Steganographic Schemes for File System and B-Tree

HweeHwa Pang, Kian-Lee Tan, *Member*, *IEEE Computer Society*, and Xuan Zhou

Abstract—While user access control and encryption can protect valuable data from passive observers, these techniques leave visible ciphertexts that are likely to alert an active adversary to the existence of the data. This paper introduces StegFD, a steganographic file driver that 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. Unlike other steganographic schemes proposed previously, our construction satisfies the prerequisites of a practical file system in ensuring the integrity of the files and maintaining efficient space utilization. We also propose two schemes for implementing steganographic B-trees within a StegFD volume. We have completed an implementation on Linux, and results of the experiment confirm that StegFD achieves an order of magnitude improvements in performance and/or space utilization over the existing schemes.

Index Terms—Steganography, plausible deniability, security, access control, StegFD, StegBtree.

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 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 [7], [13]. To support the steganographic property, these 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 the file system itself could write over hidden files. 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 StegFD, 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, StegFD 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. StegFD 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 StegFD on the

- *H.H. Pang is with the Institute for Infocomm Research, 21 Heng Mui Keng Terrace, Singapore 119613. E-mail: hhpang@i2r.a-star.edu.sg.*
- *K.-L. Tan and X. Zhou are with the Department of Computer Science, National University of Singapore, 3 Science Drive 2, Singapore 117543. E-mail: {tankl, zhouxuan}@comp.nus.edu.sg.*

Manuscript received 1 April 2003; revised 29 Aug. 2003; accepted 6 Jan. 2004. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0023-0403.

Linux operating system, and extensive experiments confirm that StegFD indeed produces an order of magnitude improvements in performance and/or space utilization over the existing schemes.

A preliminary version of this paper appears in [15]. (We have renamed our steganographic file system to StegFD to avoid confusion with the StegFS in [13].) There, we presented only StegFD. We have extended the paper to address how B-trees can be supported within a StegFD volume. We introduce two schemes for implementing steganographic B-trees and also report a performance study to evaluate the proposed B-tree 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 a steganographic file system, in particular. Our StegFD file system is introduced in Section 3, together with a discussion on some potential limitations of StegFD and ways to work around them. Section 4 presents our StegFD implementation on the Linux operating system, and profiles StegFD's performance characteristics. In Section 5, we present extensions to StegFD to support B-trees. 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 [18]. 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 [19], 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. Since each cover file can be used repeatedly by various hidden objects, the system can actually 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 pseudorandom 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) [16]. Using IDA, a file owner chooses two numbers $m \geq n$ 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.

3 StegFD: Steganographic File Driver

In this section, we present StegFD, a practical scheme for implementing a general-purpose steganographic file system. Our scheme is designed to satisfy three key objectives:

1. StegFD should not lose data or corrupt files.
2. StegFD 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.
3. StegFD 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, StegFD maintains the hidden directory/file object's

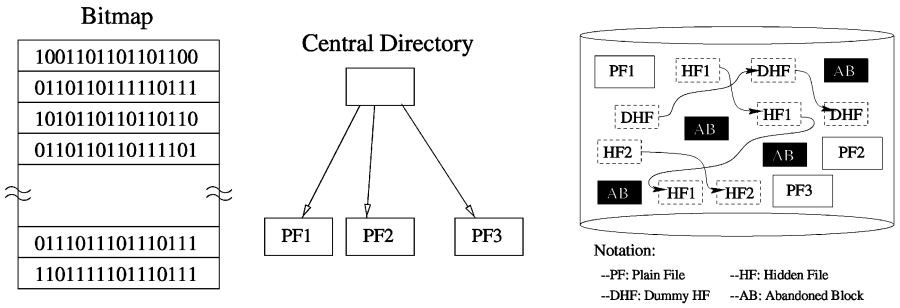


Fig. 1. Overview of the StegFD file system.

structure, e.g., 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. 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

Fig. 1 gives an overview of the StegFD 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 reallocated.

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 StegFD.

StegFD 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 StegFD and could be vulnerable to an attacker with administrator privileges, abandoned blocks offer extra protection because they cannot be traced.

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

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

1. a link to an inode table that indexes all the data blocks in the file,
2. a signature that uniquely identifies the file, and
3. 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, StegFD must be able to locate the file header using only the (physical) file name and access key. During file creation, StegFD 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, StegFD 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

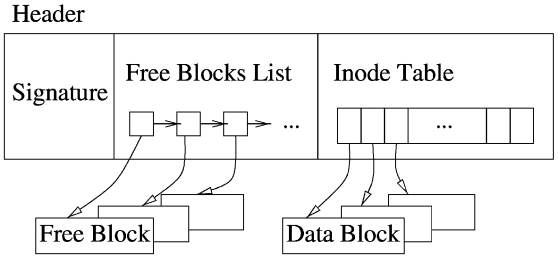


Fig. 2. Structure of a hidden file.

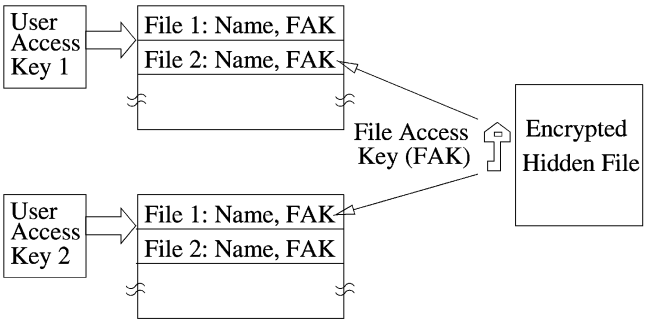


Fig. 3. Directory structure of StegFD.

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 [17].

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, StegFD prevents the intruder from distinguishing blocks that contain useful data from the free blocks. When a hidden file is created, StegFD 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 StegFD incorporates several features to safeguard files that are hidden by a user, it is most effective in a multiuser 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 belongs 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 multiuser system is the sharing of hidden files among users. As a user may want to share only selected files, StegFD secures each

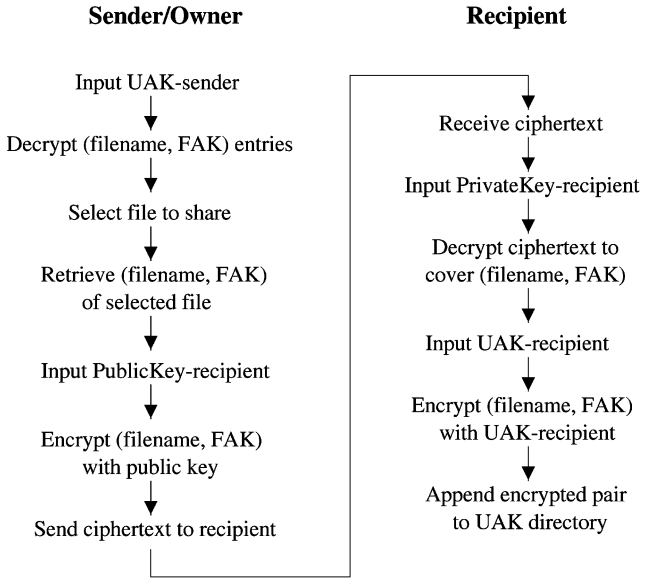


Fig. 4. File sharing in StegFD.

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.

Fig. 3 depicts the directory structure that StegFD implements to help users track their hidden files. StegFD allows a user to own several user access keys (UAK). For each UAK, StegFD maintains a directory of file name and FAK 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 some 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 StegFD 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 resulting ciphertext is sent to the recipient, for example, via email. Using a StegFD 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 StegFD, 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 StegFD. The file sharing mechanism is summarized in Fig. 4.

Finally, when the owner of a hidden file decides to revoke the sharing arrangement, StegFD 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 StegFD are shielded from even the system administrator, the usual method of backing up a 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.

StegFD 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 StegFD 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, StegFD 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 block addresses.

Many existing file systems provide data recovery tools to fix accidental errors. For example, if the file header is lost or corrupted, a regular file system can always track the lost chains and recover the lost file. StegFD can also support recovery by introducing some redundancy: The header of a hidden file can be replicated and placed in pseudorandom locations derived from its FAK. Thus, if the file header is corrupted, the replica can be retrieved to recover the hidden file. Additionally, a signature can be inserted in each data block, so that, if necessary, a hidden file can be recovered by scanning the disk volume for blocks with matching signatures.

3.4 Potential Limitations of StegFD

While StegFD 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 workaround is to restore all the hidden files to a temporary volume, from where the user can copy the required files over to the permanent StegFD 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 employ a key recovery mechanism (e.g., [21]) that allows a user to deposit a copy of his UAK with several managers through a secret sharing scheme. To reconstruct the UAK subsequently, concurrence of some minimum number of those managers is needed, thus ensuring the security of the UAK.
- 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 a key recovery mechanism.

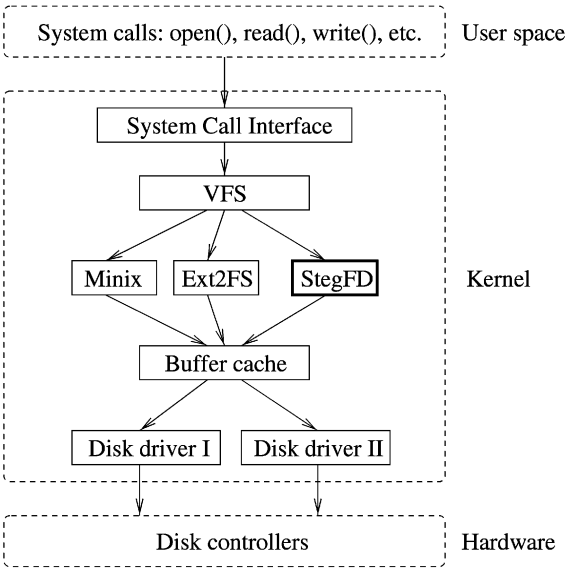


Fig. 5. StegFD implementation.

4 System Implementation and Performance Evaluation

This section begins with a description of an implementation of StegFD, then proceeds to present results from some of the more interesting experiments.

4.1 System Implementation

We have implemented StegFD on the Linux kernel 2.4; the code is available for public download at the StegFD Web site (http://xena1.ddns.comp.nus.edu.sg/SecureDBMS/). We have used 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]. Fig. 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 [20]. StegFD 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, StegFD 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 the API can also be found at the StegFD Web site.

TABLE 1
Physical Resource Parameters

Parameter	Value
Model of the CPU	Intel Pentium 4
Clock speed of the CPU	1.6 GHz
Type of the hard disk	IBM ATA/IDE
Capacity of the hard disk	60 GB

TABLE 2
Workload Parameters

Parameter	Default
Size of each disk block	1 KBytes
Size of each file	(1, 2] MBytes
Capacity of the disk volume	25 GBytes
Number of files in the file system	2000
File access pattern	Interleaved
Number of concurrent users	1

4.2 Experiment Set-Up

To evaluate the performance of StegFD, we ran a series of experiments with various workloads on an Intel PC. The key parameters of the hardware are listed in Table 1, while Table 2 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 StegFD would perform with other file systems (the allocation units in FAT16 and FAT32 are 32 KBytes and 8 KBytes, respectively).

For comparison purposes, we shall benchmark against the native file system in Linux and the two schemes proposed in [7]—*StegCover* 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 eight blocks.

The primary performance metrics for the experiments are:

1. the effective space utilization, i.e., the aggregate size of the unique data files divided by the capacity of the disk volume;
2. the file access time, defined as the time taken to read or write a file, averaged over 1,000 observations (the normalized file access time is the file access time divided by the file size);
3. the CPU consumption, defined as the CPU's nonidle time; and
4. the CPU utilization, defined as the CPU consumption divided by the total elapsed time.

4.3 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 25 GBytes, 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 percent utilized, thus giving an average space utilization of 75 percent. While we can probably improve 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. Its effective space utilization is the space utilization when the first data block is irrecoverably corrupted—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. As reported in [7], with a replication factor of 4, the space utilization can only reach seven percent for a disk with 1,000,000 blocks. Experiments on our disk volume comprising 25,000,000 blocks show that the average space utilization cannot exceed four percent even with a replication factor of 16. It is reasonable that larger storage space produces lower space utilizations since block corruptions occur more frequently in a disk volume made up of more blocks than one with fewer blocks.

Finally, we consider the StegFD scheme. Here, the only storage overheads are incurred by the abandoned blocks, 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. Assuming that the percentage of abandoned blocks in the disk volume is one percent, the dummy hidden files occupy another 1 percent of disk space, and each hidden file contains a maximum of 10 free blocks, StegFD is able to consistently achieve more than 80 percent space utilization.

To summarize, we have arrived at a couple of observations. 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 four percent space utilization for a 25 GByte volume, and less for larger disks, before data corruption sets in. Third, the proposed StegFD is capable of achieving higher space utilizations than StegCover and is at least 20 times more space efficient than StegRand.

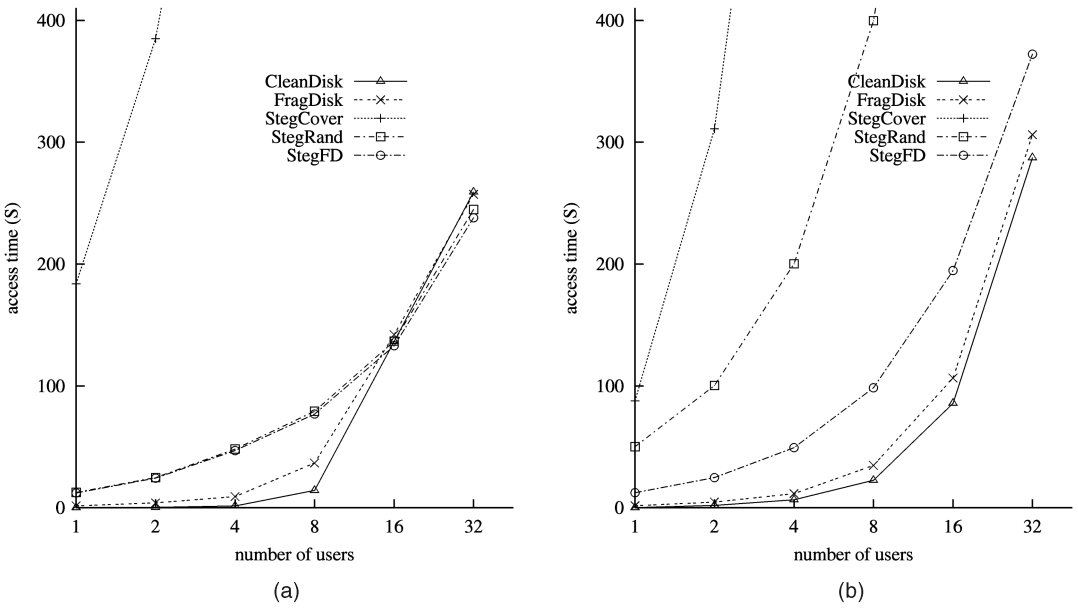


Fig. 6. Sensitivity to concurrency. (a) Read and (b) write.

4.4 Performance Analysis

Having demonstrated StegFD'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 25 GBytes and 1 KBytes, respectively, while the file sizes vary uniformly between 1 and 2 MBytes.

Figs. 6a and 6b 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 no better than StegFD's due to the need to hunt for an intact replica when the primary copy of a file is found to be corrupted, whereas the write access times are much worse because all the replicas must be updated.

As for StegFD, 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 disk. However, the differentiation diminishes with increased workload, as file operations become increasingly interleaved. In fact, StegFD matches both CleanDisk and FragDisk from 16 concurrent users onwards for read operations. For write operations, the performance of StegFD also converges toward those of CleanDisk and FragDisk with more concurrent users. Finally, the relative trade offs between the various schemes are independent of the file size, as shown in Figs. 7a and 7b (for single user context).

In summary, this experiment shows 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, StegFD is a practical steganographic file system that delivers similar performance to the native Linux file system in a multiuser environment.

4.5 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. We fix the number of concurrent users at 1, while maintaining the other workload parameters at their settings in the previous experiment.

Figs. 8a and 8b 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 eight blocks, is slower due to the overhead in seeking to each fragment. This indicates that, as the file system gets more fragmented, its performance would gradually degrade to that of StegFD even in single-user environments where file operations are not interleaved. The difference in performance is more pronounced with small block sizes where FragDisk has to perform more fragment seeks, and StegFD and StegRand incur more block seeks.

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

4.6 CPU Usage

The last set of experiments aims to evaluate the CPU usage of the various file systems. We vary the number of concurrent users and measure the CPU consumption and utilization for retrieving 1-MByte data files.

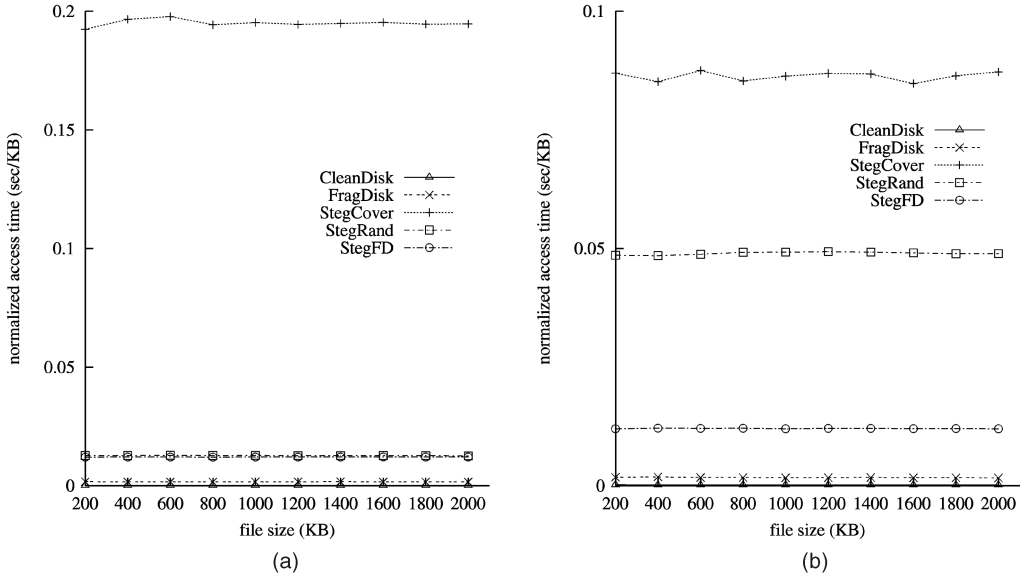


Fig. 7. Sensitivity to file size. (a) Read and (b) write.

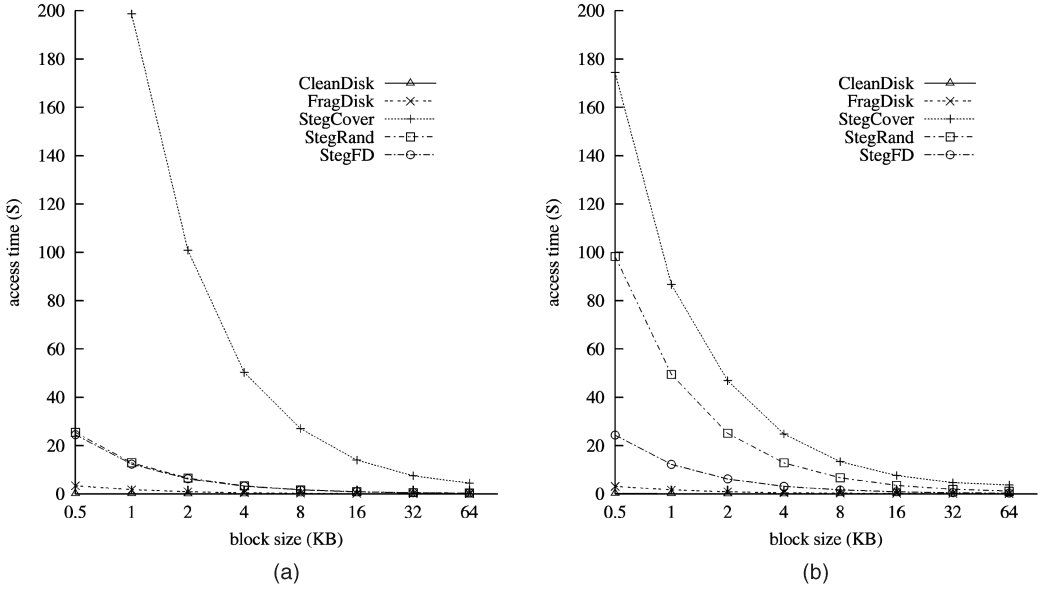


Fig. 8. Serial file operations. (a) Read and (b) write.

As shown in Fig. 9a, StegCover has the highest CPU consumption since it needs to retrieve 16 times more data than the other schemes. As StegRand and StegFD need to execute some cryptographic functions in each data retrieval or update, they incur more CPU overhead than CleanDisk and FragDisk. However, at low concurrency, StegRand and StegFD have lower CPU utilizations because their I/O costs are higher than those of CleanDisk and FragDisk. Nevertheless, with the exception of StegCover, the CPU utilizations of the tested file systems are no more than 10 percent as shown in Fig. 9b. This confirms that I/O cost is still the dominant performance determinant.

5 Steganographic B-Tree

Having devised a steganographic file system and demonstrated that it incurs only marginal access time and space utilization penalties over conventional file systems, we are keen to investigate its efficacy in supporting specialized applications; in particular, relational DBMSs that must be highly optimized. In this section, we study how efficiently operations can be carried out on B-trees, one of the key index structures in relational DBMSs, within a StegFD volume.

5.1 Construction of Steganographic B-Tree

A straightforward way to hide the existence of a database is to install a conventional DBMS on a StegFD volume. This causes the DBMS to store the database, including its B-tree indices, as one or more hidden files that are managed by StegFD. The advantage is that this entails no modification to the DBMS. However, if there is a mismatch in the block sizes of the DBMS and StegFD, StegFD would either need multiple I/O operations to satisfy each node access, or it

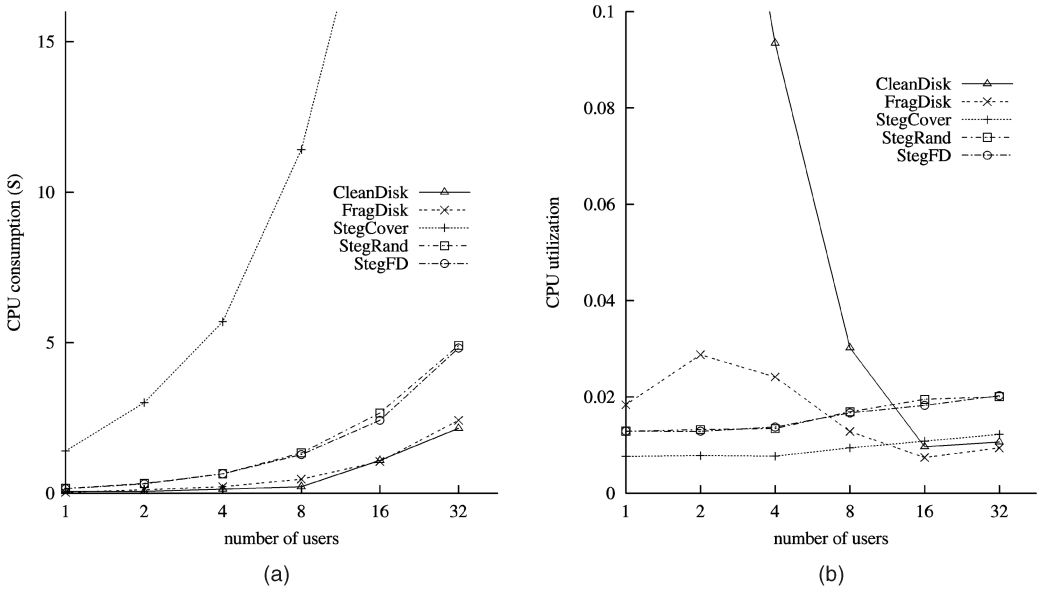


Fig. 9. CPU usage. (a) CPU consumption and (b) CPU utilization.

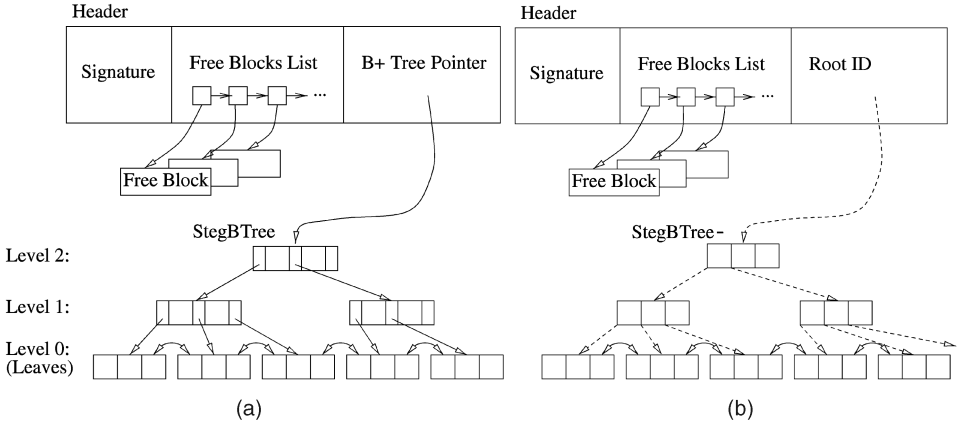


Fig. 10. Structure of StegBtree(-). (a) StegBtree and (b) StegBtree-.

would fetch more data than necessary each time. Even when the DBMS is configured with the same block size as StegFD, the node boundaries in the DBMS may not align with the block boundaries in StegFD. Hence, there is an expected performance degradation. In an attempt to overcome this penalty, we propose two schemes for implementing B-trees directly in a steganographic disk volume.

In the first scheme, each B-tree begins with a header as illustrated in Fig. 10a. The first two structures in the header, signature and free blocks list, work the same way as with hidden files (see Section 3.1). Unlike a hidden file that links its data blocks in a linear chain, here the index nodes are linked into a B-tree structure. Having located the B-tree through its header, operations like insertion, search, and deletion can be carried out according to the usual algorithms. We denote this scheme as StegBtree.

The second scheme for implementing a steganographic B-tree is similar to StegBtree, except that the child pointers in the nonleaf nodes are not stored explicitly. Instead, the address of a node P_i is calculated on-the-fly, by applying a hash function on the corresponding index entry K_i, the node's level number and the file access key, i.e.,

$$\begin{cases} P_0 = HASH(NodeAddress, level\#, FAK) \\ P_i = HASH(K_i, level\#, FAK) & \text{for all } i > 0, \end{cases}$$

where NodeAddress is the physical address of P_0's father node. The address of the root node is calculated by applying the hash function to the root id, which is recorded in the file header. Address collisions that may be encountered by the B-tree nodes are handled the same way as with file headers in StegFD. This pointer-less scheme, StegBtree-, is shown in Fig. 10b. The space saving from omitting the child pointers allows each nonleaf node to hold more keys, leading to a higher fan-out and fewer nodes, which can potentially speed up operations on the B-tree.

Algorithms for node allocation, search, and insertion on StegBtree are given in Fig. 11. Function *allocate()* allocates a new node to StegBtree-. It repeatedly applies a hash function on the input arguments until a free page is found and returns this page as the new node. Function *locate()* makes use of the same hash function and the same procedure as *allocate()* to locate an existing node from the storage space. The procedure *search()* for StegBtree- is similar to that of a regular B^{+}-tree, except that it does not use pointers to locate tree nodes, but uses the function *locate()* to calculate the node addresses

```
func allocate (K, level#, FAK) returns address
P = HASH (K, level#, FAK);
loop
    if Block *P is a free block, then
        return P;
    else,
        P = HASH (P);
end loop;
endfunc

func locate (K, level#, FAK) returns address
P = HASH (K, level#, FAK);
loop
    if Block *P's signature is correct, then
        return P;
    else,
        P = HASH (P);
end loop;
endfunc

func search (nodeaddress, K) returns address
// level# is the current level number;
// K_m is the last entry in this node;

if *nodeaddress is a leaf, return nodeaddress;
else,
    if K< K_1 then
        P = locate (nodeaddress, level#-1, FAK);
    else if K ≥ K_m then
        P = locate (K_m, level#-1, FAK);
    else,
        find i such that K_i ≤ K < K_{i+1};
        P = locate (K_i, level#-1, FAK);
    return search (P, K);
endfunc

proc range_search (K_1, K_2, (out) results)
P_1 = search (root, K_1);
begin from P_1, and follow the leaf link list until get
P_2 which contains the 1st entry greater than K_2;
add all the leaf nodes between P_1 and P_2 to results;
endproc

proc insert (nodeaddress, entry, newchildentry)
// insert 'entry' into subtree with root '*nodeaddress';
// degree is d; 'newchildentry' is null initially, and
// null upon return unless child is split;
// level# is the current level number;

if *nodeaddress is a non leaf node, say N,
    if K< K_1 then
        P = locate (nodeaddress, level#-1, FAK);
    else if K ≥ K_m then
        P = locate (K_m, level#-1, FAK);
    else,
        find i such that K_i ≤ K < K_{i+1};
        P = locate (K_i, level#-1, FAK);
    insert(P, entry, newchildentry);
    if newchildentry is null, return;
    else,
        if N has space,
            put newchildentry on it,
            set newchildentry to null, return;
        else,       // split N:
            first d key values stay,
            N2 = allocate (K_{d+1}, level#, FAK),
            last d keys move to new node N2;
            newchildentry = < K_{d+1} >;
            if N is the root,
                A_0= allocate (New Root ID, level#+1, FAK),
                insert < K_{d+1} > into *A_0;
                replace Root ID with New Root ID;
                // relocate the 1st node of each level:
                B = nodeaddress;
                for i = level# to 0, loop
                    A_1 = allocate (A_0, i, FAK);
                    copy *B to *A_1, release *B ;
                    B = locate (B, i-1, FAK), A_0=A_1;
                end loop;
            return;

else if *nodeaddress is leaf node, say L,
    if L has space,
        put entry on it and return;
    else,
        split L: first d entries stay,
        L2 = allocate (K_{d+1}, 0, FAK),
        rest move to brand new node L2;
        newchildentry = < K_{d+1} >;
        return;
endproc
```

Fig. 11. StegBTree- algorithms.

instead. The procedure *insert()* employs a similar insertion algorithm as B^+-tree, except that it calls the *allocate()* function to create new nodes for the B-tree. As Fig. 11 shows, when a node is split during insertion, the middle entry is passed to the *allocate()* function to create a new node and, thereafter, all the index entries in the original node with larger key values than the middle entry are shifted to the new node. As all the existing nodes of StegBtree- remain unchanged during insertion, it does not incur extra overhead. Only when the root node is split and the tree grows up a level, it takes a bit more effort to reorganize the StegBtree-. In that case, a new root node is allocated by passing a new root id to the *allocate()* function. The update of root id requires the first node of each level of the StegBtree- to be reallocated accordingly, as its address is directly or indirectly determined by the root id through the hash function.

To provide native support for B-tree indices in StegFD, we have added two new sets of APIs, one for StegBtree and the other for StegBtree-. The APIs can be found at the StegFD Web site (http://xena1.ddns.comp.nus.edu.sg/SecureDBMS/).

5.2 Experiments

To investigate the efficacy of *StegBtree* and *StegBtree-*, we compare them with the alternatives of a) constructing the B-trees directly on a raw disk (Btree) and b) storing the B-trees in hidden files on a StegFD volume (Btree on

TABLE 3
B-Tree Parameters

Parameter	Default
Table size	35,000 Tuples
Tuple size	256 Bytes
Node size	4 KBytes
Key size	16 Bytes
Pointer size	4 Bytes

StegFD). Table 3 summarizes the experiment parameters. The physical resource and workload parameters remain the same as in Tables 1 and 2.

5.2.1 Sensitivity to Space Utilization

We begin the profiling of the steganographic B-tree schemes by evaluating their sensitivity to the utilization level of the StegFD volume. Fig. 12 shows the average access time of 400 exact-match queries for the various B-tree schemes.

As expected, *Btree on StegFD* is much slower than the other schemes because it has a different node size from StegFD's block size, and the node boundaries are not aligned with StegFD's block boundaries, thus incurring multiple I/O operations for each node access. For *StegBtree*, there is some overhead in processing the header block to locate the B-tree, but the resulting penalty over *Btree* is well within 20 percent. In contrast, *StegBtree-* performs just as well as *Btree* initially because the former's larger fan-out and, hence, shorter height compensate for the I/Os on the header block. However, higher space utilizations lead to more frequent address collisions, and the extra I/Os in tracking down index nodes cause performance to degrade rapidly beyond 40 percent utilization.

This experiment confirms that native support for B-tree should be built into StegFD. Among the two steganographic B-tree schemes, *StegBtree-* is ideal for sparsely populated volumes, whereas *StegBtree* consistently achieves performance that is just marginally slower than *Btree*.

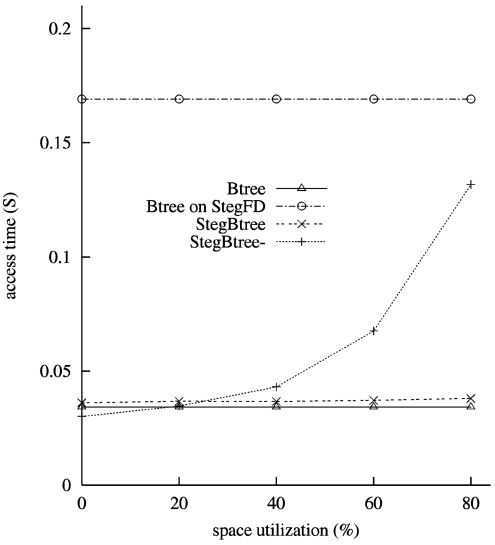


Fig. 12. Sensitivity to space utilization.

5.2.2 Sensitivity to Query Selectivity

The second set of experiments is intended to study the behavior of *StegBtree* and *StegBtree-* with range queries. Here, we vary the query selectivity from 1,000 tuples to 10,000 tuples. Figs. 13a and 13b give the results for clustered and unclustered indices, respectively.

For clustered indices, *Btree* is clearly the fastest, especially at high selectivity factors where data access time dominates index access time. This is because *Btree* benefits from sequential I/Os as data pages are stored at contiguous addresses, whereas the other three schemes incur random I/O operations. However, for unclustered indices, *Btree* has no advantage over *StegBtree* and *StegBtree-*. Finally, we observe that *Btree on StegFD* is still the worst performer.

5.2.3 Sensitivity to Concurrency

Having discovered that *Btree* can be superior to the steganographic B-tree schemes, we are interested to find

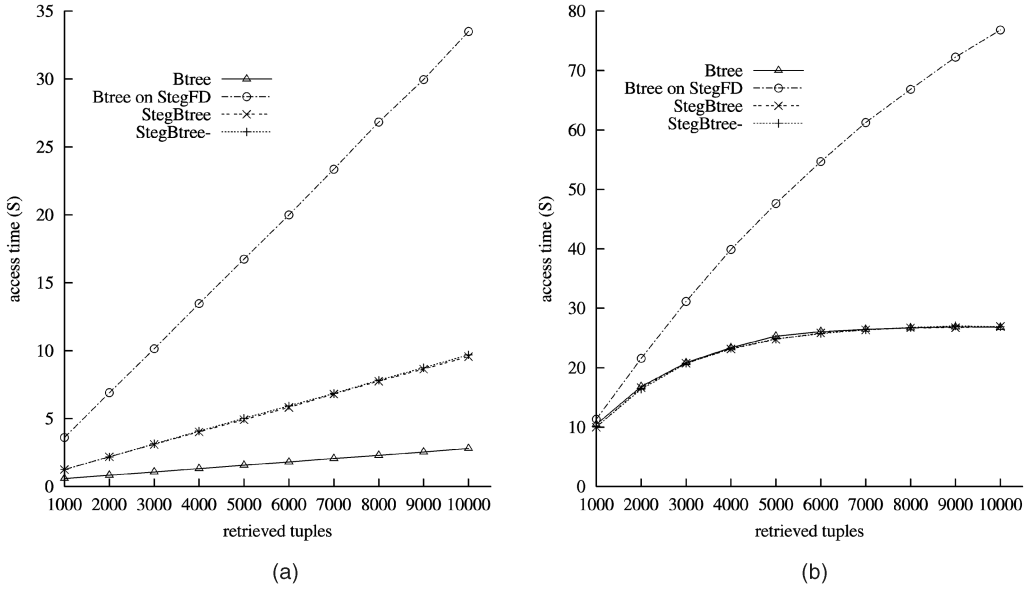


Fig. 13. Sensitivity to query selectivity. (a) Clustered and (b) unclustered.

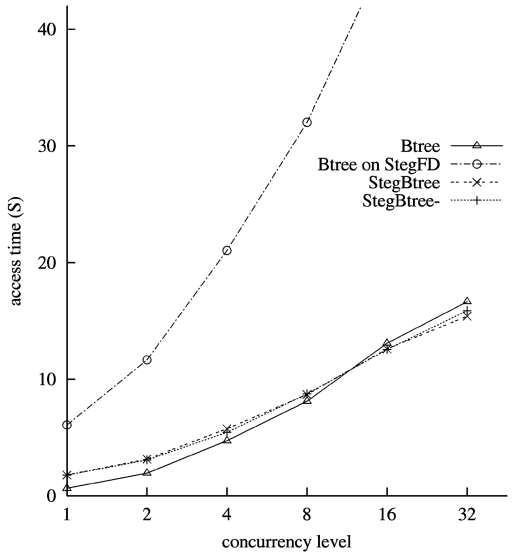


Fig. 14. Sensitivity to concurrency.

out whether this relative performance still holds in a multiuser environment. Instead of issuing queries one after another as in the earlier experiments, we now generate multiple range queries (for 2,000 tuples each) concurrently on a clustered index. Fig. 14 plots the access time against the number of concurrent queries.

As shown in the figure, increased concurrency slows down all of the schemes. Moreover, the access time of *Btree* gradually approaches those of *StegBtree* and *StegBtree-*. This is due to the larger amount of random I/O operations when queries are interleaved. Hence, in practice, *StegBtree* and *StegBtree-* are likely to fare favorably relative to *Btree*, and even clustered B-trees.

6 Conclusion

In this paper, we have introduced StegFD, a practical scheme to implement a steganographic file system that offers plausible deniability to owners of protected files. StegFD 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. StegFD achieves this steganographic property, while ensuring the integrity of the files and maintaining efficient space utilization at the same time.

We have also proposed two schemes for implementing Steganographic B-trees in a StegFD volume.

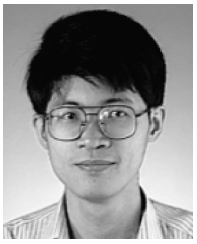
We have implemented StegFD as a file system driver in the Linux kernel 2.4. Extensive experiments on the system confirm that StegFD is capable of achieving an order of magnitude improvements in performance and/or space utilization over the existing steganographic schemes. In fact, StegFD is just as fast in a multiuser 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 StegFD 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 discussed in Section 3.4. Building a P2P-based StegFD as an application on top of BestPeer [14] is also on our agenda.

References

[1] Drivecrypt Secure Hard Disk Encryption, http://www.securstar.com, Mar. 2004.
[2] E4m Disk Encryption, http://www.e4m.net, June 2003.
[3] Encrypting File System (efs) for Windows 2000, http://www.microsoft.com/windows2000/techinfo/howitworks/security/encrypt.asp, Mar. 2004.
[4] Pgpdisk, http://www.pgpi.org/products/pgpdisk/, Mar. 2004.
[5] *Advanced Encryption Standard.* Nat'l Inst. of Science and Technology, FIPS 197, 2001.
[6] *Secure Hashing Algorithm.* Nat'l Inst. of Science and Technology, FIPS 180-2, 2001.
[7] R. Anderson, R. Needham, and A. Shamir, "The Steganographic File System," *Proc. Information Hiding, Second Int'l Workshop,* D. Aucsmith, ed., Apr. 1998.
[8] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," *Proc. First Dutch Int'l Symp. Linux,* 1995.
[9] M. Chapman and G. Davida, *Information and Communications Security—First Int'l Conf.,* Nov. 1997.
[10] S. Hand and T. Roscoe, "Mnemosyne: Peer-to-Peer Steganographic Storage," *Electronic Proc. First Int'l Workshop Peer-to-Peer Systems (IPTPS '02),* Mar. 2002, http://www.cs.rice.edu/Conferences/IPTPS02/.
[11] F. Hartung, J.K. Su, and B. Girod, "Digital Watermarking for Compressed Video," *Multimedia and Security—Workshop at ACM Multimedia '98,* Sept. 1998.
[12] N.F. Johnson and S. Jajodia, "Exploring Steganography: Seeing the Unseen," *Computer,* vol. 31, no. 2, pp. 26-34, Feb. 1998.
[13] A.D. McDonald and M.G. Kuhn, "Stegfs: A Steganographic File System for Linux," *Proc. Workshop Information Hiding, (IHW '99),* Sept. 1999.
[14] W.S. Ng, B.C. Ooi, and K.L. Tan, "Bestpeer: A Self-Configurable Peer-to-Peer System," *Proc. 18th Int'l Conf. Data Eng.,* p. 272, Apr. 2002. (Poster Paper).
[15] H. Pang, K.L. Tan, and X. Zhou, "StegFS: A Steganographic File System," *Proc. 19th Int'l Conf. Data Eng.,* pp. 657-668, Mar. 2003.
[16] M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance," *J. ACM,* vol. 36, no. 2, pp. 335-348, Apr. 1989.
[17] R.L. Rivest, *RFC 1321: The MD5 Message-Digest Algorithm.* Internet Activities Board, 1992.
[18] G. Simmons, "The Prisoners' Problem and the Subliminal Channel," *Proc. CRYPTO '83 Conf.,,* pp. 51-67, 1984.
[19] M.D. Swanson, B. Zhu, and A.H. Tewfik, "Audio Watermarking and Data Embedding—Current State of the Art, Challenges and Future Directions," *Proc. Multimedia and Security—Workshop at ACM Multimedia '98,* Sept. 1998.
[20] A.S. Tanenbaum and A.S. Woodhul, *Operating Systems: Design and Implementation,* second ed. Prentice Hall, 1997.
[21] Y. Yang, F. Bao, and R. Deng, "Improving and Cryptanalysis of a Key Recovery System," *Proc. 2002 Australasian Conf. Information Security and Privacy,* pp. 17-24, 2002.

HweeHwa Pang received the BSc degree with first class honors and the MS degree from the National University of Singapore in 1989 and 1991, respectively, and the PhD degree from the University of Wisconsin at Madison in 1994, all in computer science. His research interests include database management systems, data security and quality, operating systems, and multimedia servers. He has many years of hands-on experience in system implementation and project management. He has also participated in transferring some of his research results to industry.



Kian-Lee Tan received the BSc (Hons) and PhD degrees in computer science from the National University of Singapore, in 1989 and 1994, respectively. He is currently an associate professor in the Department of Computer Science, National University of Singapore. His major research interests include query processing and optimization, database security, and database performance. He has published more than 100 conference/journal papers in international conferences and journals. He has also coauthored three books. He is a member of the Association of Computing Machinery (ACM) and the IEEE.

Xuan Zhou received the BSc degree from Fudan University of China in 2001. Currently, he is a PhD student in the Department of Computer Science, National University of Singapore. His research interests include information security and database system.