

# XSFS: An Extensible Steganographic File System

Andrew Stone

**Abstract:** Currently there are several cryptographic file systems with various levels of functionality and security, as well as numerous other designs for securing data on untrusted storage. However there is a limited amount of work that incorporates steganography, or data hiding, into the file system. While these few steganographic filesystems may be functional and efficient, they rely on administrative control over the server to some extent. I propose a design in which a user can successfully create a steganographic filesystem on any remote server in which he has read/write access to a directory. By providing correct passphrases a user can extract several directory trees from the remote server which can only be constructed on the client machine. Furthermore, an attacker with complete control over the remote server and network would be unable to derive any useful information as to the structure of the directory trees or how much information is actually stored remotely.

## 1. Introduction

Securing data on untrusted storage is not a new idea. A number of file systems exist that utilize strong cryptography to restrict access to data. The first of these was CFS[2], followed by TCFS[4], Cryptfs[14], and NCryptfs[13]. All of these filesystems have in common the ability to take raw data and encrypt it before storing it, hence implicitly not placing trust in the server. They all have their own strengths and weaknesses, but are generally well understood and provide a high degree of security. An attacker cannot read the encrypted files in a computationally feasible way without the proper keys.

However, by analyzing what files are being accessed most frequently, what size they are, and the general directory layout of the filesystem (even with encrypted filenames), an attacker can infer much about what is contained in the system and the importance a user places upon his files. The mere existence of encryption may throw up red flags for a would be attacker.

Furthermore, an attacker may resort to torture or blackmail to get a user to reveal his keys. What is needed is a mechanism that allows “plausible deniability”[1] or the ability of the user to claim that such data does not exist. Anderson et al [1] were the first to propose a file system based on the idea of steganography. They listed a number of reasons that one may want to be able to provide “plausible deniability, including:

- Soldiers and intelligence agents may be captured and tortured into revealing cryptographic keys and other secret data.
- When conducting delicate negotiations, such as between a company and a trade union, informal offers may be made which will be denied in the event of

later litigation. However, the other side might obtain court orders for access to documents.

- Police power may be abused. An individual may be arrested on `suspicion' of a crime, found with an encrypted hard disk partition, and told that if he does not supply the password this will be taken as evidence of guilt. But the encrypted files might well contain confidential business information sought by people who have bribed the police.
- Private individuals have been tortured by robbers into revealing information such as the secret codes for their bank cards and the location of safes.

However the two solutions that they proposed had numerous problems of performance and reliability. StegFS[9] created a practical implementation based on the second solution and suffered from many of the same problems. Fortunately Pang et al [11] came up with an independent solution<sup>1</sup> which solves nearly all of the problems related to StegFS and provides a very reliable and useful implementation. Unfortunately it is based only on a local disk and therefore does not take into account issues such as traffic analysis and update analysis which may occur in a distributed system. These issues were accounted for in a later paper [15].

I am proposing a system that incorporates many of the ideas of [11 and [15], as well as some from the cryptographic filesystems[2,4,13,14], that is able to provide steganographic protection on any remote storage system, as long as the user has read/write access to a directory. The server that is being utilized does not have to be altered in any way. The client will be responsible for proper access to the server and for proper construction of a hierarchical namespace. This hierarchical namespace will only be visible to the client and will vary depending upon the secure levels[1] that are being accessed. Secure levels are such, that each level reveals different hidden data. This will provide plausible deniability in the case that a user can reveal a lower secure level to an attacker if under duress, while still maintaining the secrecy of his most important files.

## **2. Related Work**

In this section I discuss the three major designs for steganographic filesystems that are publicly available. I will explain the benefits and drawbacks of each one and work to achieve a base that XSFS can be built upon.

### **2.1 The Steganographic File System (Anderson et al.)**

In [1], Anderson et al describe two file systems employing steganography. They also present many reasons for which a person may want to employ such a system, as stated in the previous section. The first of their “ constructions assumes only that there are limits to the amount of knowledge of the plaintext that the opponent possesses, and to the number of computations that he can perform. In the second, [they] will assume the existence of a block cipher that the opponent cannot break. [1]“

The first scheme is based upon the notion of embedding hidden user files in the exclusive-or of a subset of random cover files. The subset of chosen files is based upon

---

<sup>1</sup> It is also entitled StegFS, so it will further be referred to as StegFS2.

the user's password. Multiple passwords may be used, and intermixed with dummy passwords in any number of ways as to provide plausible deniability. With a sufficient number of cover files, it is computationally infeasible to determine the plaintext without any previous knowledge of what the plaintext contains. However, as Anderson et al admit, this is a weakness in the system as it can be broken fairly trivially with a known plaintext attack. Furthermore the read-write overhead involved is directly proportional to the number of cover files in the system, such that if 16 cover files are used to encode a plaintext, then all 16 must be read or written on each access. Using more cover files may make the system more secure, but also increases its latency.

Additionally, traffic and update analysis are not taken into account. If an attacker could monitor all updates to the system as well as all reads, he could easily determine which cover files, and how many, concealed each hidden file. Thus this system is only applicable to local disks, not distributed systems.

The second scheme is more sophisticated and relies on cryptographic hashing of passwords and directory structures in order to determine which block to write to on a completely randomized disk. Furthermore hidden blocks are encrypted so as to make them indistinguishable from random blocks.

The problem with this approach is due to collisions. If a block is written to at a lower security level which is currently holding data from a higher security level, the block will appear empty and will be overwritten. In order to get around this problem, redundancy is used. However, as they admit in [1], there is still a high probability of data corruption. By reducing the probability of data corruption, the amount of disk that can be used is also reduced[1]. This scheme is also only applicable to a local disk, as traffic and update analysis are not accounted for.

## **2.2 StegFS (McDonald and Kuhn)**

McDonald and Kuhn have implemented a steganographic filesystem based upon the second scheme proposed by Anderson et al[1]. The Linux based implementation includes full filesystem semantics and sits between the VFS and buffer cache[9]. It is fully compatible with current ext2fs[3] filesystems, and instead of needing a separate partition it only requires unused space on an ext2fs filesystem. Instead of using hashing of passwords to find block locations, it uses a cryptographically sealed bitmap. Furthermore inodes contain an additional field containing the number of replicas, and references to replicas of all equivalent inodes and data blocks.

The number of different security levels is hard coded at 15. This may provide plausible deniability in that a user can claim he only used one or two levels, but the rest were created automatically. However it could lead to further convince an attacker that the 15<sup>th</sup> level must be opened, and lead to unnecessary scenarios(torture etc...). This is a problem with a fixed security level scheme.

This program also suffers from the data loss problem of Anderson et al's second scheme. Higher level data may be overwritten by lower level data or even normal data from the ext2fs filesystem. On top of this, this is only a local filesystem scheme.

## **2.3 StegFS2 (Pang et al)**

Pang et al[11] overcame the data loss problem with the novel design of StegFS2. StegFS2 partitions storage space into fixed size blocks and uses the underlying filesystem's bitmap to keep track of which files are used and which are free. Initially random patterns fill all the unused blocks. Then a set of random blocks are "abandoned" by registering them as used in the bitmap so as to limit a trace of the system from figuring out which files are hidden based on the difference between marked blocks that are not part of the regular filesystem, and used filesystem blocks that are not encrypted. Furthermore StegFS2 employs the use of dummy files that are composed of dummy blocks which serve to mislead an attacker as to what is actually hidden.

In order for the hidden data to be located, there must be a quick way to find it since it is not registered with the central directory. StegFS2 allocates hidden files by supplying "a hash value computed from the file name and access key as seed to a pseudo random block number generator, and [checking] each successive generated block number against the bitmap until the file system finds a free block to store the header.[11]" The header contains all file information including, a link to an inode table, a unique signature and a list of free blocks held by the file. Other blocks of the file are allocated by randomly selecting free blocks from the bitmap, writing them, and storing that information in the header or inode file. Finding the data is performed by once again utilizing the pseudo random block number generator and looking for a matching header signature.

StegFS2 maintains two types of keys to secure the files on its system. It maintains File Access Keys (FAK) which individually encrypt each file on the system, and User Access Keys(UAK) which are used as encryption keys for access levels. Each UAK which is directly known by the user, encrypts a hidden file that contains a directory of all of the (filename, FAK) pairs for that access level. This promotes file sharing between users in that, one user can extract the FAK with his UAK and give the FAK to another user through some other manner such as PGP encrypted email. The user who just acquired the FAK could then add this to the proper access level file and encrypt it with his UAK. One problem with this scheme is that sharing of keys is at the file level, and if multiple users want to share a lot of files, they need to distribute a lot of keys, which complicates key management and can lead to compromise. However, since each key only encrypts one file, the amount of compromise is limited. Another problem with this scheme is that it assumes that these files are accessible to all users who share the FAK. And in general this may not be the case.

StegFS2[11] is implemented as a file system driver for Linux which sits between the VFS and buffer cache, as in StegFS[9]. It only works on local filesystems, and thus fails to provide full protection against update analysis and traffic analysis. A later paper by the same authors discusses these issues and will be presented in the next subsection.

#### **2.4 Protection against update and traffic analysis**

Zhou et al [15] describe several methods that can be implemented to prevent attacks on steganographic file systems through update and traffic analysis. These serve to further protect the usefulness of such a product in distributed systems. The main idea is to issue dummy updates that are indistinguishable from data updates such that traffic

patterns cannot be mapped. Because dummy updates are issued to random locations, updates of actual data may occur regularly and stand out statistically. To remove this threat the system changes the location of data blocks at random times[15].

There are two constructions of this idea in the paper. The first relies on a trusted non-volatile agent through which all I/O operations are performed, while the second relies on a volatile agent. The non-volatile agent contains a full view of the file system and hence provides a single point of attack. It also completely destroys the purpose of a steganographic filesystem, which is to provide multiple access routes and allow plausible deniability. It is of no practical importance and will not be considered further.

The second construction, in which a volatile agent is utilized is much more practical and powerful. Each owner has his own encryption key (UAK) and dummy files which are of the average length of data files. The encryption key is given to the agent only on login. Theoretically (and practically) each user could (and should) have many UAKs for each secure level. As more users login, or more secure levels are accessed, the agent sees a larger view of the hidden filesystem and is able to carry out dummy updates on the sum of the users' dummy blocks.

Reads are more complicated, according to [15], as they require a lot of buffer reorganization, in what they call "oblivious storage." Oblivious storage is a separate remote partition which basically caches all reads and occasionally re-encrypts and shuffles them so that subsequent reads appear random. However, this seems overly complicated and unnecessary as reads may be cached at the client instead. And upon each read a lazy update may be performed, such that the read block is written back to the server in a random location at a later time (before the user ends his session.) Furthermore, even if the client runs out of cache space, the time in which the read would remain in the cache would be sufficient to allow many dummy reads and updates so that the read block would not be identifiable anyway. This is the approach that XSFS will take.

### **3. XSFS Design**

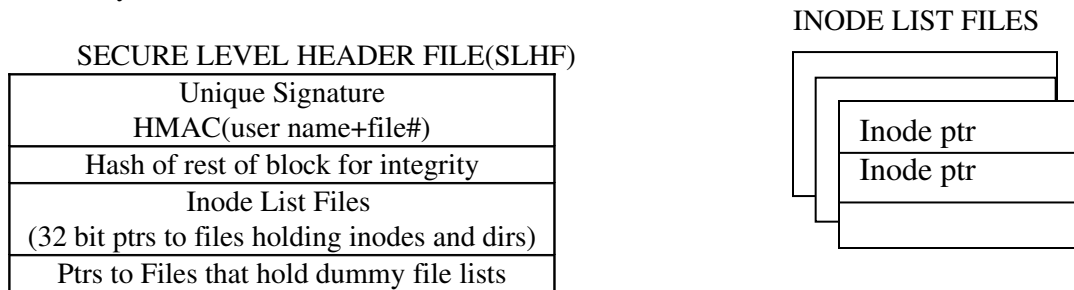
XSFS takes many of its design principles from StegFS2[11] and its extension to the distributed arena[15]. However it differs mainly in that it is able to operate without modification of the remote storage that it is utilizing for the steganographic filesystem, as well as in the fact that it is designed specifically for the distributed case. Furthermore the system does not utilize an agent, as I feel that allowing an agent to trust multiple users can lead to an unsafe condition, bearing in mind that the client in which XSFS is running must be trusted. Thus only one user can use XSFS on a client machine at one time. It is primarily meant to be used on personal computers, laptops, and workstations, not on a shared system.

#### **3.1 Layout of XSFS on the server and initialization procedure**

XSFS differs from every other steganographic filesystem in that it does not modify the server. The obvious question becomes: How is the data storage of the server utilized and organized without being modified? The answer is a relatively simple extension of StegFS2[11].

StegFS2 has a header block corresponding to each file which contains a unique signature, an inode table, and a list of free blocks. The inodes in StegFS2 correspond to physical block locations on disk. XSFS differs in that the inodes correspond to files that are of the equivalent size of the blocks on the disk. Furthermore there is only one header file for each secure level. It contains pointers to files containing lists of pointers to inodes of all the files and directories in the toplevel directory of the secure level. It also contains a unique signature ( a keyed MAC based on the user name and file # where the header is stored), a hash of the block to maintain integrity, and pointers to several files containing dummy file lists(see figure 1). All of this is cryptographically sealed (symmetric block cipher in CBC mode[2,4,11,13,14)with the key to the secure level.This is in contrast to there being a header for each hidden file in StegFS2[11]. It also differs in that each file does not have its own encryption key. The following discussion of disk initialization should help clarify these ideas. First the user must have a home directory on a remote disk. The steps are as follows:

- 1) The size of the directory quota is determined automagically or input by the user
- 2) The user determines the number of blocks in the directory (dir size/ block size)
- 3) The # of blocks needed to hold the free/used block bitmap is determined  
(#blocks for bitmap = #blocks in dir / block size)
- 4) A file is created equal to this size and is named 0 ( it will contain the bitmap)
- 5) The rest of the directory is partitioned into files equal to the block size( or best estimate for efficiency) until the user's space quota is used up, and the files are filled with random data. The name of each file is its consecutive number 1..n where n is equal to the number of files in the directory.



**Figure 1**

The system is now ready to begin creating a secure level and adding the first hidden file.

### 3.2 Creating a secure level

A user creates a secure level by calling the *CreateSecureLevel* command. The user is prompted for a passphrase. Upon entering this passphrase it is cryptographically hashed to become the user's key for the secure level, as in Cryptfs[14]. The XSFS client performs a read on the used file bitmap on the server and retrieves it. The user marks off randomly anywhere from 10-100,000 unused files in the bitmap as used and records all except one of them as dummy files. The client automatically determines how many files are needed to hold the lists of the dummy files, creates them locally and inserts the randomly selected pointers to the dummy files into the files created to hold the lists. The client then creates the Secure Level Header File (SLHF) and inserts the pointers to the

dummy lists into it, by randomly selecting which recently marked files in the bitmap will be used for valid files, and which will be dummy lists.

The user is then prompted to add a file or directory to the system. If the user selects NULL, the SLHF is signed and encrypted with the user key for the secure level and written to disk along with all of the dummy files and encrypted dummy file lists. Otherwise, the user selects a file or directory to add, and the XSFS client determines how much space will be needed to hold all the files. It then allocates inode list files to hold pointers to all of the inodes and directories of the files in the secure level. It then randomly marks off enough space in the used file bitmap and records these locations in the inode list files. Pointers to the inode list files are then inserted into the SLHF and all files except the dummy files are encrypted. The inodes themselves are then modified to point to the correct files on the server which will hold the data corresponding data blocks from the local filesystem. The inodes, directories, and data blocks of the original filesystem are encrypted along with the inode list files, the dummy list files, and the SLHF and written to the correct files on the remote disk. The dummy files are also written to the proper locations, along with the updated bitmap. All writing is randomly arranged to prevent traffic analysis. Upon write completion, the directories and files added to the secure level are deleted from the hard drive of the user. Note also, that each file written, except for the SLHF, contains a cryptographic hash in the first 16 bytes of the file, in order to maintain integrity. This hash is in the second 16 bytes of the SLHF. Figure 2 will illustrate the procedure of creating a secure level.

## CREATING A SECURE LEVEL

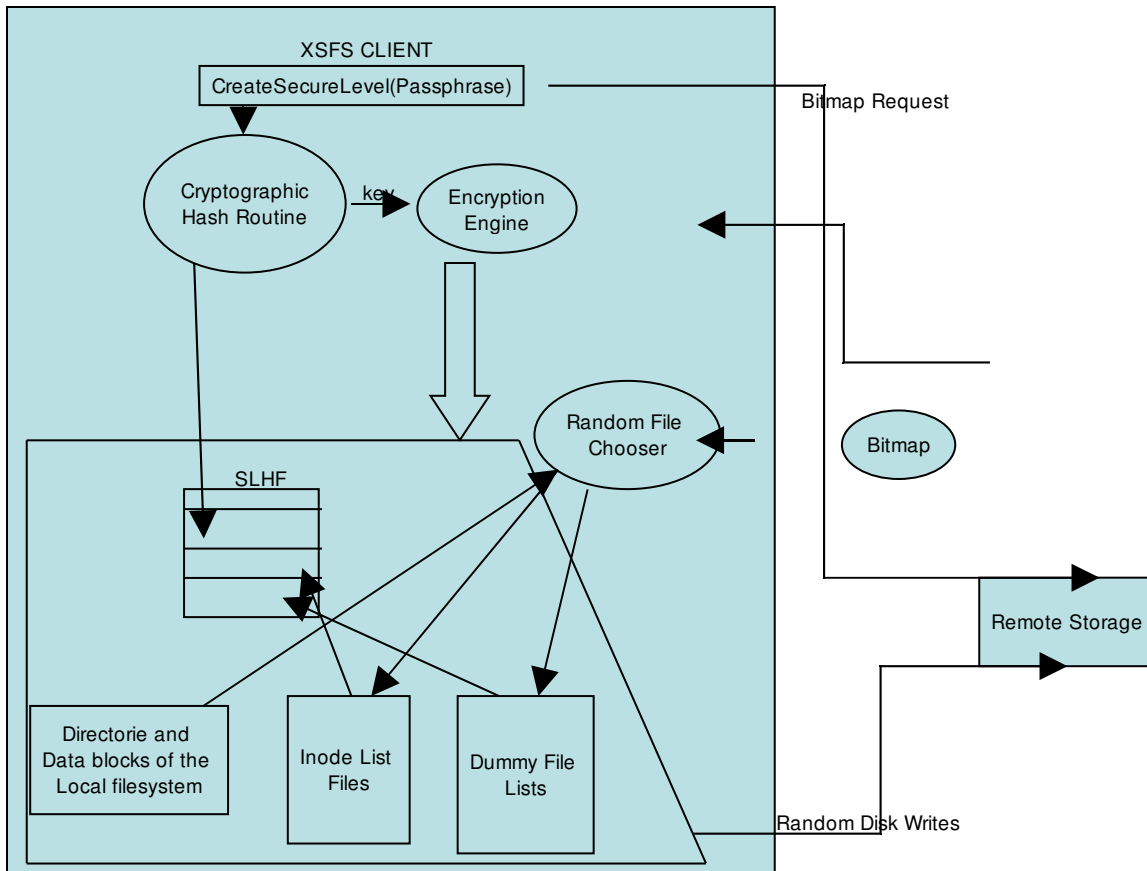


Figure 2

### 3.3 Opening a secure level

To open a secure level the user enters his passphrase and a key is generated using a cryptographic hash, such as MD5 or SHA-1. Note that keys are never stored on any non-volatile storage and are created dynamically from the user given passphrase, as was done in Cryptfs[14]. This is a very secure method of storage, as mind reading is the only computationally feasible way to determine the keys.

In order to open the secure level, the user must be able to quickly find the SLHF. This is achieved by inputting the key into a pseudorandom file finder which is similar to the pseudorandom block number generator for file headers in StegFS2[11]. Each time a file is located through the file finder it is retrieved and its signature is compared to the one computed from the secure level key on the XSFS client. If they match, we know that we have found the correct SLHF. Otherwise we find the next file with the pseudorandom file finder and repeat until we have a match. Dummy reads are mixed in, so that a pattern cannot be found. However, we must note that entering an invalid passphrase can cause the file finder to run indefinitely, so XSFS utilizes a timeout method, where the timeout is user configurable.

When a match is found, we have successfully opened the secure level. Reading, writing and updating are next.

### **3.4 Reading, Writing, and Updating**

For true steganography, the data access pattern should match the dummy access pattern[11]. Since the dummy access pattern is random for simplicity[11], the data access pattern should be random also. All files considered now will be block size.

Writing a file is similar to the procedure outlined in [15] and starts by encrypting the file data on the client with the secure level key using the remote file number as the initialization vector (IV). Then the client reads the used/free file bitmap from the server and chooses a file at random that is owned by the secure level or free. If the file chosen is free, it is written out and marked used. If the file is not free the inode lists are searched to see if it is a hidden file. If it is hidden, it is read out and the new file is written in its place. If it is a dummy file, the dummy file is read out and discarded and the new file is written in its place. The write procedure repeats in the case of a hidden file selection with the read out hidden file. Note that the reason the dummy file is read out is so that statistical analysis cannot be performed that will allow an attacker to determine which overwritten previously used files were written without being read, and then determine that the file was a dummy access. After each access the correct file numbers are recorded in the proper inodes or dummy file lists. Furthermore, if too many data files write over dummy files such that the number of dummy files drops below a configurable threshold, then new dummy files will be created[11]. Dummy writes are issued at random to conceal traffic patterns. Furthermore, we must note that dummy files must be re-encrypted on dummy writes, with the file number being written to used as the IV[15]. This is necessary to disallow matching of ciphertexts and possible tracing of dummy blocks.

Updating proceeds in the same manner as writes, except that it has to read a specific block, before writing it back to disk in a random location. The problem here is that reading is not random. We counter this traffic analysis problem by issuing dummy reads on any block in between valid reads. When the read file is updated it uses the write procedure above. The old file is placed on the dummy list for the secure level or mark it free. This is done randomly.

Reading proceeds exactly as in updating, except that the data being written out remains unchanged. We note that we must write out the data as above(although it may be done lazily) so that regular reads do not occur at the same location, and present a statistical read anomaly. We also note, that even though these reads do not change their data, they encrypt to different values, because they reside on different files, and the file number is the IV of the encryption routines. This does incur a write overhead with every read which is significant, but needed for security. Lazy updates can reduce this overhead.

Files that are larger than block size are broken into block size chunks for the read, write, and update procedures. Furthermore, note that since the inode structures and directory structures are unmodified except for substituting remote file numbers for local block numbers, these routines can readily be implemented in a kernel level filesystem, and reconstruction of the hidden filesystem is a snap.

## **4. Security of XSFS**

XSFS is secure against traffic and update analysis as well as cryptanalytic attacks. However, if all security levels are not used with some regularity, the system actually becomes less secure, in that it reveals that at least one other security level exists. This is counterintuitive, but has a simple rationale.

Note that for every update and write, the files that can be written to are only those which are free or which are owned by the secure level. If only one secure level is utilized, there will be a number of files which are not written. These will stand out as a hidden security level. If all security levels are used on a regular basis though, the traffic patterns will be such that it will be impossible to determine what is used and what isn't, in any given level.

This analysis brings up a few points worth mentioning. First of all, by not using a security level, the data in that security level is inherently more secure from a cryptographic standpoint, as it isn't vulnerable in the clear on a possibly exploited client machine. But on the other hand, if the client machine is greatly trusted, the hidden security levels should be hard to determine by utilizing the most important levels frequently. The tradeoff is basically a tradeoff between who the attacker might be. If the attacker is likely to be a network hacker exploiting the client, then it is probably better off using his most important data as little as possible. He may know you have at least one more security level, but does not know how much data is there, or what is contained in it, and has no method of getting you to reveal it. However, if the attacker has control over the server and your client machine is extremely trustworthy, you are better off using all of your secure levels frequently, so that you can plausibly deny that more security levels do not exist if coerced physically or legally[1].

## **5. Performance and Storage Space Estimates**

XSFS is not a performance rich application, although steganography by definition incurs overhead. XSFS uses significantly more space (up to twice as much) as StegFS2[11], because it does not modify the server filesystem. This occurs because XSFS partitions the filespace into many block size files. Each block size file in the underlying filesystem is already pointed to by another block size file, namely the server filesystem inode. Thus at most only half of the storage space can be used for the XSFS filesystem. However, this is equivalent to the case of having many small files on an unencrypted filesystem. Furthermore, file block size must be estimated, and if it is smaller than an actual underlying file system block, than data space will definitely be wasted. Although this should be easy to predict and configure if the filesystem on the remote disk is known ahead of time. Storage space for XSFS further degrades, because it must utilize space for header files, dummy files, dummy file lists, inode file lists, and inodes. StegFS2 has similar space needs.

StegFS[8] has a significantly large overhead because it relies on replicas to prevent data loss. It may be equivalent or even exceed the overhead in XSFS, although this is hard to gauge as XSFS is not implemented. However XSFS does benefit from the fact that there is no inherent data loss, while data loss cannot be completely eliminated in StegFS. XSFS is also far more efficient and secure than Anderson et al's first scheme[1].

XSFS incurs at least one write on every read of the filesystem as well as multiple dummy reads, rights and updates in between valid calls. This increases the overhead computationally, although I believe with organized caching and lazy writes, performance can be increased dramatically.

## 6. Ideas for future research

XSFS is designed to work on any data storage site that a user has read/write access to. I have discussed how XSFS can be applied to one specific remote data store. However, a user may be able to enter the network address and login information of multiple data storage sites, such as Google gmail etc..., so that he can utilize free internet space for multiple secure levels. XSFS may even be implemented to allow storing part of a secure level on one site and part on another to provide even more obfuscation and further restrict the ability to track data accesses.

Additionally, XSFS should be able to handle multiple network protocols for transferring encrypted files, such as NFS, AFS, FTP, etc... These should be able to be added as external modules for use in communication between the XSFS client and remote site. This is an idea taken from the x-kernel[5].

Another area which has not been discussed in this paper is backups and reliability issues in the case of a remote server or client crash while working. This is a complicated issue and was not in the scope of this paper, but should be considered in any practical design.

Currently the only way for XSFS to share data between users is for each user to be able to log into the same remote store with the same passphrase for accessing a security level. This implies that these users share the same access rights on the XSFS directory on the remote store. These users may not work on the same XSFS store at the same time however as there is no concurrency control or locking scheme and the bitmaps and hidden files could (and probably would) be extremely damaged and not reproducible. There is probably a better way to promote group sharing and it should be investigated.

## 7. Conclusion

I have shown how a steganographic file system can be constructed on any remote storage system without modification, where a user has read/write access. I have also determined that the storage overhead of such a system would be significant, although with the amount of free space being provided for free today such as Google's gmail, this should not limit the use of XSFS. I feel that XSFS could be implemented in a fairly straightforward manner, although reliability and backup issues as well as a plugin structure for protocols, and group sharing would need to be worked out. The steganographic ability of XSFS as well as the ability for it to immediately utilize the majority of remote storage could lead to an effective implementation being well utilized by many people across the globe.

## REFERENCES

[1] Ross Anderson, Roger Needham and Adi Shamir. The Steganographic File System. In *Proc. Second International Information Hiding Workshop*, Portland, Oregon, USA, April 1998.

- [2] M. Blaze, "A cryptographic file system for UNIX." 1<sup>st</sup> ACM Conference on Computer and Communications Security, 9--16 (November 1993)
- [3] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the 1<sup>st</sup> Dutch International Symposium on Linux*. ISBN 90-367-0385-9, 1995.
- [4] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano. The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 245--252, June 2001.
- [5] Norman C. Hutchinson, Larry L. Peterson, The X-Kernel: An Architecture for Implementing Network Protocols, IEEE Transactions on Software Engineering, v.17 n.1, p.64-76, January 1991
- [6] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 29--42, March 2003.
- [7] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. *Secure untrusted data repository (SUNDR)*. Technical Report TR2003-841, NYU Department of Computer Science, June 2003.
- [8] D. Mazieres et al. Separating Key Management from File System Security. In *17<sup>th</sup> Symp. on Operating Systems Principles (SOSP'99)*, Kiawah Island, SC, 1999. ACM.
- [9] Andrew D. McDonald and Markus G. Kuhn. StegFS: A Steganographic File System for Linux. *Proc. Third International Workshop on Information Hiding*, Dresden, Germany, October 1999.
- [10] E. L. Miller, W. E. Freeman, D. D. E. Long, and B.C. Reed. Strong security for network-attached storage. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 1-14, Jan. 2002.
- [11] H. Pang, K. Tan, and X.Zhou. Stegfs: A steganographic file system. In *Proceedings of the 19<sup>th</sup> International Conference on Data Engineering*, pages 657-668, Bagalore, India, March 2003.
- [12] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *Proceedings of the 1<sup>st</sup> USENIX Symposium on File and Storage Technologies (FAST '02)*, pages 14--29, Monterey, California, January 2002.
- [13] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the Annual USENIX Technical Conference*, pages 197--210, June 2003.
- [14] E. Zadok, I. Badulescu, and A. Shender. *Cryptfs: A Stackable Vnode Level Encryption File System*. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 1998.

[15] Xuan Zhou, HweeHwa Pang, Kian-Lee Tan, "Hiding Data Accesses in Steganographic File System", *Proceedings of the IEEE International Conference on Data Engineering*, Boston, USA, March 2004, 572-583.