StegFS: A Steganographic File System

HweeHwa PANG[1] Kian-Lee TAN[2] Xuan ZHOU[2]

[1]Laboratories for Information Technology
21 Heng Mui Keng Terrace
Singapore 119613
hhpang@lit.org.sg

[2]Department of Computer Science
National University of Singapore
3 Science Dr 2, Singapore 117543
{tankl, zhouxuan}@comp.nus.edu.sg

Abstract

While user access control and encryption can protect valuable data from passive observers, those techniques leave visible ciphertexts that are likely to alert an active adversary to the existence of the data, who can then compel an authorized user to disclose it. This paper introduces StegFS, a steganographic file system that aims to overcome that weakness by offering plausible deniability to owners of protected files. StegFS securely hides user-selected files in a file system so that, without the corresponding access keys, an attacker would not be able to deduce their existence, even if the attacker is thoroughly familiar with the implementation of the file system and has gained full access to it. Unlike previous steganographic schemes, our construction satisfies the prerequisites of a practical file system in ensuring integrity of the files and maintaining efficient space utilization. We have completed an implementation on Linux, and experiment results confirm that StegFS achieves an order of magnitude improvements in performance and/or space utilization over the existing schemes.

1. Introduction

User access control and encryption are standard data protection mechanisms in current file system products, such as the Encrypting File System (EFS) in Microsoft Windows 2000 and XP. These mechanisms enable an administrator to limit user access to a given file or directory, as well as the specific types of actions allowed. However, access control and encryption can be inadequate where highly valuable data is concerned. Specifically, an encrypted file in a directory listing or an encrypted disk volume is itself evidence of the existence of valuable data; this evidence could prompt an attacker to attempt to circumvent the protection or, worse, coerce an authorized user into unlocking it. An administrator may also intentionally or inadvertently grant access permission to other users in contradiction to the wishes of the owner, for example by simply adding users to a protected file's access control list, or to the group that the owner gives access permission to.

In order to protect data against such security threats, we would like to have a file system that grants access to a protected directory/file only if the correct password or access key is supplied. Without it, an adversary could get no information about whether the protected directory/file ever exists, even if the adversary understands the hardware and software of the file system completely, and is able to scour through its data structures and the content on the raw disks. Thus a user acting under compulsion would be able to plausibly deny the existence of hidden information; he can disclose only less sensitive files, e.g. his address book, but remain silent on valuable content like budget data, and the adversary would not know that the user has withheld information. Unauthorized users and even the administrators would also be unable to gain access to or sabotage the data. Steganography, the art of hiding information in ways that prevent its detection, offers a way to achieve the desired protection. It is a better defense than cryptography alone – While cryptography scrambles a message so it cannot be understood, steganography goes a step further in making the ciphertext invisible to unauthorized users.

There have been a number of proposals for steganographic file systems in recent years, e.g. [7, 13]. In order to support the steganographic property, those proposals have had to make a number of design decisions that compromise the practicality of the file systems, resulting in large increases in I/O operations, low effective storage space utilizations, and even risk of data loss as hidden files could get overwritten. With such compromises, it is unlikely that the proposed schemes could move beyond niche applications

into mass-market commercial file systems that are expected to manage large volumes of data reliably and efficiently.

In this paper, we introduce StegFS, a scheme to implement a steganographic file system that enables users to selectively hide their directories and files so that an adversary would not be able to deduce their existence. To ensure its practicality, StegFS is designed to meet three key requirements – it should not lose data or corrupt files, it should offer plausible deniability to owners of protected directories/files, and it should minimize any processing and space overheads. StegFS excludes hidden directories and files from the central directory of the file system. Instead, the metadata of a hidden directory/file object is stored in a header within the object itself. The entire object, including header and data, is encrypted to make it indistinguishable from unused blocks to an observer. Only an authorized user with the correct access key can compute the location of the header, and access the directory/file through the header. We have implemented StegFS on the Linux operating system, and extensive experiments confirm that StegFS indeed produces an order of magnitude improvements in performance and/or space utilization over the existing schemes.

The remainder of this paper is organized as follows: Section 2 summarizes related work, including classical approaches to steganography in general and proposals for steganographic file system in particular. Our StegFS file system is introduced in Section 3, which also contains a discussion on some potential limitations of StegFS and ways to work around them. Following that, Section 4 presents our StegFS implementation on the Linux operating system, and Section 5 profiles StegFS's performance characteristics. Finally, Section 6 concludes the paper and discusses future work.

2 Related Work

Current operating systems allow users to specify access policies for their directories and files. For example, a Unix user can set read, write and execute permissions for the owner, users in the same group, and other users, while Windows 2000 allows a directory owner to specify read or modify permissions for a list of users. These access control mechanisms can be extended by or complemented with file encryption. Encrypted file system products include the Encrypting File System (EFS) in Windows 2000/XP [3] that encrypts selected files within a folder using password- or public key-based techniques, and E4M [2] and PGPDisk [4] that maintain separate encrypted disk volumes, among others. While access control and encryption can safeguard the content of protected folders, an unauthorized observer can still establish their existence and coerce the owner(s) into unlocking them.

Steganography provides a countermeasure against this vulnerability, by preventing an attacker from verifying whether a user acting under compulsion actually discloses all of the data. Derived from a Greek word that literally means "covered writing", steganography is about concealing the *existence* of messages and encompasses a wide range of methods like invisible ink, microdots, covert channels and character arrangement. This contrasts with cryptography, which is about concealing the *content* of messages. While the practice of steganography dates back many centuries, the modern scientific formulation was first given in [17]. Since then, many studies have investigated ways of embedding a secret message, be it an electronic watermark, a covert communication or a serial number, within still images [12], text [9], audio [18] and video [11].

The classical approaches to steganography are concerned with embedding relatively small messages within large cover texts, e.g. using the least significant bit of the pixels in an image to hide copyright information. While some products apply these approaches directly to secure data files, e.g. DriveCrypt [1] is capable of hiding entire disk volumes in music files, the resulting overhead in storage space is unacceptable for a general-purpose file system that needs to hold large volumes of data with high space usage efficiency.

In [7], Anderson et al proposed two schemes for implementing steganographic file systems. Both schemes allow a user to associate a password with a file or directory object, such that requests for the object will be granted only if accompanied by the correct password. An attacker who does not have the matching object name and password, and lacks the computational power to guess them, cannot deduce from the raw disk data whether the named object even exists in the file system. The first scheme initializes the file system with a number of randomly generated cover files. When a new object is deposited, it is embedded as the exclusive-or of a subset of the cover files, where the subset is a function of the associated password. Compared to the classical steganography techniques, this scheme entails a lower space overhead as it can accommodate as many objects as there are cover files. However, the performance penalty is very high as every file read or write translates into I/O operations on multiple cover files.

In contrast, the second scheme in [7] writes the blocks of a hidden file to absolute disk addresses given by some pseudo-random process. An implementation based on the second scheme was reported in [13]. The problem with this scheme is that different files could map to the same disk addresses, thus causing data loss. While the risk can be controlled by replicating the hidden files and by limiting the loading factor, it cannot be eliminated completely. In [10], Hand and Roscoe extended the scheme to provide better resilience on a peer-to-peer platform, by replacing simple replication with the information dispersal algorithm (IDA) [15]. Using IDA, a file owner chooses two numbers $m \geq n$

Parameter	Meaning	Default
$Abandon\%$	Percentage of abandoned blocks in the disk volume	1%
$FreeBlk_{min}$	Minimum number of free blocks within a hidden file	0
$FreeBlk_{max}$	Maximum number of free blocks within a hidden file	10
$DumNum$	Number of dummy hidden files in the file system	10
$DumSize$	Average size of the dummy hidden files	1 MB

Table 1. Parameters of StegFS

and encodes the hidden file into m cipher-files such that any n of them suffice to reconstruct the hidden file. However, this is achieved at the expense of higher storage and read/write overheads, and there is still the possibility of data loss when more than $(m - n)$ cipher-files get corrupted.

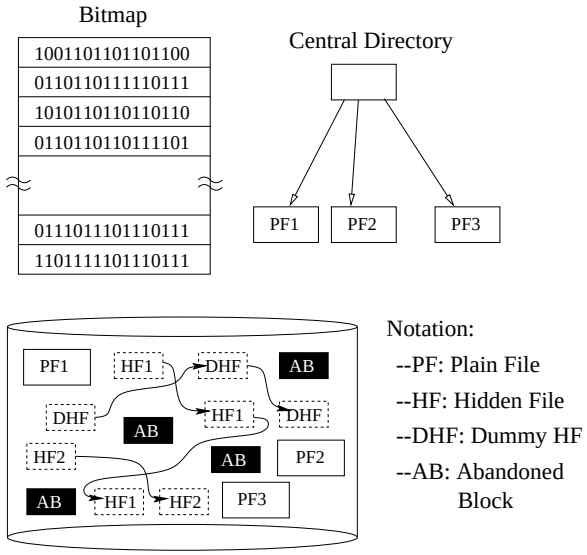


Figure 1. Overview of the StegFS File System

3 StegFS: Steganographic File System

In this section, we present StegFS, a practical scheme for implementing a general-purpose steganographic file system. Our scheme is designed to satisfy three key objectives: (a) StegFS should not lose data or corrupt files. (b) StegFS should hide the existence of protected directories and files from users who do not possess the corresponding access keys, even if the users are thoroughly familiar with the implementation of the file system. (c) StegFS should minimize any processing and space overheads.

To hide the existence of a directory/file, it should be excluded from the central directory of the file system. Instead, StegFS maintains the hidden directory/file object's structure, eg. its inode table, in a header within the object itself. Similarly, all records pertaining to the object, for example usage statistics, should also be isolated within the object instead of being written to common log files. The entire object, including header and data, is encrypted to make it indistinguishable from unused blocks in the file system to an unauthorized observer. Only a user with the access key is able to locate the file header and, from there, the hidden directory/file. The parameters of StegFS, which will be explained below, are listed in Table 1. To simplify the description, we will henceforth focus on hidden files, with the understanding that the discussion applies equally to hidden directories.

3.1 File System Construction

Figure 1 gives an overview of the StegFS file system. The storage space is partitioned into standard-size blocks, and a bitmap tracks whether each block is free or has been allocated – a 0 bit indicates that the corresponding block is free, while a 1 bit signifies a used block. All the plain files are accessed through the central directory, which is modeled after the inode table in Unix. Hidden files are not registered with the central directory, though the blocks occupied by them are marked off in the bitmap to prevent the space from being re-allocated.

When the file system is created, randomly generated patterns are written into all the blocks so that used blocks do not stand out from the free blocks. Furthermore, some randomly selected blocks are abandoned by turning on their corresponding bits in the bitmap. These abandoned blocks are intended to foil any attempt to locate hidden data by looking for blocks that are marked in the bitmap as having been assigned, yet are not listed in the central directory. The higher the number of abandoned blocks, the harder it is to succeed with such a brute-force examination for hidden data. However, this has to be balanced with space utilization considerations. In practice, the number of abandoned blocks may be determined by an administrator, or set randomly by StegFS.

StegFS additionally maintains one or more dummy hidden files that it updates periodically. This serves to prevent an observer from deducing that blocks allocated between successive snapshots of the bitmap that do not belong to any plain files must hold hidden data. The number of dummy hidden files can also be set manually or automatically. Note that dummy files do not eliminate the need for abandoned blocks – whereas dummy files are maintained by StegFS and could be vulnerable to an attacker with administrator privileges, abandoned blocks are untraceable and hence of-

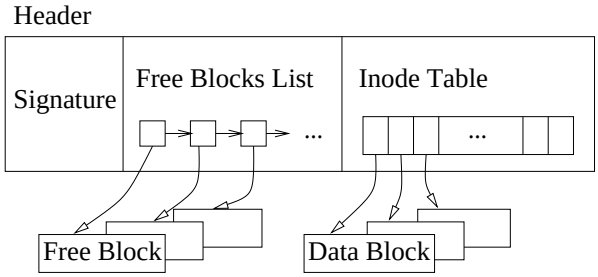


Figure 2. Structure of Hidden File

fer extra protection.

In the example in Figure 1, the file system contains two hidden user files, a dummy hidden file and three plain files, each of which comprises one or more disk blocks. There are also abandoned blocks scattered across the disk.

The structure of a hidden file is shown in Figure 2. Each hidden file is accessed through its own header, which contains three data structures:

- A link to an inode table that indexes all the data blocks in the file.
- A signature that uniquely identifies the file.
- A linked list of pointers to free blocks held by the file.

All the components of the file, including header and data, are encrypted with an access key to make them indistinguishable from the abandoned blocks and dummy hidden files to unauthorized observers.

Since the hidden file is not recorded in the central directory, StegFS must be able to locate the file header using only the (physical) file name and access key. During file creation, StegFS supplies a hash value computed from the file name and access key as seed to a pseudorandom block number generator, and checks each successive generated block number against the bitmap until the file system finds a free block to store the header. Once the header is allocated, subsequent blocks for the file can be assigned randomly from any free space by consulting the bitmap, and linked into the file's inode table. To prevent overwriting due to different users issuing the same file name and access key, the physical file name is derived by concatenating the user id with the complete path name of the file.

To retrieve the hidden file, StegFS once again inputs the hash value computed from the file name and access key as seed to the pseudorandom block number generator, and looks for the first block number that is marked as assigned in the bitmap and contains a matching file signature. The initial block numbers given by the generator may not hold the correct file header because they were unavailable when the file was created. Thus the signature, created by hashing the file name with the access key, is crucial for confirming that the correct file header has been located. To avoid false matches, the file signature has to be a long string. A one-way hash function is used to generate the signature so that an attacker cannot infer the access key from the file name and the signature. Examples of such hash functions include SHA [6] and MD5 [16].

Another characteristic of a hidden file is that it may hold on to free blocks. Here the intention is to deter any intruder who starts to monitor the file system right after it is created, and hence is able to eliminate the abandoned blocks from consideration, then continues to take snapshots frequently enough to track block allocations in between updates to the dummy hidden files. Such an intruder would probably be able to isolate some of the blocks that are assigned to hidden files. By maintaining an internal pool of free blocks within a hidden file, StegFS prevents the intruder from distinguishing blocks that contain useful data from the free blocks. When a hidden file is created, StegFS straightaway allocates several blocks to the file. These blocks, tracked through a linked list of pointers in the file header, are selected randomly from the free space in the file system so as to increase the difficulty in identifying the blocks belonging to the file and the order between them. As the file is extended, blocks are taken off the linked list randomly for storing data or inodes until the number of free blocks falls below a preset lower bound, at which time the internal pool is topped up. Conversely, when the file is truncated, the freed blocks are added to the internal pool until it exceeds an upper bound, wherein some of the free blocks are returned to the file system.

3.2 Directory Support for File Sharing

While StegFS incorporates several features to safeguard files that are hidden by a user, it is most effective in a multi-user environment. This is because when many blocks are allocated for hidden files, an attacker may be able to estimate the amount of useful data in these files, but there is no way to ascertain just how much of that belong to any particular user. Hence a user acting under coercion is likely to have a lot of leeway in denying the existence of valuable data that is accessible by him.

One of the natural requirements of a multi-user system is the sharing of hidden files among users. As a user may want to share only selected files, StegFS secures each hidden file with a randomly generated file access key (FAK) rather than the user's access key, so that the file name and FAK pair can be shared among multiple users.

Figure 3 depicts the directory structure that StegFS implements to help users track their hidden files. StegFS allows a user to own several user access keys (UAK). For each UAK, StegFS maintains a directory of file name and FAK

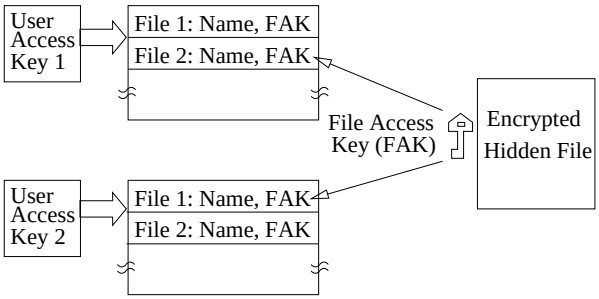


Figure 3. Directory Structure of StegFS

pairs for all the hidden files that are accessed with that UAK. The entire directory is encrypted with the UAK and stored as a hidden file on the file system. The UAKs could be managed independently, for example stored in separate smart cards for maximum security. Alternatively, to make the file system more user-friendly, UAKs belonging to a user could be organized into a linear access hierarchy such that when the user signs on at a given access level, all the hidden files associated with UAKs at that access level or lower are visible. Thus, under compulsion, the user could selectively disclose only a subset of his UAKs. Without knowing how many UAKs the user owns, the attacker would not be able to deduce that the user is holding back on some of his UAKs.

To share a hidden file with another user, the owner has to release its file name and FAK pair to the recipient. Since neither the owner nor StegFS has the UAK of the recipient, the sharing cannot be effected automatically. Instead, the file information is encrypted with the recipient's public key, and the ciphertext is sent to the recipient, for example via email. Using a StegFS utility, the recipient then decrypts the ciphertext with his private key and associates the hidden file with his own UAK, at which time the file information is added to the UAK's directory and the ciphertext is destroyed. The practice of transmitting the file information is a relatively weak point in StegFS, as the ciphertext could alert an attacker to the existence of the hidden file. However, as each hidden file has its own FAK, a compromised ciphertext does not expose other hidden files in StegFS. The file sharing mechanism is summarized in Figure 4.

Finally, when the owner of a hidden file decides to revoke the sharing arrangement, StegFS first makes a new copy with a fresh FAK and possibly a different file name, then removes the original file to invalidate the old FAK. The outdated FAK will be deleted from the directories of other users the next time they log in with their UAKs.

3.3 File System Backup and Recovery

Since the hidden files in StegFS are shielded even from the system administrator, the usual method of backing up a

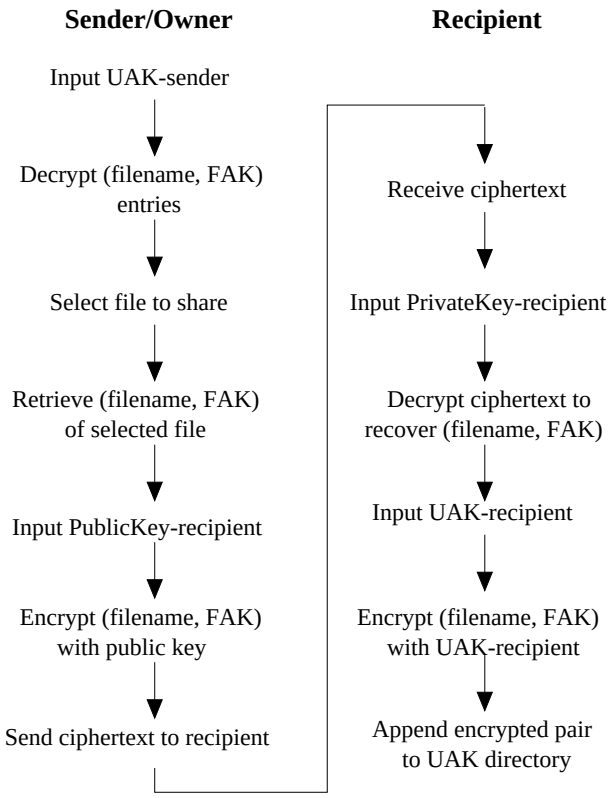


Figure 4. File Sharing in StegFS

file by copying its content no longer works for them. Yet a brute force approach of saving the image of the entire file system would be too time-consuming, in view of the ever-growing capacity of modern storage devices.

StegFS saves the image of only those blocks that are allocated in the bitmap but do not belong to any plain file in the central directory. Plain files are still backed up by copying their content. This limits the overhead of StegFS to the space that is occupied by abandoned blocks, dummy hidden files, and free blocks held within the user hidden files.

To recover a damaged file system, StegFS first restores the image of the abandoned and hidden blocks to their original addresses. This is necessary because the hidden files contain their own inode tables that cannot be adjusted by the recovery process to reflect new block assignments. The plain files are reconstructed last, possibly at new addresses.

3.4 Potential Limitations of StegFS

While StegFS offers an extra feature over a "vanilla" file system in hiding the existence of protected files, this is achieved at the expense of introducing a number of limitations:

- All the hidden files must be restored together; it is not possible to roll-back hidden files selectively. A work-

around is to restore all the hidden files to a temporary volume, from where the user can copy the required files over to the permanent StegFS volume.

- The file system is unable to defragment hidden files to improve their retrieval efficiency, without cooperation from the users who possess the file access keys. This is a common problem among secure file system products. A solution is to offer users the option of depositing a copy of the FAKs with the system administrator, and to adopt measures to minimize the likelihood that the administrator account is compromised.

- The file system cannot remove hidden files belonging to expired user accounts without cooperation from the users who possess the file access keys. Again, this limitation is common for secure file system products, and can be addressed by keeping a copy of the FAKs with the administrator.

4 System Implementation

We have implemented StegFS on the Linux kernel 2.4. Figure 5, adapted from [13], shows the system architecture. It is implemented as a file system driver between the virtual file system (VFS) and the buffer cache in the Linux kernel, alongside other file system drivers like Ext2fs [8] and Minix [19]. StegFS implements all the standard file system APIs, such as open() and read(), so it is able to support existing applications that operate only on plain files. In addition, StegFS introduces several steganographic file system APIs for creating hidden directories/files, converting between hidden and plain directories/files, revealing hidden directories/files, and sharing hidden directories/files. Details of these APIs are as follows.

1. **void steg_create(char* objname, char* UAK, char objtype)**

   This function creates a hidden file or directory. It uses SHA256 [6] as the pseudorandom number generator for locating the hidden object (the seed is recursively hashed to generate the pseudorandom numbers), and the block cipher for encrypting data blocks is based on AES [5]. The parameters are: (a) *objname*: name of the hidden object to be created; (b) *UAK*: user access key for the hidden object; and (c) *objtype*: type of the hidden object ('f'— regular file, 'd'— directory).

2. **void steg_hide(char* pathname, char* objname, char* UAK)**

   This function converts a plain file or directory into a hidden object. The plain source object is deleted upon completion. The parameters are: (a) *pathname*: path name of the plain source file/directory to be converted; (b) *objname*: name of the target hidden object; and (c) *UAK*: user access key for the hidden object.

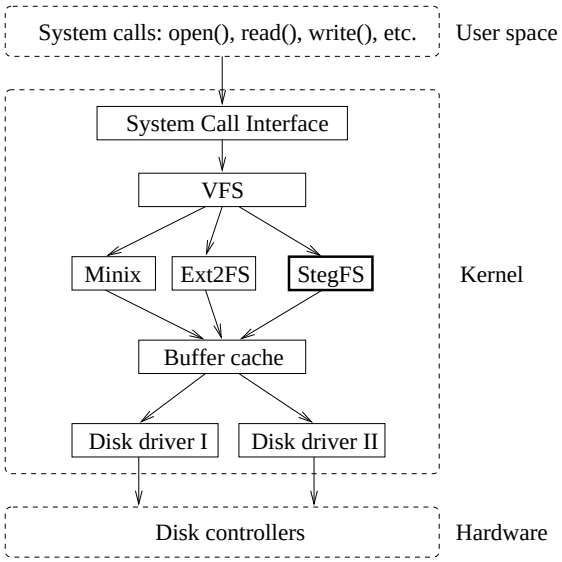


Figure 5. StegFS Implementation

3. **void steg_unhide(char* pathname, char* objname, char* UAK)**

   This function converts a hidden file or directory into a plain object. The hidden source object is deleted upon completion. The parameters are: (a) *pathname*: path name of the target plain object; (b) *objname*: name of the hidden source file/directory to be converted; and (c) *UAK*: user access key for the hidden object.

4. **void steg_connect(char* objname, char* UAK)**

   This function connects a hidden object to the current user session. It first locates the hidden object through the (objname, UAK) pair, then adds an entry to the current working directory to make the hidden object visible. Connecting a hidden directory reveals all its offsprings as well. Data blocks of the hidden object are not decrypted by this function; they are decrypted on-the-fly during retrieval. The parameters are: (a) *objname*: name of the hidden object to be connected; and (b) *UAK*: user access key for the hidden object.

5. **void steg_disconnect(char* objname)**

   This function disconnects a hidden object from the current user session, so that the object becomes invisible again. When the user logs off, all the connected hidden objects are automatically disconnected. The parameter is *objname*, the name of the hidden object to be disconnected.

6. **void steg_getentry(char* objname, char* entryfile, char* publickey)**

To share a hidden object with another user, the owner calls *steg_getentry* to store the particulars of the shared object in entryfile. The entryfile, encrypted by the recipient's public key, is then sent to the recipient who in turn calls *steg_addentry* to add the object particulars to his own directory. The parameters are: (a) *objname*: path name of the object to be shared with another user; (b) *entryfile*: path name of the file into which to write the encrypted object particulars; and (c) *publickey*: public key of the recipient for encrypting entryfile.

7. **void steg_addentry(char* objname, char* entryfile, char* privatekey)**

   This function decrypts and adds the object particulars in entryfile to the destination directory. The parameters are: (a) *objname*: path name of the destination directory; (b) *entryfile*: path name of the file that stores the encrypted object particulars; and (c) *privatekey*: private key for decrypting entryfile.

8. **void steg_backup(char* fsdevice, char* backupfile)**

   To backup the file system, the system administrator calls *steg_backup* to store the current snapshot of the file system into backupfile. The parameters are: (a) *fsdevice*: path name of the device on which the file system is mounted; and (b) *backupfile*: path name of the file that stores the backup image.

9. **void steg_recovery(char* fsdevice, char* backupfile)**

   This function recovers the file system by restoring from the backup image in backupfile. The parameters are: (a) *fsdevice*: path name of the device on which the file system is mounted; and (b) *backupfile*: path name of the file that stores the backup image.

5 Performance Evaluation

Having presented the design and implementation of our steganographic file system, we are now ready to investigate its efficacy relative to the alternative schemes. This section begins with a description of the experiment set-up, then proceeds to present results from some of the more interesting experiments.

5.1 Experiment Set-Up

To evaluate the performance of StegFS, we ran a series of experiments with various workloads on an Intel PC. The key parameters of the hardware are listed in Table 2, while Table 3 summarizes the workload parameters. Note in particular that we expect many file servers to use a block size of 1 KBytes – the allocation unit is 1 KBytes in NTFS, and 512 Bytes or 1 KBytes in Unix – hence we set that as the default. However, we will also experiment with larger block sizes to study how StegFS would perform with other file systems (the allocation units in FAT16 and FAT32 are 32 KBytes and 8 KBytes, respectively).

Parameter	Value
Model of the CPU	Intel Pentium 4
Clock speed of the CPU	1.6 GHz
Type of the hard disk	Ultra ATA/100
Capacity of the hard disk	20 GB

Table 2. Physical Resource Parameters

Parameter	Default
Size of each disk block	1 KBytes
Size of each file	(1, 2] MBytes
Capacity of the disk volume	1 GBytes
Number of files in the file system	100
File access pattern	Interleaved
Number of concurrent users	1

Table 3. Workload Parameters

Parameter	Meaning
$StegFS$	Our proposed StegFS scheme
$StegCover$	Steganographic scheme using cover files in [7]
$StegRand$	Steganographic scheme using random block assignment in [7]
$CleanDisk$	Freshly defragmented Linux file system
$FragDisk$	Well-used Linux file system with fragmentation

Table 4. Algorithm Indicators

For comparison purposes, we shall benchmark against the native file system in Linux and the two schemes proposed in [7] – *StegCover* that hides each file among 16 cover files as recommended by the authors, and *StegRand* that writes a hidden file to absolute disk addresses given by a pseudorandom process and replicates the file to reduce data loss from overwritten blocks (see Section 2). As for the native Linux file system, its performance provides an upper bound to what any file protection scheme can achieve at best; we shall examine two separate cases – *CleanDisk* and *FragDisk*. With CleanDisk, files are loaded onto a freshly formatted disk volume and occupy contiguous blocks; this is intended to highlight the best possible performance limit. In contrast, FragDisk reflects a well-

used disk volume where files are fragmented, and is simulated by breaking each file into fragments of 8 blocks.

The primary performance metrics for the experiments are: (a) the effective space utilization, i.e., the aggregate size of the unique data files divided by the capacity of the disk volume; and (b) the file access time, defined as the time taken to read or write a file, averaged over 1000 observations. We exclude from consideration the cost of decrypting hidden files after they are retrieved, which is insignificant relative to I/O costs. For example, a 2 MBytes file can be decrypted in less than 120 ms on our test system, whereas the I/Os take at least 2 seconds depending on the block size.

5.2 Effective Space Utilization

We begin our investigation with an experiment to profile the space utilization of the steganographic file systems. Here the size of the disk volume is set to 1 GBytes, the block size ranges from 512 Bytes to 64 KBytes, while the file sizes vary uniformly between 1 and 2 MBytes.

Let us first examine the StegCover scheme. Since the cover files must be big enough to accommodate the largest data file, the most efficient space utilization is achieved by setting the cover files to 2 MBytes. With file sizes in the range of (1, 2] MBytes, each set of cover files can be 50% to 100% utilized, thus giving an average space utilization of 75%. While we can probably improved upon the original StegCover scheme by packing several files into each set of cover files, and by letting large files span multiple sets of cover files, that would introduce indexing complexities and performance penalties, and is beyond the scope of our work.

Turning our attention to StegRand, we note that its resilience against data corruption can be improved by file replication. For each replication factor in the range of 1 and 64, we load the data files one at a time until all copies of any data block of a file are overwritten – that is when StegRand has just passed the limit where it can safely recover all its hidden files, and beyond which more files will be corrupted and lost permanently. At that point, we sum up the size of the loaded files and divide it by the disk volume size to derive the effective space utilization. Each file is counted only once in this process regardless of the number of replicates. The results are given in Figure 6. As expected, the space utilization rises initially as increasing replication factors bring about greater resilience against data loss. However, beyond the window of 8 to 16, higher replication factors lower the space utilization due to the dominant effect of replication overheads. Furthermore, smaller block sizes produce lower space utilizations. This is because block corruptions occur more frequently in a disk volume made up of many small blocks than one with fewer large blocks.

Finally, we consider the StegFS scheme. Here, the only storage overheads are incurred by the abandoned blocks,

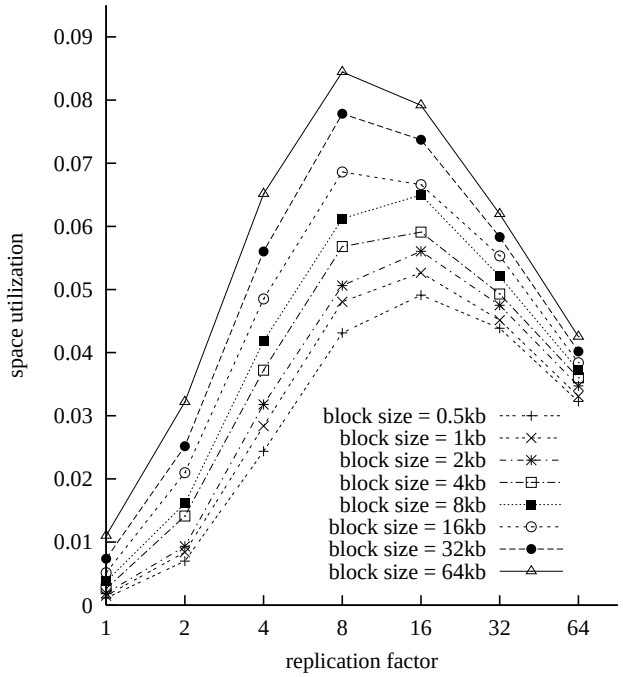


Figure 6. StegRand Space Utilization

the dummy hidden files, the inode structures, and the free blocks held within the hidden files. Since there is no danger of data blocks being overwritten, all of the remaining space can be used for useful data. With the default settings in Table 1, StegFS is able to consistently achieve more than 80% space utilization.

To summarize, we have arrived at a couple of observations in this experiment. First, the StegCover scheme cannot achieve full space utilization without extending it to perform file packing and spanning. Second, StegRand works reliably only when the disk volume is very sparsely populated; file servers that are typically formatted with a 1 KByte block size can achieve only 5% space utilization for a 1 GByte volume, and less for larger disks, before data corruption sets in. This result is consistent with the findings in [7]. Third, the proposed StegFS is capable of achieving higher space utilizations than StegCover, and is at least 10 times more space-efficient than StegRand.

5.3 Performance Analysis

Having demonstrated StegFS's superior space utilization, we now focus on its performance characteristics. This experiment is intended to study how well it works, relative to the native file system and the other steganographic schemes, on file servers where I/O operations from several users or applications are interleaved. For StegCover, the number of cover files is 16, while a replication factor of 4

is used for StegRand, both according to the authors' recommendation in [7]. The disk volume size and the block size are set to 1 GBytes and 1 KBytes, respectively, while the file sizes vary uniformly between 1 and 2 MBytes.

Figures 7(a) and (b) give the read and write access times, respectively, for the various file systems. Since StegCover spreads each hidden file among multiple cover files, every file operation translates to several disk I/Os, hence its read and write access times are very much worse than the rest. As for StegRand, its read performance is worse than StegFS's due to the need to hunt for an intact replicate when the primary copy of a file is found to be corrupted, whereas the write access times are much worse because all the replicates must be updated.

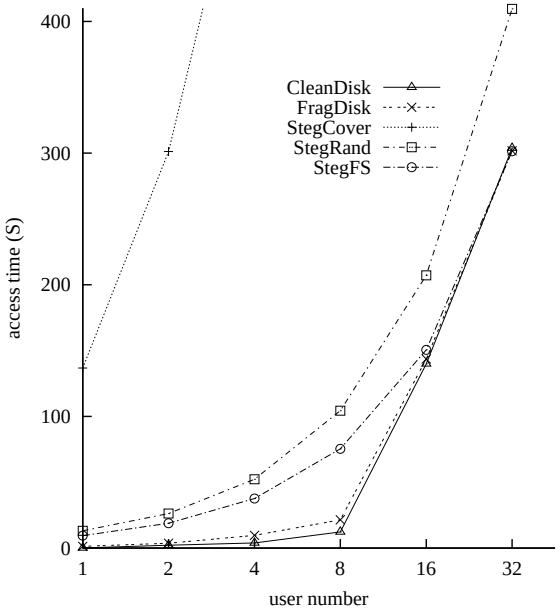
As for StegFS, its access times are slower than those of CleanDisk and FragDisk under very light load conditions as they produce sequential I/Os on contiguous data blocks, particularly for read operations that benefit from the read-ahead feature of the hard disk. However, the differentiation diminishes with increased workload, as file operations become increasingly interleaved. In fact, StegFS matches both CleanDisk and FragDisk from 16 concurrent users onwards for read operations, and from just 8 users for write operations. Finally, the relative trade-offs between the various schemes are independent of the file size, as shown in Figures 8(a) and (b).

In summary, this experiment demonstrates that both of the previous steganographic schemes introduce very high read and/or write penalties and are not suitable for file servers that must handle heavy loads. In contrast, StegFS is a practical steganographic file system that delivers similar performance to the native Linux file system in a multi-user environment.

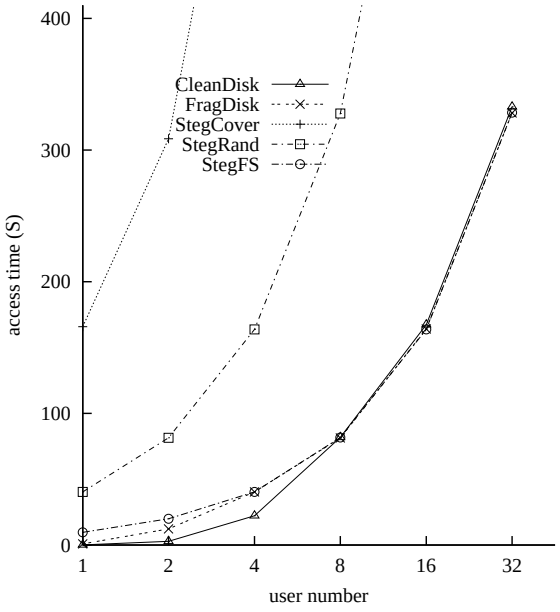
5.4 Sensitivity to File Access Patterns

The next experiment is aimed at discovering the sensitivity of the various file systems' performance to the file access pattern. Specifically, we are looking at a situation where each file is retrieved in its entirety before the next file is opened, as may happen in a very lightly loaded file server. Besides the number of concurrent users which is fixed at 1, the other workload parameters remain as in the previous experiment.

Figures 9(a) and (b) show the read and write access times for the various file systems, with the file size fixed at 1 MBytes. Here, CleanDisk delivers the best performance as expected since all its files occupy contiguous blocks. FragDisk, which breaks each file into fragments of 8 blocks, is slower due to the extra overhead in seeking to each fragment. This indicates that as the file system gets more fragmented, its performance would gradually degrade to that of StegFS even in single-user environments where file opera-

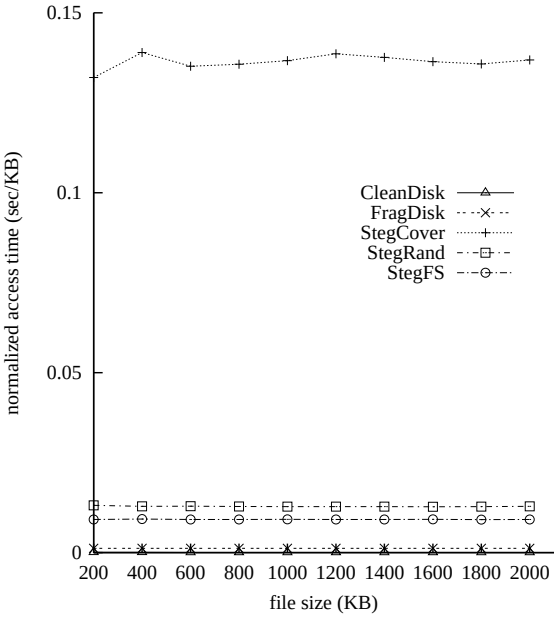


(a) Read

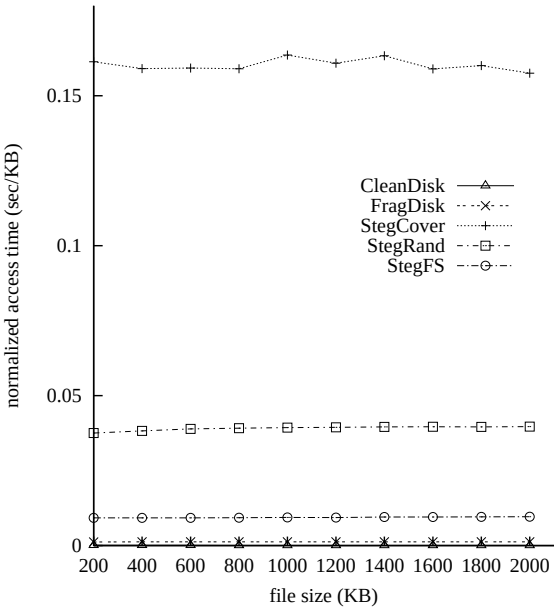


(b) Write

Figure 7. Multiple Concurrent Users

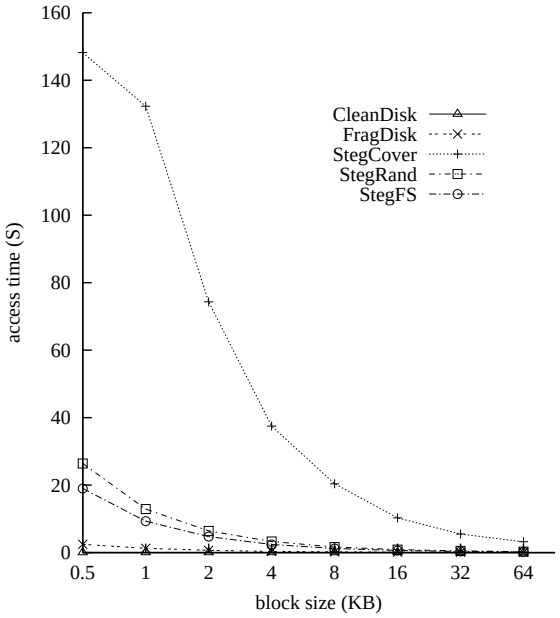


(a) Read

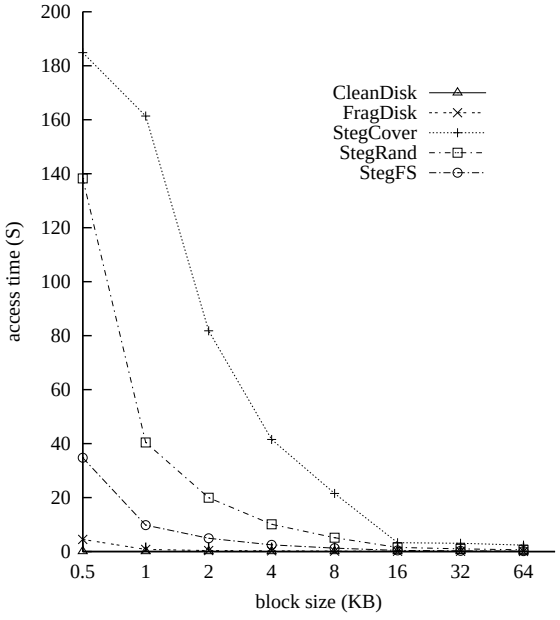


(b) Write

Figure 8. Sensitivity to File Size



(a) Read



(b) Write

Figure 9. Serial File Operations

tions are not interleaved. The difference in performance is more pronounced with small block sizes where FragDisk has to perform more fragment seeks, and StegFS and StegRand incur more block seeks.

This experiment demonstrates that while StegFS achieves similar performance to the Linux file system in a multi-user environment, the penalty that StegFS incurs in hiding data files is noticeable when the load is so light that file I/Os are not interleaved. Even then, StegFS still delivers acceptable access times and outperforms the previous steganographic schemes significantly.

6 Conclusion

In this paper, we introduce StegFS, a practical scheme to implement a steganographic file system that offers plausible deniability to owners of protected files. StegFS securely hides user-selected files in a file system so that, without the corresponding access keys, an attacker would not be able to deduce their existence, even if the attacker understands the hardware and software of the file system completely, and is able to scour through its data structures and the content on the raw disks. Thus a user acting under compulsion would be able to plausibly deny the existence of hidden information. StegFS achieves this steganographic property while ensuring the integrity of the files, and maintaining efficient space utilization at the same time. StegFS excludes hidden directory and file objects from the central directory of the file system. Instead, the metadata of a hidden object is stored in a header within the object itself. The entire object, including header and data, is encrypted to make it indistinguishable from unused blocks to an observer. Only an authorized user with the correct access key can compute the location of the header, and access the hidden directory/file through the header.

We have implemented StegFS as a file system driver in the Linux kernel 2.4; the code is available for public download at http://xena1.ddns.comp.nus.edu.sg/SecureDBMS/. Extensive experiments on the system confirm that StegFS is capable of achieving an order of magnitude improvements in performance and/or space utilization over the existing steganographic schemes. In fact, StegFS is just as fast in a multi-user environment as the native Linux file system, which is the best that any file protection scheme can aim for.

For future work, we are extending the techniques in StegFS to DBMS. Specifically, we are investigating how database tables, hash indices and B-trees can be hidden effectively while preserving the DBMS' ability to control concurrency and recover data. We are also looking for better ways to overcome the limitations described in Section 3.4. Building a P2P-based StegFS as an application on top of BestPeer [14] is also on our agenda.

References

[1] Drivecrypt secure hard disk encryption. http://www.drivecrypt.com.

[2] E4m disk encryption. http://www.e4m.net.

[3] Encrypting file system (efs) for windows 2000. http://www.microsoft.com/windows2000/techinfo/howit works/security/encrypt.asp.

[4] Pgpdisk. http://www.pgpi.org/products/pgpdisk/.

[5] *Advanced Encryption Standard*. National Institute of Science and Technology. FIPS 197, 2001.

[6] *Secure Hashing Algorithm*. National Institute of Science and Technology. FIPS 180-2, 2001.

[7] R. Anderson, R. Needham, and A. Shamir. The steganographic file system. In *Information Hiding, 2nd International Workshop*, D. Aucsmith, Ed., Portland, Oregon, USA, April 1998.

[8] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings of the 1st Dutch International Symposium on Linux*. ISBN 90-367-0385-9, 1995.

[9] M. Chapman and G. Davida. *Information and Communications Security – First International Conference*. ISBN 3-540-63696-X, November 1997.

[10] S. Hand and T. Roscoe. Mnemosyne: Peer-to-peer steganographic storage. In *Electronic Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, March 2002. http://www.cs.rice.edu/Conferences/IPTPS02/.

[11] F. Hartung, J. Su, and B. Girod. Digital watermarking for compressed video. In J. Dittman, P. Wohlmacher, P. Horster, and R. Steinmetz, editors, *Multimedia and Security – Workshop at ACM Multimedia '98*, volume 41 of GMD Report, United Kingdom, September 1998.

[12] N. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. In *Computer, 31(2):26-34*, February 1998.

[13] A. McDonald and M. Kuhn. Stegfs: A steganographic file system for linux. In *Proceedings of the Workshop on Information Hiding, IHW'99*, Dresden, Germany, September 1999.

[14] W. Ng, B. Ooi, and K. Tan. Bestpeer: A self-configurable peer-to-peer system. In *Proceedings of the 18th International Conference on Data Engineering*, San Jose, CA, April 2002. (Poster Paper).

[15] M. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. In *Journal of the ACM*, volume 36, No. 2, pages 335–348, April 1989.

[16] R. Rivest. *RFC 1321: The MD5 Message-Digest Algorithm*. Internet Activities Board, 1992.

[17] G. Simmons. The prisoners' problem and the subliminal channel. In *Proceedings of the CRYPTO '83*, pages 51–67. Plenum Press, 1984.

[18] M. Swanson, B. Zhu, and A. Tewfik. Audio watermarking and data embedding – current state of the art, challenges and future directions. In J. Dittman, P. Wohlmacher, P. Horster, and R. Steinmetz, editors, *Multimedia and Security – Workshop at ACM Multimedia '98*, volume 41 of GMD Report, Bristol, United Kingdom, September 1998.

[19] A. Tanenbaum and A. Woodhul. *Operating Systems: Design and Implementation, 2nd Edition*. Prentice Hall, 1997.