NeFS: Network Encrypted File System

Robin Cheng, Xunjie Li, Ronnachai Jaroensri
{robinc, xunjieli, tiam}@mit.edu

6.858 Final Project December 13, 2013

1 Introduction

In recent years, many convenient file sharing solutions have been developed: Google Drive[1], Dropbox[2], Microsoft's Skydrive[3], etc. However, in all these commercial products, end users do not have control over file servers. It is not uncommon to see cases of data breach on remote servers[4]. On the other hand, currently several encrypted file systems exist, such as BitLocker[5] and TrueCrypt[6], but they only encrypt local file data and do not support sharing.

Our goal is to develop a remote encrypted file system that enables easy file sharing with a strong data confidentiality guarantee. We are able to find several recent projects in this area. The first is SPORC(2010)[7], a project that deals with fork-consistency in group collaboration on an untrusted server. Depot(2011)[8] improves liveness of an untrusted server, by making the file system distributed. K2C(2012)[9] proposes an lazy key revocation scheme to improve the performance of a remote file server. Authenticated Storage Using Small Trusted Hardware(2013)[10] designs a novel solution to use a small trusted hardware to ensure the data security of file server. However, these solutions deal with very specific problems, and thus we do not find them very relevant to our goal. Nevertheless, it shows that there are many possible ideas and novel solutions that we can look into.

Our project builds upon SiRiUS(2003) [11], a solution to securing remote untrusted storage. SiRiUS embeds file encryption and file signature keys directly with the file. Users gain access to files by gaining access to the file keys encrypted using their public key(a trusted PKI will be needed). This way, sharing of file can be done with minimal out of band communication. We designed NeFS, a network encrypted file system, by extending SiRiUS's design to meet the security requirement set out by 6.858 course staff [12]. In particular, our system places minimal trust on the server, which makes it very easy to defend against attack by a malicious server, or a malicious client connecting

to the server. This, however, comes at a cost of more complicated client, as we shall see in section 3 and 4.

In this paper, we will explain the design of NeFS, and illustrate the key features of a prototype. We will discuss performance of our design, and conclude by noting the possible future directions of this project.

2 Threat Model

As shown in Figure 1, there are three system components in NeFS -- clients, a file server and a public key repository. We assume the public key repository is a trusted party which always provides correct public key for each user in the system. The primary goal of NeFS is to ensure data confidentiality and integrity while allowing sharing of files. We address two threats against confidentiality and integrity of user data: malicious client and compromised server.

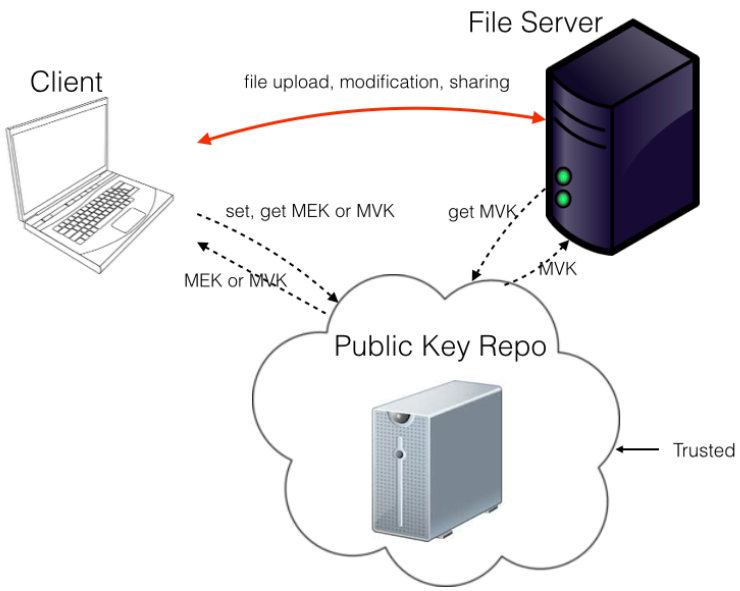

Figure 1 NeFS System Diagram and Threat Model

Malicious Client: An adversary may have full control of his client machine and possibly has a legitimate account within the system, but he should not be able to read the plain texts of the files that he doesn't have access to. Furthermore, he should not be able to compromise the integrity of data, by creating new file, or replacing content of a given file that he doesn't have proper permission to. The server may collude with him, and

their goal to compromise confidentiality (read plain text), or integrity (altering file content) should not succeed or go undetected.

Compromised Server: A file server may be compromised, either by a hacker or a curious administrator. An adversary who has full control of the server machine should not be able to read plain text of the data store on it. Of course, since the file is store on the server, the adversary will be able to write anything on the disk, replacing any content that the [legitimate] user may have put there. Equally bad, the adversary may decide to supply the content of one file with another. However, this class of attacks should not go unnoticed, and the user will be able to tell if their data has been tampered with the next time they communicate with the server.

Note that the current version NeFS does not provide freshness guarantee, and the server may return stale file resulting in successful rollback attack.

3 Design

3.1 Overview

The key idea of our design lies in the cryptographic keys. Each file on the server is encrypted by a *symmetric* file encryption key (FEK) and signed by an *asymmetric* file signature key (FSK). Sharing of the file is done by giving collaborators access of FEK or both FEK and FSK depending on whether read or write access to grant. Each user holds an *asymmetric* master encryption key (MEK), and an *asymmetric* master signature key (MSK). The FEK and FSK are encrypted by owner's MEK for each collaborator, and are embedded directly in the metadata accompanying the file itself.

To sum up, Figure 2 illustrates the keys used in NeFS:

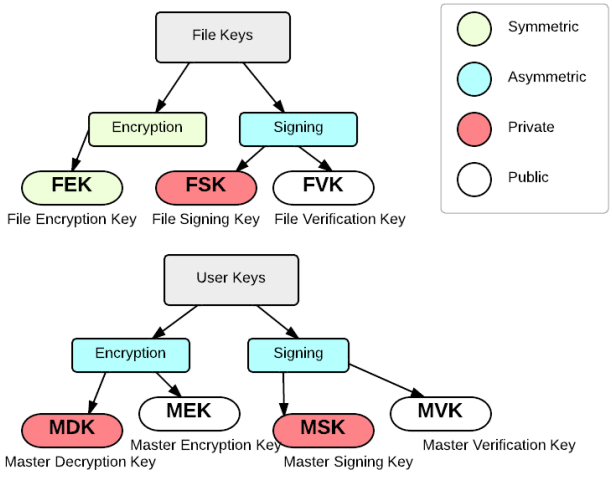

Figure 2 Key Overview

In Section 3.2, we will explain these keys, as well as how they interact with files. We use a shorthand, `{content}key` , to mean `content` is encrypted with `key`.

3.2 File Encryption and Representation

Upon uploading a file to the remote server, NeFS client will generate file keys for the file. NeFS client then generates a **data file** and a **metadata file**.

A **data file** consists of:

1. File data encrypted with FEK (using AES).
2. a SHA-1 hash of file data signed with FSK.

Figure 3 Data File

A **metadata file** consists of:

1. A owner encrypted key block, tagged with username of the owner
    a. Containing FEK and FSK encrypted with owner's MEK
2. An encrypted key block for each collaborator, with unencrypted tag which includes username of the collaborator, and permission flag (read/write). The key block is encrypted with the collaborator's MEK, and contains:
    a. both FEK and FSK for collaborator who has write access
    b. FEK for collaborator who has read-only access
3. *Unencrypted* File Verification Key (FVK)
4. *Encrypted* file metadata block:
    a. File ID
    b. A boolean of whether the file represents a directory
    c. timestamp of the last metadata modification time
5. A signature (signed with MSK) of the hash of the metadata content

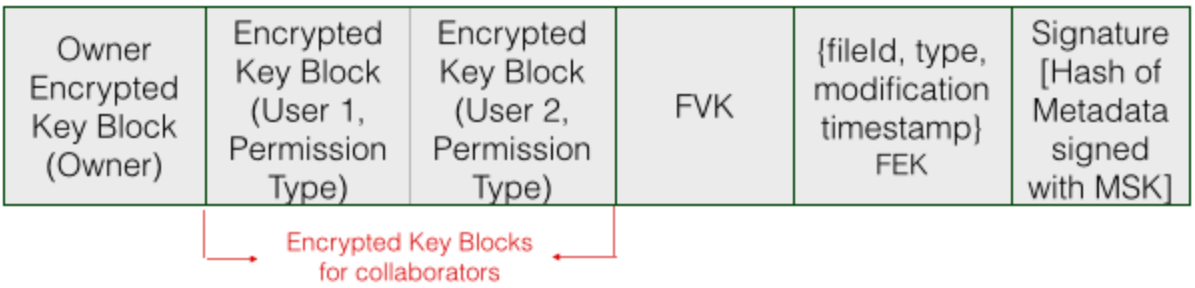

Figure 4 Metadata File

3.3 System Components and Interaction

As illustrated in Figure 1, there are three components in NeFS. In this section, we will explain the responsibilities, and document the major operations for each components.

3.3.1 Client

NeFS client provides an interface for users to interact with the remote file server. Because files are encrypted on the server, the server itself does not know file names, types, or directory structures. NeFS client hides this complexity of an encrypted file system from the end user, with its convenient APIs that facilitate file lookup, retrieval, modification, removal, and sharing. In this section, we describe different aspect of the client, and how it abstracts low-level detail from the end users.

Directory Structure and Changing Directory

The server has a flat file hierarchy, and all files are referred by their file IDs which consist of username, underscore, and an integer number, as illustrated in Figure 5. Directory structure is exposed entirely by the client. Each directory is a file, mapping file names to file IDs. These file IDs are the way by which the client and the server identify a file. File IDs are generated upon file creation by the client, and do not reveal anything about the file name or content. All directory files are encrypted just like regular files. The user continues to refer to the file by its filename, as long as they have the read permission on the containing directory file, and on the file itself.

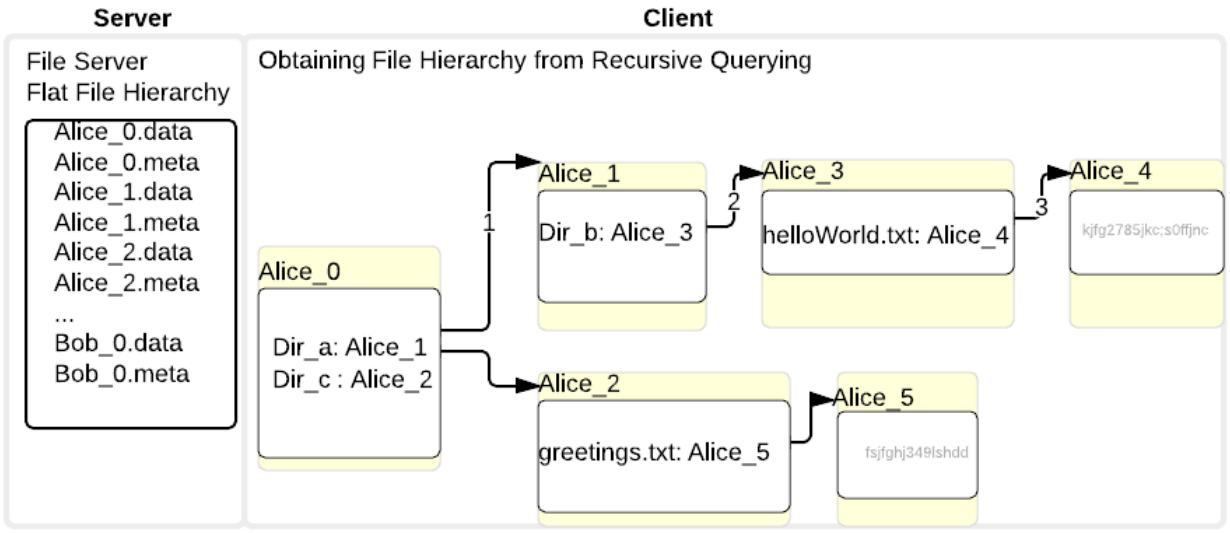

Figure 5 Flat Hierarchy on File Server and Obtaining Hierarch by Recursive Querying. Alice's root directory has a file ID of Alice_0. Fetching "/Alice/Dir_a/Dir_b/helloWorld.txt" involves recursive querying of three containing directory files.

The client keeps a local copy of the current remote directory file, and its parent directory. The current directory starts at the user's root directory, and changes upon user actions (eg. "cd"). When a user "cd" into a directory, the client fetches the new directory file, decrypts and perform checks. It then updates its internal state (current directory and its parent directory). If the user doesn't have read access to the file, the client reports the error, and restores the previous state.

Upload a File

The client checks if the file already exists (by look up the name given by user). If the file doesn't exists, the client generates file keys (FEK, FSK, and FVK) for the local file, along with metadata as specified in section 3.2. The client also generates a file ID which an integer incremented from the last used value (saved from session to session, for the user's life time). It also adds an entry to the directory mapping. If the file does exists (there is a directory mapping from file name to file ID), it recognizes the operation as updating the file, and retrieves FEK and FSK necessary for file modification. The client finally encrypts the data file with FEK, signs it with FSK, and uploads the files to the server.

Read a File

The client obtained the file ID from directory mapping (user refers to file by file name, usually), then it downloads the desired data file (correspond to the requested file). The

client decrypts the file as followed: it first obtains the encrypted key block (see Figure 4) that corresponds to user's username, and decrypts it using the user's MDK. Finally, the client verifies the integrity of the file in two steps: 1. verify the authenticity of the metadata file using the owner's *M*VK, and 2. verify the integrity of the data file using the *F*VK (embedded in the metadata file). Once the client obtains the desired data file, it decrypts the data file using the file's FEK. Then it returns the decrypted file to the user. The client may need to recursively query directory files in the path to obtain the final file ID, but since directory file is the same as regular file, reading a directory file follows the same procedure except the result is used to update the client's internal state instead of being returned to the user.

Note that, without read access, the user can at most see who owns the file, who has access to it, and the file verification key (from the metadata file), but not the file type, file ID, or the plain text of the file.

Link/Unlink a File

Linking and unlinking involves modification to the directory file. When the user wishes to link a file, the client looks up the file ID of the containing directory (which is defaulted to the user's root directory), decrypts it, adds the desired mapping, re-encrypts, and uploads to the server. Unlinking works in the opposite direction, the client removes the directory entry. Note that write permission on the destination directory is required in order to perform these actions.

Rename a File

Renaming a file involves update the mappings in the containing directory file, which is equivalent to modify the content of directory file. The file ID remains the same.

Share a File

When a user wants to share a file with another user, the client obtains MEK of the collaborator from the public key repository, encrypts FEK or FSK as appropriate into a key block tagged with the collaborator's username, puts it in the metadata file, updates the metadata file on the server. Note, only the owner of a file is allowed to carry out sharing, since the metadata is signed with the master signature key.

To facilitate file access for users, each user has a google doc-styled "*shared*" folder living separately from their root. For instance, Alice may "*befriend*" Bob, and create a folder

called “Bob” in her shared folder, and give Bob write access to that folder. When Bob shares a file with Alice, in addition to giving adding Alice’s key block to the file, Bob’s client will also link the file to Alice’s shared folder titled “Bob”, so Alice has access to all files shared with her without having to remember file ID of each file.

Unshare a File

Unsharing a file is slightly more complicated. In addition to removing user’s FEK and FSK from the metadata, the client will also need to generate a new FEK and FSK, and re-encrypt the file, and re-sign the file and metadata. This is because the user being revoked access may have saved the encryption key offline, changing the keys completely, will prevent his future access to the file, while storing keys in the metadata avoids notifying all other collaborator of the change.

Account Creation

When a user registers for an account, the client generates MEK, MDK, MSK, and MVK for the user. The client then registers the user with the public key repository, and sets the public keys (MEK and MVK) on the repository. The client will keep the private keys (MDK and MSK), in addition to MEK and MSK locally for convenience and offline operations.

3.3.2 Server

NeFS server has a flat file hierarchy (see Figure 5) , and is incognizant of file information other than who the file’s owner and collaborators are. It enforces access control, and prevents unauthorized users from corrupting file data. It exposes the following APIs to the client. In this section, we will explain what these APIs do, and how access control is incorporated. Here we assume that client with username, client_id, has been authenticated (See 3.3.4 Authentication).

`READ_FILE(client_id, fileID):`

> The server checks the encrypted key blocks of the metadata file correspond to file ID, `fileID`. The server makes sure there is an encrypted key block correspond to the user with username, `client_id`, and the permission type is `PERMISSION_READ` or `PERMISSION_WRITE`. When the verification passes, the server responded with `{metadata file, data file}`.

`READ_METADATA(client_id, fileID):`

This API is identical to `READ_FILE`, except it returns `{metadata file}`.

`UPLOAD_FILE(client_id, fileID, metadata_file, data_file):`

The server checks that `fileID` is well formatted, and is in the client's namespace. It also checks whether a file,with the same id exists. If all checks pass, the server saves the metadata_file and data_file under `fileID`, with corresponding suffixes (see Figure 5).

`MODIFY_METADATA(client_id, fileID, metadata_file):`

The server checks to see if the client is the owner of the file, by verifying whether the owner key block is tagged with `client_id`. Only if the client is the owner of the file, the server replaces the current metadata file with `metadata_file`.

`MODIFY_DATAFILE(client_id, fileID, data_file):`

The server checks to see if the client has write permission, by verifying there is an encrypted key block with `PERMISSION_WRITE` corresponded to the client. Only if there is, the server replaces the current data file with `data_file`.

`REMOVE_FILE(client_id, fileID):`

The server only allows the owner of the file to remove the file. Checking involves verifying the fileID and the owner's encrypted key block in metadata file.

3.3.3 Public Key Repository

The public key repository is a trusted component. It can be implemented using an external trusted key distribution infrastructure, such as PGP public key servers and Identity Based Encryption (IBE) master key servers. In the real deployment, the public key repository should provide functionality for the user to securely create and account, and securely set/reset their public key. The getter can be open to public. At the minimum, it provides the following,

`SET_MASTER_ENCRYPTION_KEY(KEY) :`

sets MEK of the client

`GET_MASTER_ENCRYPTION_KEY(USER_ID):`

gets MEK of a client with username USER_ID

`SET_MASTER_VERIFICATION_KEY(KEY):`

sets MVF of the client

`GET_MASTER_VERIFICATION_KEY(USER_ID):`
gets MVK of a client with username USER_ID

3.3.4 Authentication and Communication Protocols

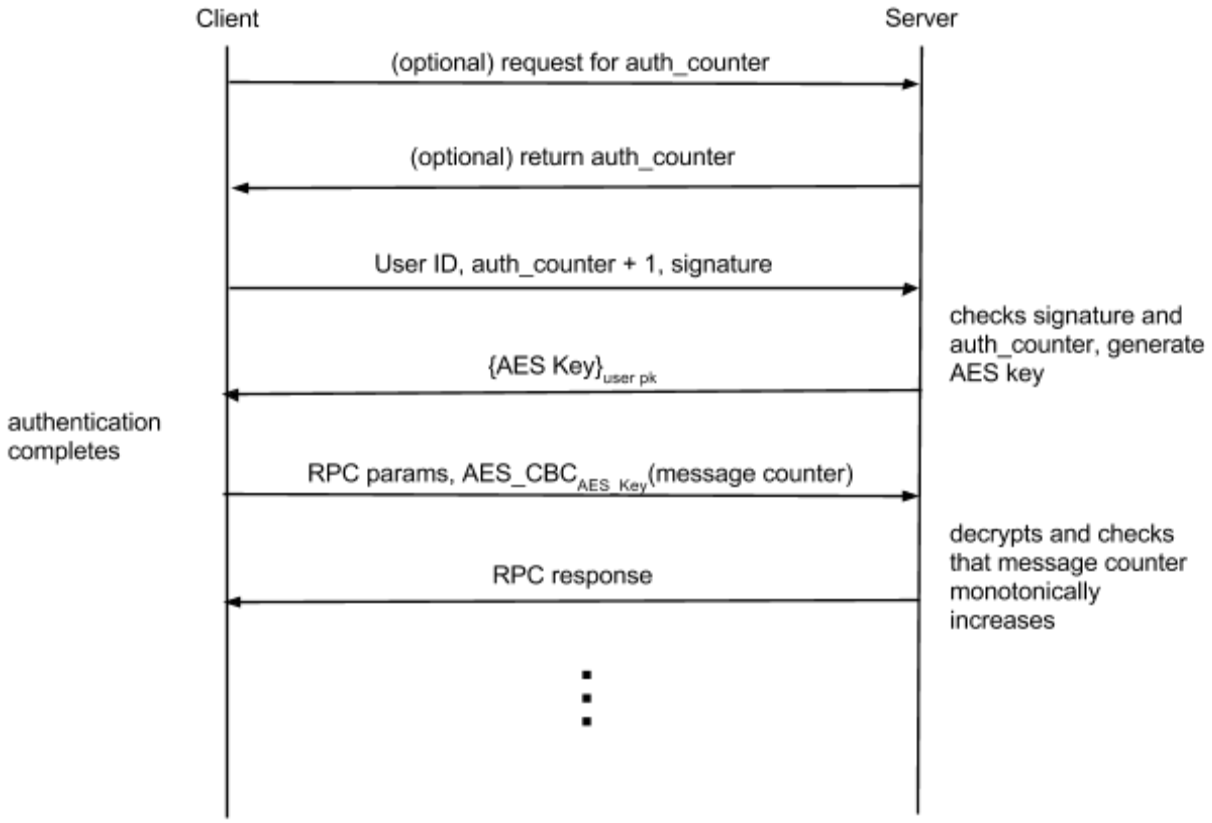

Figure 6 Authentication Protocol

The authentication process is essential for the server to ensure that a client is who it claims to be. Because we use RPC instead of raw sockets, we do not have persistent state per session; this is similar to HTTP, where browsers need to keep cookies in order to authenticate with the server for every request. We do something similar here, by designing a simple authentication protocol as shown in Figure 6. Note that one obvious choice for authentication is to set up the server as an HTTPS server, or to make the RPC go over SSL. However, setting up HTTPS server or SSL is tedious and we did not choose to do so in a prototyping environment.

For each user, there is an incrementing integer called the “authentication counter” stored at the server. The counter is initialized to zero at account creation. Every time the client wants to start a session with the server, it asks for the authentication counter from the server, increments it by 1, and signs the new counter with its MSK, and sends it over to the server. The server then checks if the number is greater than the last one it has seen, and checks that the signature is correct. If the authentication counter sent by the client is valid as well as the signature, then this proves that the client does in fact have the ability to sign messages with MSK. The reason why we use an incrementing counter is to prevent against replay attacks, as an authentication message can only be used once. After validation, the server generates a random AES key, and sends it back to the client, encrypted by the client’s MEK. The client decrypts the AES key and use this key as the session key. If the client is legitimate, the decryption will succeed and provide the same AES key as generated by the server.

Along with each subsequent RPC call, a token is sent to the server. The token is the encryption of the message counter, which starts with 0 right after authentication completes, and increments for each RPC call. The encryption uses the AES key in CBC mode, which includes a random initialization vector so as to make sure that the input to the AES cipher is random enough (it’s not obvious from a cryptographic point of view whether this is really needed, or if AES without an IV is already secure for this purpose). The server decrypts the message counter and verifies that it monotonically increases. Note here that we did not simply use a stream cipher, because individual RPCs can theoretically be lost.

The authentication protocol is specific to this implementation. In a real implementation it would be preferable to use a more established protocol such as SSL.

4 Implementation

We implemented a prototype of NeFS in python to illustrate the feasibility of our design. Our prototype has a shell interface, offering convenient UNIX style commands. Although we did not integrate with an existing OS, it is in theory easy to do if given sufficient time. In this section, we will explain how we implemented our prototype.

4.1 Architecture

The implementation consists of four modules: common, server, client, and

public_key_repo:

- common: Cryptographic primitives, authentication protocol, data packing, configuration
- server: File server (RPC server), file manager (stores files on server's disk)
- client: RPC client, user-friendly shell
- public_key_repo: A simple implementation of a trusted public key repository server (RPC)

4.2 Cryptography

We use pycrypto as the underlying cryptographic library. On top of this, we provide convenient wrapper functions to abstract out the details in pycrypto. The very import ones are:

- Generation of symmetric key. This returns a random key from pycrypto's cryptographically secure Crypto.Random module.
- Generation of asymmetric key. This is done directly by the Crypto.Public.RSA module and returns as a tuple of (N, e, d) for easy processing.
- Encryption/decryption using symmetric key. This uses Crypto.Cipher.AES in CBC mode.
- Encryption/decryption of small messages using asymmetric keys. This uses Crypto.Cipher.PKCS1_OAEP.
- Encryption/decryption of large messages using asymmetric keys. Public key encryption does not directly support encrypting arbitrarily large messages due to the finite RSA modulus. So we split the message into chunks and separately encrypt each chunk in ECB mode (chunk-by-chunk independently). This still provides IND-CPA security, but not IND-CCA2 security anymore due to malleability by reordering blocks. However, because the only usage of encrypting long messages with a public key is to encrypt keys in the metadata, which itself is signed with a signature, an adversary does not have the power to choose ciphertext. In other words, this scheme is still secure.
- Signing/Verification of messages using asymmetric keys. This uses Crypto.Signature.PKCS1_PSS after SHA1 hashing.
- Importing/exporting of keys. This converts keys between string and tuples of RSA values, for communication and storage of keys.

4.3 Data packing

It's an important consideration of what format various data should be stored in. For example, encrypted keys in the metadata needs to be stored in a structured way, and RPC requests need to be serialized to binary streams. Such serialization and deserialization needs to be done in a safe way, so that adversaries who modify serialized

streams cannot exploit code that deserialize the stream.

We considered several existing approach to doing this:

- JSON. This scheme is easy to use and deserialization cannot execute arbitrary code, but it cannot embed binary streams directly (such as encrypted data), and doing so would require string escapes or base64 encoding, both of which increase data size considerably. Furthermore, JSON does not restrict the data types or structure, meaning that code that expects a string as an argument may get a number instead, and if not careful, this can lead to exploitable bugs.
- XML (as in xmlrpclib). This is very similar to JSON, and has similar issues as JSON. Furthermore, it is noted in the xmlrpclib documentation that the XML parser may not be secure.
- Google's Protocol Buffer. Protocol buffers provide strict data structure guarantees, and encodes data in a very efficient manner. However, setting up protocol buffers is very inconvenient, and the API is not as flexible, so it is not a great option to use when prototyping.

Since none of these approaches are perfect for what we want to do, we invented our own scheme for packing data. There are two primitives for our scheme:

1. Packing a tuple of strings. This is done by writing a 4-byte string count, followed by a sequence of 4-byte string lengths, followed by a concatenation of the strings. When decoding, the count and the lengths are verified to ensure that they are valid, including checking whether the lengths are non-negative and whether they sum to the correct length from the encoded stream.
2. Packing arbitrary data involving six data types. NoneType, int, str, bool, tuple, and list. This is done by encoding a type tag (a constant byte predefined for each type), followed by the encoded stream of that type, which are:
    - NoneType: The empty stream
    - int: A 4-byte stream (packed with "<i")
    - str: The string itself
    - bool: a byte of `0x00 or 0xff`
    - tuple: Pack each element of the tuple into binary strings, and then pack the binary strings using string packing.
    - list: Same as tuple.

When decoding packed data, the caller may optionally specify a format specification to mandate the structure of the decoded data stream, by giving a prototype of the decoded stream. For example, a prototype of [("", 0)] indicates that the stream must represent a list of 2-element tuples, where each tuple has elements of type string and integer. If the decoded data does not match this format, then an exception is thrown.

This way, we provide code safety just like protocol buffers, and encode almost as efficiently (binary strings are encoded as is), while at the same time providing convenience when writing the code.

4.4 RPC

We considered using xmlrpclib, but that suffers from the XML problems discussed previously in Section 4.3. And we could not find an RPC library that is both simple to use and encodes data safely and efficiently.

As a result we used an RPC library derived from the rpclib we were given in the labs and modified it to encode and decode requests and responses in the data packing format from Section 4.3. Specifically, each request or response is sent as a 4-byte length of the request/response body followed by a packed stream of the request/response body.

4.5 RPC Relays

After integrating all the components we discovered a serious drawback from using the staff-provided rpclib, and possibly other RPC libraries: each socket is handled on a new fork, so we cannot keep any shared state in the server, or they will not actually share state, since each fork has its own copy of virtual memory. We considered several solutions to this problem:

- Rewrite everything to use databases and keep no state at the server itself. This would require a lot of work and we would have to think carefully about race conditions. We decided not to do this due to this being a prototype and there being no need to support multiple users' concurrent actions in a prototype implementation.
- Rewrite the RPC library or find a new RPC library so that synchronous I/O is used, i.e. with select(), so that the server can still be single-threaded. We found no easy-to-drop-in implementations of such RPC library or socket implementation.

So we decided to take advantage of NodeJS's event-based I/O and set up a relay server that accepts socket connections just like the actual RPC server would. The relay server splits the streams into individual requests using the 4-byte length information, and sends requests to the actual server on a single RPC connection. The relay server keeps track of the requests sent, so that when it receives responses from the real server, it knows which user to send the responses to.

Even though the relay server is more of a hack, the same idea can be used in other contexts to opaquely setup load balancing, i.e. accept requests from any relay node, and based on the load of individual servers, forward the request to an appropriate server.

4.6 Server and file manager

The server is a simple RPC server using techniques described in Section 4.4 and 4.5, and exposes APIs described in Section 3.3.2. The file manager implements helper functions and access control.

4.7 Client

The client is where most of the cryptography takes place. In our implementation, we have defined abstractions such as creating files, uploading files, downloading files, creating directories, linking and unlinking files, sharing files, etc. Each of these operations eventually translate into reading and writing of metadata and data files, and only encrypted versions of the metadata/data files are ever sent to the server.

4.8 CLI Shell

We implemented a CLI environment (like that of Python) that supports the following commands:

1. cd <dir>: Change the current directory
2. ls: List the contents of the current directory
3. ul <local> <remote>: Recursively upload from local path to remote path
4. dl <remote> <local>: Recursively download from remote path to local path
5. rm <remote>: Recursively delete remote path
6. mv <from> <to>: Move/rename in NeFS
7. shr <path> <user> <shared name>: Share read permission to a file or directory to a user under a certain name
8. shw <path> <user> <shared name>: Same as above, except also granting write permission (read permission implied)
9. unshare <path> <user>: Unshare the file/directory from the user
10. friend <user>: Set up a shared folder for the user so that the user can now share files with you

A remote path can be relative to the current directory, or be an absolute path that starts with either:

- "~user": The root directory of the user. If the user string is omitted, refers to the user's own root directory.
- "!user": The shared directory of the user. If the user string is omitted, refers to the user's own shared directory.

5 Security Analysis

NeFS has a simple design, and addresses all the requirements in 6.858 Final Project's handout[12]. In this section, we will briefly explain, in context of our threat model, how our design meets the requirements for data security.

NeFS addresses security requirements [12] as follows:

- File names (and directory names) should be treated as confidential.
    - file names and directory names are only available in the directory files, which are encrypted just like ordinary files.
- Users should not be able to modify files or directories without being detected, unless they are authorized to do so.
    - The files are signed with FSK, which is not available without write access, and without the FSK, an adversary cannot modify the content of the file and produce a valid signature at the same time.
    - An adversary can perform a rollback attack and provide an older version of the file with the correct signature. However, since our system does not provide freshness guarantee, this attack is beyond the scope of our project.
- If the server is not malicious, unauthorized users should not be able to corrupt the file system.
    - Users are authenticated. Without both the private key and the signing key of the user, an adversary cannot impersonate the user.
    - When modifying the metadata file, the server checks whether the user owns the corresponding file.
    - When modifying the data file, the server checks whether the signature on the file is valid, using the file verification key stored in the metadata file.
- The file server should not be able to read file content, file names, or directory names.
    - File content is encrypted with FEK.
    - The FEK is encrypted with the MEKs of all the users who have read access. The server does not have any MDKs, so it cannot decrypt the FEK.
    - File names and directory names are in directory files which are ordinary files and are thus secure for the same reason.
- The server should not be able to take data from one file and supply it in response to a client reading a different file.
    - The fact that the metadata is signed with the owner's MSK ensures that the FVK and the file ID must be supplied in the correct combination by the server, i.e. the server cannot take the FVK of one file and the file ID of a different file and combine them into a valid metadata file.
    - Therefore if the file ID matches the requested file ID, the FVK must also

be correct. Then, the server cannot supply the contents of a different file as it will fail the signature check against FVK.

- A malicious file server should not be able to create or delete files or directories without being detected.
    - File/directory creation: A malicious server cannot sign any metadata file, so any metadata file that it creates will not pass validation by any client. And without creating a metadata file, a file/directory cannot be created.
    - File/directory deletion: Even though the server can simply remove the metadata and data file from its disk, it cannot unlink the file from its parent directory (as that requires modifying the directory file), so when the user tries to access it through the link, it will detect that the server fails to provide the file.

6 Performance Analysis

The most expensive operation in this design is the generation of asymmetric keys. We use PyCrypto to generate 1024-bit RSA keys. This is the smallest size allowed by the PyCrypto library, and sizes above this (such as 4096) takes unacceptable time to generate.

The generation of asymmetric keys is involved in two situations:

1. The creation of a user. This is not a big concern at all because this is done only once per user.
2. The creation of a file/directory. This is especially noticeable when recursively uploading a local folder. This is one reason why downloading is significantly faster than uploading. To mitigate this, one may pre-generate keys when the system is idle.

We benchmarked several uploading and downloading test cases. For each test case, we repeat the same operation three times, and take the average time. The timing results are shown in the following table:

Action	Time 1 (sec)	Time 2 (sec)	Time 3 (sec)	Avg. Time (sec)
Uploading one empty file	0.83	0.97	0.8	0.87
Uploading one file of size 1MB	0.91	1.39	0.79	1.03

Uploading one file of size 100MB	3.37	2.99	2.19	2.85
Uploading a 6.858 lab folder (containing 93 files and folders)	83.86	80.63	78.41	80.97
Uploading the same lab folder but with pre-generated keys	38.10	40.81	36.95	38.62
Downloading one empty file	0.12	0.10	0.12	0.11
Downloading one file of size 1MB	0.13	0.13	0.13	0.13
Downloading one file of size 100MB	1.42	1.61	1.38	1.47
Downloading the same lab folder	10.38	10.4	10.36	10.38

The performance of our prototype seems satisfactory. Most operation takes time in the order of seconds. The bulk of computation time seems to go into cryptographic operation. One may try to use a different cryptographic library or implement the system in a faster language like C to gain performance.

Note also that it is possible to modify the server API to allow more fine-grained read, such as reading the data file without reading the metadata file, or letting the server decode certain unencrypted parts in the metadata file. This can lead to increased performance and decreased bandwidth. However, in this prototype, which we focus on functionality and debuggability rather than performance, we do not include such optimizations.

The code of our prototype is located at https://github.com/xunjieli/malicious .

7 Future Work

There are one major limitation of our design. First the server can carry out rollback attacks, without being detected by the users. Specifically, the server can replace an existing pair of data file and metadata file with an older version. The signature and checksum will still pass client's checks. A possible extension to this system is to add a scheme to guarantee freshness, such as those described in [7] and [8].

Furthermore, because file sharing is tied to the user's master encryption key, changing

user's security credential can be extremely difficult. If a user's MDK is leaked, all the files that the user has access to will be compromised. The user will need to track down every single file that he has access to, and request the file owner to re-encrypt the file key for that him. This is a major drawback that comes with a method where the encryption key is tied to a single file. We presently do not have a solution for this without relaxing security guarantee (enforcing permission per folder instead of per file, for example), and we believe this could be an interesting future work.

Another issue with our design is whether to allow "mixed ownership". What this means is, suppose user A shares with user B a folder owned by A, and gives user B write access on that folder. Intuitively, user B should be able to create, rename, and delete files in this folder. However, because only the owner of a file can sign the metadata of that file, if user B were to create a file in the folder, that file would have been owned by user B, which leads to mixed ownership in that folder, since the folder is owned by user A but a file inside is owned by user B. This is not a big problem, because a file being in a folder is only a matter of linking, but it introduces a lot of usability inconsistencies. For example, if user A wants to further share the folder with user C, user A would not be able to grant any access on the new file (which is owned by user B) to user C, creating an inconsistent experience for both user A and user C. It would be interesting to find a way to work around this, so that users can create files in a shared folder, while still being able to transfer the ownership to the owner of the folder.

8 References

[1] Google Drive. http://drive.google.com

[2] Dropbox. http://dropbox.com

[3] Microsoft SkyDrive. http://skydrive.live.com

[4] A. Chen. GCreep: Google engineer stalked teens, spied on chats. *Gawker*, September 2010. http://gawker.com/5637234/.

[5] BitLocker. http://windows.microsoft.com/en-us/windows7/products/features/bitlocker

[6] TrueCrypt. http://ww.truecrypt.org

[7] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proceedings of the 9th Symposium on Operating Systems De- sign and Implementation*, Vancouver, Canada, October 2010.

[8] J. Li, M. Krohn, D. Mazie`res, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 91– 106, San Francisco, CA, December 2004.

[9] Zarandioon, Saman, Danfeng Daphne Yao, and Vinod Ganapathy. "K2C: Cryptographic cloud storage with lazy revocation and anonymous access."*Security and Privacy in Communication Networks*. Springer Berlin Heidelberg, 2012. 59-76.

[10] H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the 10th Network and Distributed System Security Symposium*, 2003.

[11] H. Shacham, N. Modadugu, and D. Boneh. Sirius: Securing remote untrusted storage. In *Proceedings of the 10th Network and Distributed System Security Symposium*, 2003.

[12] http://css.csail.mit.edu/6.858/2013/labs/lab7.html