the root) are also accessed in chunks, because verifying each internal node requires obtaining all of its children, which are stored adjacently on the disk. We defined the internal nodes' maximal number of children (the fan-out) so that each node's children fit in a 4 KB block. For instance, when SHA-1 is used for hashing and when the hash tree leaves are interleaved with the blocks' 32-bit epoch identifiers, their parents' fan-out is set to 170.

The number of nodes on the paths from the root to the leaves (the height) of the hash tree also affects performance, because an entire path is accessed in order to authenticate a single block of data. CRUST limits the height of the hash tree to three nodes, including the root and the leaves. Note that the height limitation is possible because the root's fan-out is unlimited. CRUST caches the root and all of its children. Having this information cached and verified in advance, only a single block of hashes needs to be read and hashed in order to verify any randomly read data block.

6.4 Coherency

Caching improves performance, but it comes at the price of coherency problems between clients accessing the same file simultaneously. For instance, when writing is performed, a reader may read stale data due to caching on his side or due to the writing being delayed by the writer. However, a coherency problem would occur in our case even without caching, because CRUST reading and writing are not atomic operations. Each of these operations requires performing several actions on the underlying file system, which may get mixed with other clients' actions.

To resolve such problems, CRUST locks the files while such operations are performed. The files are locked using the `fcntl` system call, which is supported by many file systems. File reading in CRUST involves acquiring a shared lock, whereas writing involves an exclusive lock. Shared locks can be acquired by an unlimited number of clients at the same time, but an exclusive lock cannot coexist with another lock of any kind. Thus, a reading client waits for writers to finish their tasks, whereas a writing client waits for every client currently accessing the file.

Locking and unlocking files for every read and write operation can be very inefficient, especially when the files are stored on a remote server. Thus, CRUST acquires locks when the files are opened and releases the locks only when the access is finished. The file locking procedure is hidden from the CRUST users; an operation issued by the user completes after the necessary waiting is done. The use of file locking assures coherency as long as the system is legitimately used; however, it is impossible to guarantee coherency when various kinds of attacks occur.

7 Performance evaluation

We performed a series of tests in order to compare the performance of CRUST layered over NFS to native NFS. Some of the tests also evaluated a simple FUSE pass-through file system client. The pass-through file system was used for estimating how much of the overhead of CRUST is due to implementing it in user-level.

Our experimental setup included a server machine and a client machine. The NFS server (using NFS protocol version 3) was run on a 1.8 GHz Pentium 4-M machine with 256 MB memory. The clients were run on a 2.4 GHz Pentium 4 machine with 512 MB memory. The operating systems of the server and the client were Fedora Core 4 Linux (version 2.6.11) and Fedora Core 6 (version 2.6.18), respectively. The machines were connected by a 100 Mbps Ethernet link. The CRUST code was written in C++ and includes about 5,000 lines of code.

7.1 Micro-benchmarks

The micro-benchmarks are primarily designed to compare between CRUST and NFS. They also allow us to compare the performance of CRUST with the extrapolated performance of two related works—SiRiUS [12] and Plutus [16]. Our benchmarks include similar operations to those tested in the SiRiUS and Plutus micro-benchmarks. Our tests include creating files, deleting files and sequentially reading and writing files of different sizes. Each test was executed a hundred times on different files. The average results are shown in Fig. 7.

The figure demonstrates a considerable improvement in CRUST. File creation in CRUST is slower than in NFS, because it involves key generation and initialization of both the data file and the meta-data file. However, it is particularly efficient relative to public-key systems as a result of avoiding the lengthy process of RSA key generation. The overhead for accessing small files in CRUST mostly consists of meta-data access times and context switches. The cryptographic overhead is small due to avoiding public-key operations. Optimizations such as improved meta-data organization for small files also contribute to the performance measured in these tests.

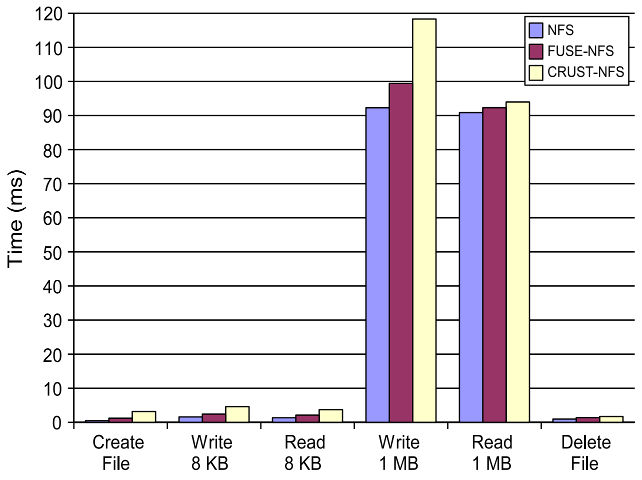

Fig. 7 Micro-benchmark results. The NFS columns show the times for a standard NFS system. FUSE-NFS means a pass-through FUSE over NFS, and CRUST-NFS shows our system's times

File deletion in CRUST is slightly slower than in NFS because it performs deletion of two files, compared to just one when using NFS. The figure also shows that the overhead of the pass-through FUSE file system in all tested operations is small relative to our system's overhead. This finding implies that the user-level implementation of CRUST has little effect on the results.

We also wanted to compare the performance of CRUST to that of SiRiUS and Plutus. Since SiRiUS was originally evaluated in comparison to NFS, it was straightforward to compare its overhead with the overhead of CRUST over NFS. We also compare the overhead of Plutus with the overhead of CRUST over NFS, despite the fact that the Plutus prototype was built over OpenAFS instead of over NFS. We argue that the comparison is meaningful because CRUST does not take advantage of any particular feature of NFS that would achieve improved performance.

The average time overhead of each operation in each system, relative to the time it takes on its underlying file system, is summarized in Table 1. The data for SiRiUS and Plutus is taken from the original authors' reports. We show the overhead in percent over the underlying file system to compensate for the different computing environments used for each of the three systems. The table shows that the CRUST overhead is always lower than both SiRiUS and Plutus—sometimes dramatically so.

Table 1 Relative overhead comparison between CRUST, SiRiUS, and Plutus

Operation	CRUST over NFS (%)	SiRiUS over NFS (%)	Plutus over OpenAFS (%)
Creating a file	550	3,500	Unspecified[a]
Reading a 8 KB file	180	1,900	Unspecified[a]
Writing a 8 KB file	190	1,900	Unspecified[a]
Reading a 1 MB file	3	130	44[b]
Writing a 1 MB file	28	530[c]	44[b]
Deleting a file	80	270	Unspecified[a]

[a] The authors of Plutus did not include file creation and deletion tests, or tests on small files
[b] Plutus incurs an overhead of about 3 s in comparison to its underlying file system for reading and writing a 40 MB file. As an estimation, we divide this time by 40 to match the 1 MB file we used, and assume that our processors are about twice as fast. This optimistic estimation concludes that Plutus would have achieved a time overhead of about 40 ms, or 44%, if it was run in our experimental setup
[c] This specific slowdown is mostly affected by technical limitations of the protocol (NFSv3) as used in the SiRiUS file system interface

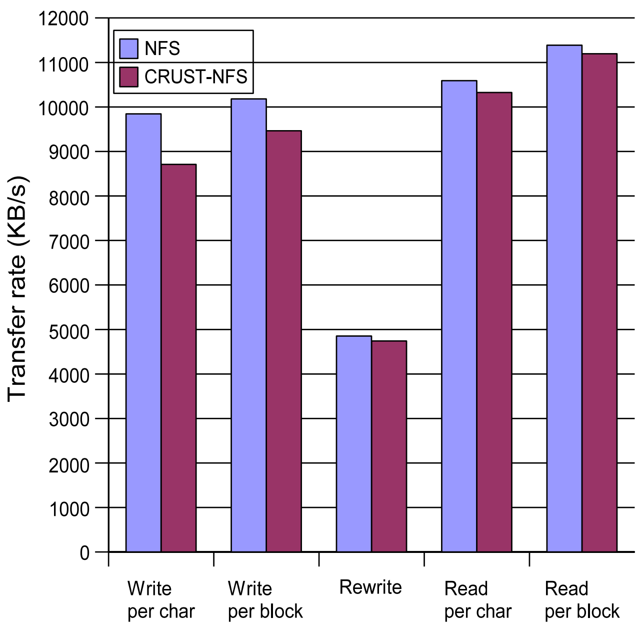

Fig. 8 Bonnie benchmark results

7.2 The Bonnie benchmark

Bonnie [7] is a well-known file system benchmark. It performs a series of tests on a single large file. The tests are:

- *Sequential output.* The file is created and written one character at a time. Then, it is written again block by block. Lastly, each block is read, altered and rewritten.
- *Sequential input.* The file is read one character at a time. Then, it is read again block by block.
- *Random seeks.* Several processes concurrently seek to random locations in the file and read one block at a time. The block is modified and rewritten in 10% of the cases.

We executed Bonnie with a file size of 1 GB, which is twice the amount of memory on the client machine. This reduces the amount of read requests that are trivially resolved using the operating system's page cache. Bonnie measures the throughput of the operations, as well as the percentage of CPU usage. The CPU usage results are ignored in our evaluation because they do not include the overhead of the CRUST client process.

The sequential input and output results are shown in Fig. 8. For each test, we compare NFS and NFS via CRUST. The performance is measured in transfer rate, as reported by Bonnie.

CRUST performed block reads with only a 2% time overhead, as compared to NFS. Block writes were performed with an 8% overhead. These results are indeed better than those presented in the micro-benchmarks, because the larger file size increases the effect of caching and optimizations. Character reads and writes in CRUST were performed 3% and 13% slower, respectively. Rewriting was performed by CRUST with a 2% time overhead.

The random seek results of the Bonnie benchmark are presented in the following table:

System	Operations per second
NFS	93.4
CRUST-NFS	22.8

CRUST performed random seeks with a 4.1 times slowdown. Random seeks were expected to have worse performance than sequential access because they eliminate the effect of most caches used by our system. Moreover, random seeks in a large CRUST file translate to random seeks in both the data file and the meta-data file, accumulating overhead from both seeks. Therefore, the above result is satisfying, and proves that the random access optimizations included in CRUST pay off.

7.3 Privilege modifications

In this section, we present a performance evaluation of the operations that modify file access privileges in CRUST. Since our per user privileges are more expressive than standard UNIX file permissions, a concrete comparison with NFS is not possible for these operations. Instead, we compare the performance of permission modifications relative to other CRUST operations.

Write permissions are expected to be slightly more expensive to change than read permissions, because changing them involves additional actions. Specifically, a new master MAC key is generated only when a writer is revoked. This change requires both modifying all the lockboxes and recalculating the hash tree MACs, and thus may incur a longer delay. Therefore, we focus on evaluating modifications of only write privileges.

Our test creates a 1 MB file, grants 1,000 users write permissions to the file and then revokes each one of them. The time for granting permissions to each user is measured and the times are averaged. Revocation time is measured in the same way. The results are presented in the following table:

Operation	Time (ms)
Permission granting	31.5
Revocation	32.3

The results show that both modifications can be performed more than 30 times per second. Moreover, their performance is comparable to the time required to read a few hundreds of kilobytes from an ordinary CRUST file. Thus, the results are acceptable.

Granting access permissions to a user requires CRUST to calculate and add a new record to the meta-data file, recompute meta-data MACs and rarely reorganize the meta-data file. Revocations, on the other hand, involve several additional actions, such as changing keys and performing key regression operations. Thus, we indeed expected that revocations would require more time than permission granting. The fact that the time difference between the two operations is only 3% confirms that the key regression mechanism is efficient and valuable for the system.

7.4 Lazy revocation overhead

Lazy revocation using key regression has the benefit of making revocation operations extremely efficient. However, it comes at the expense of possibly harming a file's access performance following a large number of revocations. This phenomenon may occur when each block in the file is modified in a different epoch, and the epoch variance throughout the file is large. Calculating the encryption keys for each block would take a longer time for such a file than for a file entirely written in a single epoch, where the keys are equal for each block.

In this section, we compare the access performance of a revocation-free file with a revocation-rich file. Our test begins by creating the two files, each filled with 100 MB of data. The first file is created normally, without special permission modifications. The other file is manipulated as follows: a specific user is alternately granted write permissions and revoked from the file; this process is repeated a million times, while writing data to a random block in the file following each revocation. The random blocks are chosen in advance such that the resulting revocation-rich file has blocks of random epochs picked uniformly from a wide range of a million epochs. After initialization, we compared the two files for performance of sequential reading and overwriting. The results are shown in Fig. 9.

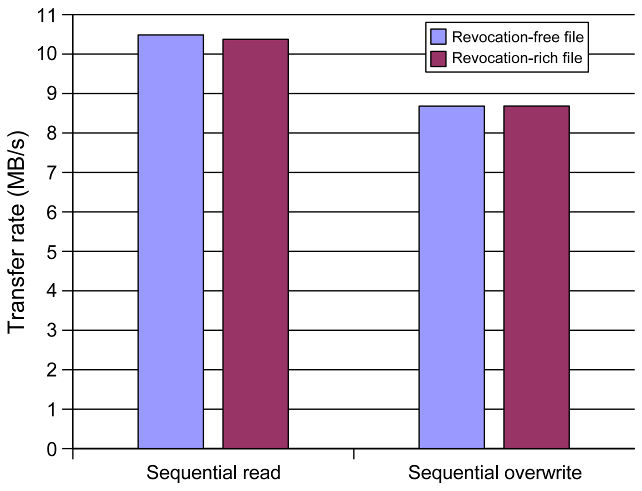

Fig. 9 Performance of CRUST for sequentially accessing revocation-free and revocation-rich 100 MB files

The revocation-rich file had only a 1% slowdown in read performance, as compared to the revocation-free file. The performance difference for overwriting is negligible—less than 0.1%. This makes sense because overwriting entire blocks does not involve decrypting the original blocks, but instead encrypts the new data with the current key. As we expected, the variance of the blocks' epochs had a negligible effect on the performance of accessing the file. This benchmark further demonstrates that lazy revocation using key regression is a practical solution, even in extreme conditions.

7.5 Cryptographic cost

In this section we analyze the amount of cryptographic operations involved in the various CRUST procedures. Since most file operations may involve a key regression computation, they include up to 105 HMAC calculations as indicated in Sect. 4.2.3. Any file data access requires traversing the hash tree, involving at least 3 hash operations. The worst-case limits on the amount of cryptographic operations involved in CRUST procedures, where r and w denote the number of readers and writers, respectively, are summarized in the following table:

Procedure	Hash or HMAC operations	4 KB encryption/ decryption operations
File creation	109	1
4 KB file reading	114	2
4 KB file overwriting	$117 + 2r$	3
Permission granting	$5 + 4\,(r + w)$	2
Revocation	$109 + 7r + 5w$	$2 + r + w$

8 Extensions

In this section we present several extensions for the basic CRUST functionality that we did not yet implement.

8.1 Supplemental confidentiality

Confidentiality of file names and access lists is not provided by the basic CRUST design. These properties can be achieved using methods similar to those included in Plutus [16]. File name confidentiality can be achieved by encrypting each file's name with a different random key that is generated by its owner. To allow file name decryption by the file's users, the key should be included in their encrypted lockboxes that are stored in the meta-data file.

An access list can also be encrypted with an owner-generated key, provided that the key is included in the lockboxes and that an additional method is used to distinguish the lockboxes, so that each file user can detect his own lockbox. A possible method is to include the target user's ID inside his encrypted lockbox. In this method, a user can identify his lockbox by decrypting its contents (using his common secret key with the owner) and verifying that the decrypted user ID matches his own ID.

8.2 Freshness

Recall that an adversary with full control of the server can mount rollback attacks [24]. The following solution for meta-data freshness was proposed in SiRiUS [12], and can be adopted in CRUST. The owner's file hierarchy is periodically timestamped and the files' meta-data is signed in a tree construction. This solution can be converted to use symmetric-key operations by replacing the public-key signature of the tree's root with a MAC for each user granted access to any file in the hierarchy.

Note that the above solution does not provide freshness guarantees for the file data, since the file writers need the ability to sign the file contents without the owner's intervention. Solutions for data freshness are left for future work.

8.3 File links

UNIX file systems usually implement a symbolic link as an ordinary file containing a textual reference to the target of the link, and an indicator marking it as a symbolic link. CRUST can implement symbolic links cryptographically, by having a CRUST file whose data contains the textual reference. The file is encrypted and involves access control, as ordinary files in our system. An indicator for symbolic links should be added to the meta-data file, unless support for symbolic links exists in the underlying file system. In any case, the indicator must be specifically verified for integrity and authenticity, as any other part of the meta-data, by including the indicator's value in the existing meta-data MACs.

Hard links allow a file to be referenced by multiple names. The basic CRUST design does not support hard links since only one file name is signed in the meta-data file.

8.4 Crash recovery

Since CRUST operations are not atomic, a system crash on the client or server side may leave a file in an unreadable state. Moreover, the write caching mechanism increases the probability for the loss of some written data following a crash. Well-known crash recovery solutions that conform to our security model usually maintain non-volatile logs of the actions waiting to be carried out, which can be resumed following a crash. Since the actions performed on the underlying file system are not secret in our design, the logs can be saved in an insecure location, and they may be operated with only a little overhead to the overall performance of the system. Implementation of such mechanisms in CRUST is left for future work.

9 Conclusions

We introduced a new file system layer that allows secure file sharing over untrusted storage systems. Our solution includes: in-band key distribution; flexible control on file access privileges; cryptographic access control, and strong security while assuming that the file server is untrusted and unmodifiable. We achieve all these results without any public-key cryptography. Our approach is especially useful in situations where the users have no control over the underlying file system. It is also useful for sharing files between users that are rarely online, because direct communication between the users is not necessary. An additional insight of our approach is that the servers are not required to carry out cryptographic operations and thus the server scalability and the overall performance may be superior.

A significant part of our work focused on performance and usability issues of the system. We have implemented our designed system as a Linux stackable file system and shown that it has very modest overhead, despite the strong security that it provides. We conclude that our approach is convenient to use, performs well in high workloads and supports any underlying file system.

A Protocols

A.1 Creating a file

CRUST creates a file anywhere in the user's directory hierarchy by taking the following steps.

1. Generate a random master key regression state for the file, as well as a file master MAC key.
2. Create the meta-data file and initialize the file's access lists.
3. Set the current epoch to zero.
4. Derive the current key regression state and encryption key for the current epoch from the master key regression state.
5. Save the owner's lockbox, encrypted with his encryption key, $K_i^{\text{Self enc}}$, and a MAC of the owner's meta-data using his MAC key, $K_i^{\text{Self MAC}}$.
6. Create the data file and encrypt the file data using the derived encryption key from step 4. Update the hash tree on-the-fly, while encrypting. Set the block epochs to zero.
7. MAC the hash tree root using the file's master MAC key (from step 1).

A.2 Sharing a file

CRUST takes the following steps when a file owner, Alice, wishes to share the file with another user, Bob.

1. Alice reads her lockbox, verifies it using her MAC key, $K_{\text{Alice}}^{\text{Self MAC}}$, and decrypts it using her encryption key, $K_{\text{Alice}}^{\text{Self enc}}$. The hash tree root is verified using the file's master MAC key obtained from the lockbox.
2. Alice obtains Bob's user ID. If the cached version of the user table is not sufficient, a fresh version of the table is read and verified using Alice's user table MAC key, $K_{\text{Alice}}^{\text{User table MAC}}$.
3. Alice adds Bob's ID to the file's access list (as a reader or a writer) and re-signs the list.
4. Alice derives her common encryption and MAC keys for communicating with Bob, using the Leighton–Micali scheme.
5. Alice prepares Bob's encrypted and authenticated lockbox. It contains the current key regression state and Bob's file reader MAC key, $K_{\text{Bob}}^{\text{File MAC}}$, if Bob is granted read-only access. If Bob is given write permission, the file's master MAC key is given instead of the reader MAC key.
6. Alice calculates a MAC of the meta-data for Bob using her common MAC key with him. If Bob is granted read-only access, Alice also calculates a MAC of the hash tree root using $K_{\text{Bob}}^{\text{File MAC}}$.

A.3 Writing to a file

The following procedure is performed by CRUST when Bob writes to a file owned by Alice (it is located under Alice's directory). Note that a similar procedure is carried out if the file is owned by Bob instead.

1. Bob obtains Alice's user ID. If the cached version of the user table is not sufficient, a fresh version of the table is read and verified using Bob's user table MAC key. Bob also obtains his common encryption and MAC keys with Alice according to the Leighton–Micali scheme.
2. Bob locates and reads his writer lockbox. The lockbox is decrypted and verified using his common encryption and MAC keys with Alice. The hash tree root is also verified using the file's master MAC key.
3. Bob derives the current encryption key from the current key regression state.
4. For every data block to be written:
   - (a) If the data block is only partially updated, the stored block is firstly read:
     - i. The encrypted block and its epoch are read and verified using the hash tree.
     - ii. The block's encryption key is derived from the current key regression state, based on the block's epoch identifier.
     - iii. The decrypted block is updated with the written data.
   - (b) The block is encrypted with the current encryption key.
   - (c) The block's epoch is updated to the current epoch.
   - (d) The hash tree is updated.
5. Bob updates the MACs of the hash tree root with the file master MAC key and with each reader's MAC key.

A.4 Reading a file

The following procedure is performed by CRUST when Bob reads a file owned by Alice.

1. This step is identical to the first step for file writing.

2. Bob locates and reads his lockbox. The lockbox is decrypted and verified using his common encryption and MAC keys with Alice. The hash tree root is also verified using either the file's master MAC key or Bob's reader MAC key, according to Bob's permissions.
3. For every data block to be read, the encrypted block and its epoch are read and verified using the hash tree. Each block is decrypted using an encryption key derived from the current key regression state, based on the block's epoch identifier.

A.5 Revoking a file sharing

CRUST takes the following steps when Alice wishes to revoke the sharing of a file owned by her with another user, Bob.

1. This step is identical to the first step for file sharing.
2. Alice obtains Bob's user ID.
3. Alice removes Bob's ID from the file's relevant access list and re-signs it.
4. Alice increases the current epoch and derives the new key regression state. If Bob had write permissions, a new random file master MAC key is generated. All the lockboxes are updated with these modifications, and re-signed.
5. If a new file master MAC key was generated, the MACs of the hash tree root are recalculated.

References

1. Adya, A., Bolosky, W.J., Castro, M., Cermak, G., Chaiken, R., Douceur, J.R., Howell, J., Lorch, J.R., Theimer M., Wattenhofer R.: FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In: Proceedings of OSDI (2002)
2. Backes, M., Cachin, C., Oprea, A.: Secure key-updating for lazy revocation. In: Proceedings of ESORICS. Lecture Notes in Computer Science, vol. 4189, pp. 327–346. Springer, Berlin (2006)
3. Bellare, M., Canetti, R., Krawczyk, H.: HMAC: Keyed-hashing for message authentication. RFC 2104, February (1997)
4. Black, J., Urtubia, H.: Side-channel attacks on symmetric encryption schemes: the case for authenticated encryption. In: Proceedings of the 11th USENIX Security Symposium, pp. 327–338. USENIX Association, Berkeley (2002)
5. Blaze, M.: A cryptographic file system for Unix. In: Proceedings of the ACM Conference on Computer and Communications Security, pp. 9–16 (1993)
6. Boneh, D., Franklin, M.K.: Identity-based encryption from the Weil pairing. In: CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology, pp. 213–229. Springer, London (2001)
7. Bray, T.: The Bonnie home page. Located at http://www.textuality.com/bonnie (1996)
8. Cattaneo, G., Catuogno, L., Del Sorbo, A., Persiano, P.: The design and implementation of a transparent cryptographic file system for Unix. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, pp. 199–212. USENIX Association, Berkeley (2001)
9. Dworkin, M.: Recommendation for block cipher modes of operation. Special Publication 800-38A, NIST (2001)
10. Fu, K., Kamara, S., Kohno, T.: Key regression: Enabling efficient key distribution for secure distributed storage. In: Proceedings of NDSS (2006)
11. Gobioff, H., Gibson, G., Tygar, D.: Security for network attached storage devices. Technical Report CMU-CS-97-185, Carnegie Mellon University, October (1997)
12. Goh, E.-J., Shacham, H., Modadugu, N., Boneh, D.: SiRiUS: Securing remote untrusted storage. In: Proceedings of NDSS. The Internet Society, Geneva (2003)
13. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM **33**, 792–807 (1986)
14. Heidemann, J.S., Popek, G.J.: File-system development with stackable layers. ACM Trans. Comput. Syst. **12**(1), 58–89 (1994)
15. Jakobsson, M.: Fractal hash sequence representation and traversal. In: IEEE International Symposium on Information Theory (2002)
16. Kallahalla, M., Riedel, E., Swaminathan, R., Wang, Q., Fu, K.: Plutus: scalable secure file sharing on untrusted storage. In: Proceedings of FAST. USENIX, Berkeley (2003)
17. Kher, V., Kim, Y.: Securing distributed storage: challenges, techniques, and systems. In: StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability, pp. 9–25. ACM Press, New York (2005)
18. Kher, V., Seppanen, E., Leach, C., Kim, Y.: SGFS: Secure, efficient and policy-based global file sharing. In: Proceedings of the 23rd IEEE/14th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2006) (2006)
19. Kleiman, S.R.: Vnodes: an architecture for multiple file system types in Sun UNIX. In: Proceedings of the USENIX summer conference, pp. 238–247 (1986)
20. Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S.E., Eaton, P.R., Geels, D., Gummadi, R., Rhea, S.C., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B.Y.: OceanStore: an architecture for global-scale persistent storage. In: Proceedings of ASPLOS, pp. 190–201 (2000)
21. Leighton, F.T., Micali, S.: Secret-key agreement without public-key cryptography. In: Proceedings of CRYPTO. Lecture Notes in Computer Science, vol. 773, pp. 456–479. Springer, Heidelberg (1993)
22. Li, J., Krohn, M.N., Mazières, D., Shasha, D.: Secure untrusted data repository (SUNDR). In: Proceedings of OSDI, pp. 121–136 (2004)
23. Mazières, D., Kaminsky, M., Kaashoek, M.F., Witchel, E.: Separating key management from file system security. In: Proceedings of the 17th ACM Symposium on Operating System Principles, pp. 124–139 (1999)
24. Mazières, D., Shasha, D.: Don't trust your file server. In: Proceedings of HotOS, pp. 113–118. IEEE Computer Society, Washington, DC (2001)
25. Miller, E.L., Long, D.D.E., Freeman, W.E., Reed, B.: Strong security for network-attached storage. In: Proceedings of FAST, pp. 1–13, (2002)
26. Naor, D., Shenhav, A., Wool, A.: Toward securing untrusted storage without public-key operations. In: StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability, pp. 51–56. ACM Press, New York (2005)
27. NIST: Advanced encryption standard. Federal Information Processing Standards, FIPS PUB 197 (2001)
28. NIST: Secure hash standard. Federal Information Processing Standards, FIPS PUB 180-2 (2004)
29. Riedel, E., Kallahalla, M., Swaminathan, R.: A framework for evaluating storage system security. In: Proceedings of FAST, pp. 15–30 (2002)
30. Rubin, A.D.: Kerberos versus the Leighton-Micali protocol. Dr. Dobb's J. Softw. Tools **25**(11), 21–22 (2000)

31. Stanton, P.: Securing data in storage: a review of current research. *CoRR*, cs.OS/0409034 (2004)
32. Steiner, J.G., Neuman, B.C., Schiller, J.I.: Kerberos: an authentication service for open network systems. In: Proceedings of the USENIX Winter Conference, pp. 191–202 (1988)
33. Szeredi, M.: Filesystem in userspace. Located at http://fuse.sourceforge.net
34. The OpenSSL project. Located at http://www.openssl.org
35. Wright, C.P., Martino, M.C., Zadok, E.: NCryptfs: A secure and convenient cryptographic file system. In: Proceedings of the Annual USENIX Technical Conference, pp. 197–210 (2003)
36. Zadok, E., Badulescu, I., Shender, A.: Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98. Computer Science Department, Columbia University (1998)