CryptoShare: Secure Verifiable Shared Remote File System

Adel Hanna Nikolaos Michalakis Todd Nightingale
hanna@lcs.mit.edu nikos@lcs.mit.edu tnight@lcs.mit.edu

***Abstract*—There are several authentication techniques that allow clients to trust the identity of remote file servers. However, remote servers cannot always be trusted for preserving the integrity and privacy of the stored data [9]. In this paper we present the design and implementation of *CryptoShare*, a file system that allows clients to store data in remote file servers in a way that protects privacy and resists tampering by a malicious file server. Our system supports multiple client reads and writes. Data secrecy is obtained by sending only encrypted files to the server. To avoid data tampering we keep on the server a meta-file for each file, that contains hashes for each file data block. The hashes are verified using hashing trees, so the client needs only to store a single hash to detect tampering of any file in the tree. Encryption, decryption and verification are transparent to the file system user. In addition to single users, we provide the same protection to groups that have access to shared files. To detect tampering in group files, we use a Trusted Administrative Server to store a single hash per tree for data verification. We compare our design with respect to previous ones that have addressed the same problem, and evaluate it in terms of its security, flexibility and performance.**

I. Introduction

The Self-certifying File System, SFS ([7]), was designed to span the Internet such that users can access files from anywhere and share files with anyone. A user is willing to use a remote file system that is owned by someone different than herself only if her files are protected as though they were locally stored.

A first step in providing file protection is to authenticate the remote file server, so that the user can be sure that is using a trusted remote file server. SFS currently provides a solution to this problem, using Self-certifying path names.

We believe that some users would benefit from an extra level of protection, when they decide to store their files on remote file servers. Namely, they would not like their data to be exposed or be tampered with, if a remote file server proves to be malicious. Also server owners could benefit by renting their disk space to users, because users will be more willing for such a service, when they can have control over checking the authenticity of their remote data.

Currently, SFS does not address the issue of data exposure and tampering by remote file servers that are not trustworthy. In this paper, we discuss the design and implementation of *CryptoShare*, a system that provides data secrecy and tamper resistance in SFS that is transparent to the user. We have designed our mechanisms to support multiple users and provide both single user and group read/write access. Although, our implementation is currently built on top of SFS, our design could be implemented on top of network file systems other than SFS.

In Section II we present previous work that is relevant to our system. In Section III we present our design goals. In Section IV we describe our data structures and mechanisms for providing data secrecy and tamper-resistance for single users and groups. In Section V we discuss our implementation built on top of SFS. In Section VI we analyze the security of the system, and in Section VII we evaluate our design with respect to its flexibility and performance.

II. Related Work

There is a lot of work done on cryptographic file systems that allow files to be stored remotely using encryption/decryption techniques transparently to the user. However, not a single system alone satisfies all of our requirements. For that reason, our design uses pieces contributed by each different system and combines them to enhance SFS. Here is a discussion of the inefficiencies encountered in previous work:

A. CFS

The Cryptographic File System ([3]) guarantees the privacy of data stored to remotely by encrypting files before sending them to the server and decrypting them after reading from the server. However, there is no group sharing in CFS, and there is no data verification support. Furthermore, the encryption and decryption of files is not transparent to the user.

B. CryptFS

CryptFS ([13]) provides better security and performance than CFS, but like CFS it does not provide guarantees on data integrity. Therefore, a compromised server could tamper with user data.

C. TCFS

TCFS ([4]) provides the same functionality as CFS and CryptFS, but the encryption and decryption of files occurs transparently to the user. It provides a mechanism that uses a Threshold Secret Sharing scheme for a group of users to share files. That means that two or more members of a group have to be on-line in order for the decryption key to be successfully reconstructed in order to read a shared file. TCFS has a mechanism to provide protection from tampering with the files. However, since all authenticators about the files are stored at the remote server, the server can still return stale data to the client and not be detected.

D. TDB

In [5] a Trusted Database system is build using untrusted storage. To ensure data secrecy and resistance to tampering the system relies on a small secret store and a small tamper-resistant store. Secrecy is obtained similarly to the above systems by file encryption and decryption at the client. Resistance to tampering is obtained by saving a hash of the data in the large untrusted store. Although TDB is designed for databases its design could easily be useful for a remote file system, as well. However, TDB does not address the problem of protecting shared data from tampering.

E. SUNDR

The Secure Untrusted Data Repository ([9], [8]) is the design that comes closest to ours. SUNDR provides data secrecy and tamper-resistance both to single users and groups. To provide tamper-resistance for group files it provides a consistency protocol, where the server maintains a version number for each group member. Every member signs not only her own version number, but also provides an update certificate that modifies the version numbers of other users. This protocol guarantees fork-consistency, such that if a server tampers with data then the clients are partitioned into groups, where each group cannot see operations performed by other groups. However, SUNDR still suffers from *forking attacks*, where a server can create multiple copies of the file system in parallel and serve a different one for each user. It relies on off-line communication among the clients to detect such attacks. Also, the server is modified in order to run the consistency protocol. In our case, an administrator client is used to notify the group when data tampering occurs. Group members have access to the administrator client and update its state. That way we don't need to modify the server to achieve tamper-resistance. This makes the system easier to use, since a group of users can simply rent storage service from any remote file server.

III. Design

In our system we have a set of trusted components and a set of untrusted ones.

We assume that the client hardware and software are trusted and under full control of the user.

On the other hand, network traffic is considered untrusted. More specifically, an eavesdropper could observe unencrypted data on the network. The remote server is also considered untrusted and adversarial. It could read, modify or erase stored data. Also, it could provide false service and when asked to return a file, return instead another file or an older copy of the same file.

SFS protects against an adversary eavesdropping on the untrusted network by encrypting data sent by the client and the server.

Therefore, our design needs to address the following issues:

- The contents of files stored at the remote file server must be encrypted at all times. The server should not be able with the information it knows to decrypt and see the contents of files.
- The client must be able to detect if a file or directory that is returned by the server has been modified illegally. That includes the case where the server provides stale data back to the client. Data that are modified illegally are assumed to be lost. We do not require that data that the server has tampered with can be recovered.
- The mechanism for providing data secrecy and resistance to tampering must be transparent to the user. That means that the user never has to encrypt or decrypt files.
- Users must see the same semantics as in the original file system. Random access reads and writes must still work.
- Administrators must be able to backup files as before, so the filenames and the directory contents must not be encrypted, nor must the directory hierarchy be hidden. The file server must see no difference between encrypted files and unencrypted files. It can see the contents of directories and the filenames as in regular files. However, if it reads the contents of a file, it reads blocks that it cannot decipher and read the corresponding clear-text.
- A group of users should be able to share files under the same file system. Users are able to encrypt and decrypt group files independently of each other. In addition, each group member can differentiate between a legal modification from a group member on a certain file or directory and an illegal modification by the file server.
- Our system must not require a modified file server to provide tamper-resistance and secrecy. All these operations are handled by the clients. The clients should be able to use unmodified file servers to store their data. This is an important property especially in a file system like SFS that scales the Internet. File servers should not be expected to be modified for a special class of users that needs data secrecy and tamper-resistance.

IV. Architecture

CryptoShare is built as an additional layer on top of SFS to provide additional features for data secrecy and tamper-resistance. Figure 1 shows the CryptoShare layered architecture.

CryptoShare provides *secure* and *verifiable* remote read and write permissions to files and directories. By *secure* we mean that remote data are encrypted and only privileged users that are explicitly named can read or write them. Furthermore, when writing the contents are never seen in the clear. By *verifiable* we mean that, when reading files, privileged users can verify that their

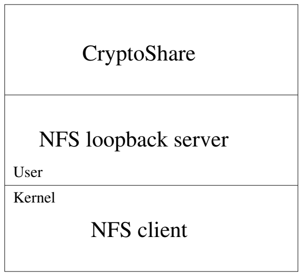

Fig. 1. CryptoShare layered architecture.

contents are correct (i.e. have not been modified without their knowledge). When writing, privileged users can produce evidence that their writes were legal. We will distinguish from now on files and directories that are secure and verifiable from regular ones by calling them CryptoShare files and directories.

Table I presents the file permission semantics that CryptoShare can support. The execute bit is omitted, since we are not concerned with remotely executable files.

Owner	Group	World
rw	-	-
rw	rw	-
drw	d-	d-
drw	dwr	d-

TABLE I
FILE SEMANTICS OFFERED BY CRYPTOSHARE.

Our system does not support world readable and writable files. Both secrecy and data verification have no meaning if anyone can have access to such files remotely. Therefore, our architecture focuses on providing secrecy and tamper-resistance to single users and groups. Users can use CryptoShare without Encryption/Decryption or Authentication (which reduces to the unmodified SFS) for world readable/writable files. In Section VI we discuss in more detail the implications of providing additional permissions.

A. Keys

As in TCFS ([4]), users have a pair of public-private keys ($Client.PK$, $Client.SK$) to perform operations on the remote file system, but the files are encrypted using symmetric keys. This adds flexibility and improves performance. A user can encrypt a whole file using the same symmetric key K_f or encrypt each block with a different key for additional security. Performance increases since symmetric key operations are about three orders of magnitude faster than public key operations.

B. Meta files

All the data necessary to encrypt/decrypt file blocks and verify the contents of a remote file are stored in a "sibling" file, called the *meta-file*. The meta-file is located at the remote server. It contains the following:

- The name of the regular file.
- The set of symmetric keys (K_f) that are used to encrypt/decrypt file blocks. To protect each K_f, the owner of the file encrypts them with its public key ($Client.PK$). Only the user that has the corresponding private key can obtain the symmetric encryption key. If more than one users have access to a file, a copy of K_f is encrypted with each user's $Client.PK$. Only users that can read K_f can add valid user entries in the meta-file.
- A hash H_b for each encrypted block b of the file, that serves as an authenticator for that block. H_b is $H(E\{b\}_{K_f}$, where H is a weakly collision resistant hash function (such as MD5 or SHA-1) to prevent the server from retrieving the contents of the block nor replace the contents of the block without changing the value of the hash. The weakly collision property guarantees that a malicious server will not be able to find another $E\{b'\}$ that hashes to the same result.

Table II illustrates the contents of a meta-file for a particular remote file.

Each directory also has an associated meta-file. A directory meta-file contains the following:

- The name of the directory.
- Hashes $H\{m\}$ of the meta-files of all the first level files and directories inside the directory.

Table III illustrates the contents of a meta-file for a directory on the remote file system.

C. Single User Encryption Scheme

A file is encrypted on a per block basis. Blocks have size 512 bytes. That way, random access to files is allowed without the cost of decrypting the whole file. In the worst case we will need to decrypt one extra block than the regular read, if the data requested lie in two different blocks. When the file is created, the client generates an encryption key K_f for the whole file or one key per block. The client creates the corresponding meta-file and an encrypted version of K_f using $Client.PK$ is written to the meta-file. To decrypt blocks from a file the client obtains the encrypted K_f from the meta-file, decrypts K_f using $Client.SK$ and then decrypts the appropriate blocks using K_f. For single encryption key per block the same procedure is performed per block instead per file.

The name of the file (or directory) is not encrypted. The contents of a directory are not encrypted either. So, a regular `ls` command on the remote UNIX file server inside a CryptoShare directory will return the expected unmodified results. More importantly, having unencrypted file names facilitates administration of the remote file server and file sharing.

$.meta\_filename$			
N=$filename$			
C=$Client_1$	$Client_1.PK$	$E\{K_f\}_{Client_1.PK}$	
C=$Client_2$	$Client_2.PK$	$E\{K_f\}_{Client_2.PK}$	
...	...	...	
C=$Client_N$	$Client_N.PK$	$E\{K_f\}_{Client_N.PK}$	
$H\{b_1\}$	$H\{b_2\}$	...	$H\{b_n\}$

TABLE II

CONTENTS OF A META-FILE FOR A REMOTE FILE.

$.meta$	
N=$filename$	
F=$file_1$	$H\{.meta\_file_1\}$
F=$file_2$	$H\{.meta\_file_2\}$
...	...
F=$file_N$	$H\{.meta\_file_N\}$
...	...
F=dir_1	$H\{.meta\}$
...	...
F=dir_M	$H\{.meta\}$

TABLE III

CONTENTS OF A META-FILE FOR A REMOTE DIRECTORY.

D. Group Encryption

As in the single user encryption scheme, every group member has its own public-private key pair and each client uses an encryption key for each file. When a group member creates a file it uses the same procedure as in the single user case. Subsequent accesses to that file by other group members will use the same encryption key picked by the creator. In order for the creator to allow the other members to be able to decrypt the file, the creator stores a copy of K_f encrypted with each member's public key in the meta-file. In other words, it stores $E(K_f)_{Client_1.PK}$,$E(K_f)_{Client_2.PK}$, ..., $E(K_f)_{Client_N.PK}$.

E. Single User Data Verification Scheme

Each client needs to store a small amount of data to be able to verify that the server has not tampered with its data. A user verifies both the contents of files and directories.

Clients store hashes of all the *CryptoShare roots* that they have permissions to on the remote file server. We define *CryptoShare root hashes* as hashes of the directories or files in the remote file system which have no parent directory or a non CryptoShare parent directory.[1]

Roots are specific for each client. Figure 2 gives an example of the roots for a certain client.

CryptoShare maintains the invariant that the client keeps the latest root hashes locally at all times. As explained in Section VI this prevents stale data attacks by a malicious server.

[1] In the case of files, parent directory means enclosing directory.

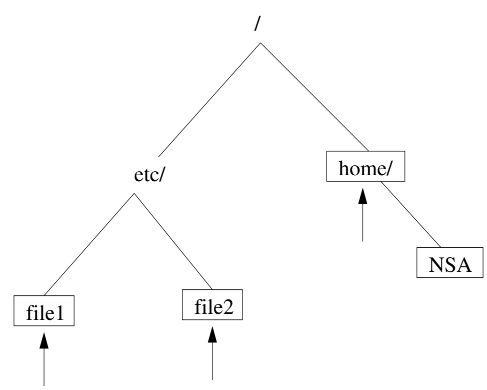

Fig. 2. An example illustrating roots. Boxes indicate CryptoShare files and directories. The arrows point to the roots.

The root hash is computed using hash trees introduced by [10], also known as Merkle trees. Root hashes are updated by the client when writing new data to a remote file or directory.

E.1 Changing Data

When writing a directory, the client performs the following:

If creating a file or a directory:

- It creates the corresponding meta-file and computes its hash.
- It updates the hash of the directory, where the new file or directory is located.
- It updates all the parent directories up to the root.
- It stores locally the root directory hash.

If removing a file or directory:

- It removes its meta-file. In the case of deleting a directory, the meta-file is removed with it, since it is inside the directory.
- It updates the hash of the directory, where the removed file or directory was located.
- It updates all the parent directories up to the root.
- It stores locally the root directory hash.

When writing a block to a file the client performs the following steps:

- It encrypts the modified block locally.
- It computes the hash of the encrypted block locally.
- It writes the encrypted block to the remote file.
- It writes the hash in the corresponding meta-file.
- It updates the hash entry in the meta-file of the file's directory.
- It updates the parent directory hashes up to the root directory.
- It stores locally the hash of the root directory.

E.2 Verifying Data

Files are verified on a per block basis. To achieve that the client performs the following steps when reading a file:

- It reads the encrypted block from server.
- It computes the hash of the block.
- It verifies that that hash is present in the meta-file.
- It computes the hash of the meta-file.
- It verifies that the computed hash is in the parent directory's meta-file.
- It computes the hash of the parent directory's meta-file. It recursively verifies that each directory's meta-file's hash is in its parent's meta-file.
- It verifies that the root hash is identical to the locally stored root hash.

The algorithm for verifying a directory is similar:

- The client reads the hashes of each meta-file inside the directory.
- It computes the hash of the directory meta-file.
- It verifies the computed hash using the hashes of the parent directories up to the root.

F. Group Data Verification Scheme

Unlike single users, group members should be able to distinguish between legal changes on the group data by other members from illegal changes done by the remote server. It is necessary for group members to share information off the server in order to verify data on the server. This information is stored on a *Trusted Administrative Server (TAS)*. Similarly to single clients, the TAS needs to store only a small amount of data.

CryptoShare maintains the invariant that the TAS keeps the latest root hashes for all group members locally at all times. To maintain the invariant group members act according to the following protocol:

- After writing data, group members ask the TAS to update the hashes on the remote file server and then store locally the root hash.
- Before reading data, group members ask the TAS to verify the integrity of that data.

In addition, the TAS preserves the integrity of files, using the following verification protocol:

- When asked by a group member to update hashes, it stores locally not only the root hash of that particular group member, but it also checks whether it knows about root hashes of parent directories of the group member's *root*, and updates those, too. That way, it keeps the latest copies of all the root hashes it knows.
- Before verifying a meta-file hash, it checks whether it has a copy of it locally and compares that to the computed hash.
- If it does not have a local copy, then it retrieves the hash from the remote server (from the parent directory meta-file) and compares that to the computed hash.

The above actions taken by the TAS ensure consistency when verifying data. If the TAS does not update root hashes itself, but instead clients tell the TAS which root hashes to store, then inconsistencies can occur. We illustrate this, using the following example:

- User A mounts to `/` and updates the hash to the file `/home/A/foo`, so it stores the root hash of `/` to the TAS. Then user B mounts `/home/` and updates the hash of the file `/home/B/loo`. When user A tries to verify `/home/A/foo`, it will use the hash of `/home/` from the remote server (which is now different, because B updated the `/home/` hash on the remote file server and the TAS). When A reads the root hash for `/` from the TAS, it receives the old copy, which will be incorrect now.
- If the TAS updates root hashes instead, then when user B asks it to update hashes, it would update hashes up to `/` and then A would get the latest version.

Other than the above protocols, group members perform the same operations when reading and writing data to and from the remote server as in the single user case. Every group member presents the necessary credentials to authenticate herself to the TAS. It is the responsibility of the group administrator to define the authentication procedure for accessing the TAS.

V. IMPLEMENTATION

We made our modifications to the SFS client source for Linux. The server side was left untouched. SFS runs on the client as a user level process that forwards NFS calls to the SFS server located at the remote file server ([6]). Our modifications were made on the SFS-client process. We added a *Crypt* and a *Verify* module that are responsible for maintaining data secrecy and tamper resistance respectively. Figure 3 illustrates the client space and our modifications on the SFS client.

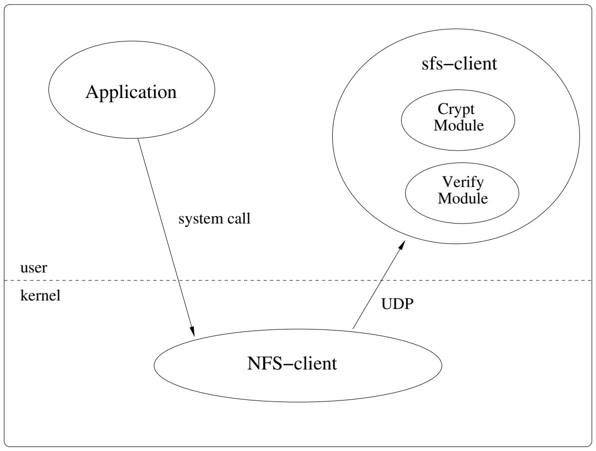

Fig. 3. Client using SFS. Crypt and Verify modules were added to the SFS-client process.

A. Cryptographic Tools

The Crypt and Verify modules use a set of cryptographic tools that we developed. We used standard cryptographic algorithms to design our encryption/decryption and hashing tools. We implemented both an asymmetric and symmetric key package. The symmetric key package is used to generate K_f keys for encrypting/decrypting file blocks. It uses 64-bit DES ([2]). The asymmetric key package is used to generate $Client.PK$ and $Client.SK$ and encrypt symmetric keys. It uses the 1024-bit RSA ([12], [1]). In addition we used MD5 for hashing data ([11]). Both cryptographic packages use the `openssl` cryptographic library implementations of DES and RSA.

B. Meta-file Tools

The Verify module uses a set of tools that deal with meta-file contents that we implemented. Our meta-file package handles read-write operations in meta-files and uses our cryptographic tools to update data inside the meta-files and verify the integrity of remote files. The package uses the algorithms for Merkle trees as described in [10] to implement the algorithms mentioned in Section E for updating and verifying information on our meta-file structure.

VI. Security Analysis

The adversary for our system is the remote file server. We assume that the file server has complete knowledge of the actions (reads and writes) performed by clients and it is able to distinguish between different clients. Furthermore, it is assumed to have enough storage to keep multiple copies of data.

A. Secrecy

Our file system provides data secrecy by encrypting file blocks with a file key K_f. Since K_f is encrypted using a user's public key, only the holders of the corresponding private keys can obtain K_f and decrypt blocks. Furthermore the decryption occurs locally at the client. Our system holds the invariant that the remote file server only sees encrypted versions of K_f in the meta-files. This guarantees that the file server cannot obtain K_f if all the clients are trustworthy.

Although the server cannot obtain data directly, it can obtain information about the structure of a file, by comparing blocks. If all the blocks are encrypted using the same K_f, the cipher texts of blocks with the same data will be the same. Our system, provides the option of encrypting blocks with different keys, so files with contents of higher sensitivity can have an extra level of protection.

In addition, since filenames or directory names are not encrypted, the server can obtain information about the files by reading their names. This attack is effective however, only if a user puts sensitive information on the filename.

B. Tamper-resistance

In the case of a single client, our file system provides tamper-resistance by saving on the client the hashes of all the root directories the client owns. In the case of a group, the root hashes are stored in the TAS. The reason that the root hashes are not stored remotely (even if they were encrypted) is that the server can provide stale data to the client that would be both secure and verifiable.

If the server attempts to modify the data of a single user, the verification mechanism will detect the attack, since at some point during the Merkle tree verification algorithm the hash chain will break. Although the server can produce valid hashes of all the encrypted files up the user's root directory, it cannot however update the root hash, because it is stored locally on the client or the TAS, which is considered as trusted storage. A client can detect such an attack, because our system maintains the invariant that the client sees locally the latest version for all its root hashes. To maintain the same invariant for groups the TAS provides sequential consistency and serializes root hash updates.

Meta-files are what distinguishes CryptoShare files from regular files. They contain the necessary information for encrypting/decrypting CryptoShare files. A malicious server could delete a meta-file to deny service. However, since the hash trees are computed using the meta-files, that action can be detected, so such an attack cannot be successful.

By including a file name inside its meta-file we protect against attacks that modify the name of a file to deny service. When the client tries to compute the hash of the enclosing directory. It will search for the hash of the corresponding file. If the file is renamed illegally then the hash will not be computed properly and thus detect the inconsistency.

Since a malicious server has full control of its operating system, not only can it modify file names but it can modify the time on the server's clock and also the file meta data that are stored in the i-node blocks. Our system does not protect file meta-data. We believe that the most important meta-data for a file are its permissions. Since CryptoShare enforces the file permissions described in Table I by the use of cryptographic tools, such attacks will not have an effect.

Our system does not provide data recovery, so the server can still delete data if the cost of it being detected is not important. Therefore, clients are responsible for replicating highly sensitive data in many remote servers.

C. Permission semantics

As Table I shows, our system provides only secure and verifiable read-write permissions. For secure and verifiable read-only or write-only permission semantics matters become complicated. For secure and verifiable read-only and write-only access one cannot use symmetric encryption/decryption keys K_f anymore, since the same key is used for both encryption (write) and decryption (read). Instead, file blocks have to be encrypted using public-private key pairs (PK_f, SK_f).

For example, for secure read-only permissions a user would encrypt file blocks with her private key, so others can read her files, but not modify them. To make read-only permissions verifiable she would have to store her evidence that the files are correct in a public place that is tamper-proof (similar to the TAS). Similar techniques can be used for secure and verifiable remote write-only permissions. We believe that the additional complexity of including these permissions in our system is not justified by their practical use, so we do not consider them.

VII. Evaluation

We evaluated our system in terms of its flexibility and performance.

A. Flexibility

Our focus was to provide a fair amount of flexibility both for server administrators and for clients. Without having to modify the remote file server CryptoShare provides data secrecy and tamper-resistance.

In addition, the design of CryptoShare allows encrypted and verifiable files and directories to coexist with unencrypted ones on the same underlying file system. Providing this feature adds only storage cost on the client, since it might need to store more root hashes compared to the case where CryptoShare directories contain only CryptoShare directories and files.

B. Performance

In order to measure the performance of CryptoShare, we measure the additional latency added to the system. Here we present the encryption/decryption and verification primitives that CryptoShare reads and writes use. We measured these times locally using the `time` call on a Pentium II machine running RedHat 8.0, so there were no network delays.

Table IV shows the time it takes to perform our encryption/decryption and verification primitives. The times were averaged over many runs, where each run performed each operation 100 times. The block verify and update tests were run on a file that was one directory deep (/DIR/File).

Operation	Time (ms)
Enc/Dec block with K_f retrieval	16.9
Enc/Dec block with K_f cached	0.5
Verify Block Hash	1.1
Update Block Hash	1.8

TABLE IV

PERFORMANCE OF ENCRYPTION/DECRYPTION AND DATA VERIFICATION PRIMITIVES.

From the results above we see that reading a block from a file one directory deep will incur an additional 18 ms of delay. 16.9 ms are spent retrieving the encrypted K_f, decrypting it with $Client.SK$ and decrypting the block with K_f. Verification of the block takes an additional 1.1 ms.

Unfortunately, we don't have a working TAS, so we lack experiments for reading and writing shared files. We also lacked time to conduct more extensive performance experiments and get more results on the performance both for single users. In the future we would like to measure the time it takes read and write multiple files and directories on a remote file server. For example, measure the time compress and uncompress a tar archive on the remote server with CryptoShare and without CryptoShare and compare the overall performance.

VIII. Future Work

In the future, we would like to implement our design of the Trusted Authentication Server and provide shared file access.

Our current version of CryptoShare does not deal with client and server failures, NFS errors, or network partitions. Currently, if a client fails before completing the hash updates all the way to the root, then when it tries to verify valid data, it will get an error. We believe that designing our data verification mechanisms to take into these failures adds considerable robustness to our system.

IX. Conclusion

We have presented *CryptoShare*, a secure verifiable shared remote file system, that allows single users and groups to share data in an untrusted remote file server. The system guarantees data secrecy and tamper-resistance, but not recovery in case of illegal modifications from a malicious server. It provides limited read/write permissions for simplicity and it can be used on an unmodified file server. We have provided flexibility for the cost of performance and we acknowledge that additional work need to be done to make its performance comparable to other cryptographic file systems.

REFERENCES

[1] RSA Laboratories, PKCS #1 v2.1: RSA Cryptography Standard.

[2] National Bureau of Standards, Data Encryption Standard, 1977.

[3] BLAZE, M. A Cryptographic File System for Unix. In *1st ACM Conference on Communications and Computing Security, Fairfax, VA* (Nov 1993).

[4] CATTANEO, G., CATUOGNO, L., SORBO, A. D., AND PERSIANO, P. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *FREENIX Track: USENIX Annual Technical Conference, Boston, MA* (Jun 2001).

[5] MAHESHWARI, U., VINGRALEK, R., AND SHAPIRO, W. How to Build a Trusted Database System on Untrusted Storage. In *USENIX Symposium on Operating Systems Design and Implementation, pages 135–150* (Oct 2000).

[6] MAZIERES, D. A toolkit for user-level file systems. In *USENIX Technical Conference, Boston, MA* (June 2001).

[7] MAZIERES, D., KAMINSKY, M., KAASHOEK, M. F., AND WITCHEL, E. Seperating key management from file system security. In *17th ACM Symposium on Operating Systems Principles (SOSP), Kiawah Island, SC* (Dec 1999).

[8] MAZIERES, D., AND SHASHA, D. Building secure file systems out of byzantine storage.

[9] MAZIERES, D., AND SHASHA, D. Don't trust your file server. pp. 113–118.

[10] MERKLE, R. C. A certified digital signature. In *Proceedings on Advances in cryptology* (1989), Springer-Verlag New York, Inc., pp. 218–238.

[11] RIVEST, R. The md5 message-digest algorithm, April 1992.

[12] RIVEST, R. L., SHAMIR, A., AND ADELMAN, L. M. A METHOD FOR OBTAINING DIGITAL SIGNATURES AND PUBLIC-KEY CRYPTOSYSTEMS. Tech. Rep. MIT/LCS/TM-82, 1977.

[13] ZADOK, E., BADULESCU, I., AND SHENDER, A. Cryptfs: A stackable vnode level encryption file system, 1998.