StegFS: A Steganographic File System for Linux

Andrew D. McDonald and Markus G. Kuhn[*]

University of Cambridge, Computer Laboratory, New Museums Site,
Pembroke Street, Cambridge CB2 3QG, United Kingdom
a.d.mcdonald@bcs.org.uk
mgk25@cl.cam.ac.uk

Abstract. Cryptographic file systems provide little protection against legal or illegal instruments that force the owner of data to release decryption keys for stored data once the presence of encrypted data on an inspected computer has been established. We are interested in how cryptographic file systems can be extended to provide additional protection for such a scenario and we have extended the standard Linux file system (Ext2fs) with a plausible-deniability encryption function. Even though it is obvious that our computer has harddisk encryption software installed and might contain some encrypted data, an inspector will not be able to determine whether we have revealed the access keys to all security levels or only those to a few selected ones. We describe the design of our freely available implementation of this steganographic file system and discuss its security and performance characteristics.

1 Introduction

Various implementations of cryptographic file systems have been made widely available. Examples include CFS [1] for Unix, TCFS [2] for Linux, and EFS [3] for Windows, which transparently encrypt individual files, as well as the Linux loopback device [4] and SFS [5] for Microsoft platforms, which encrypt entire disk partitions. Cryptographic file systems store files and associated metadata only in encrypted form on non-volatile media. They can provide the user some protection against the unwanted disclosure of information to anyone who gets physical control over the storage unit.

Assuming correctly implemented encryption software is used as designed, and cryptanalysis remains infeasible, an attacker can still chose among various tactics to enforce access to encrypted file systems. Brief physical access to a computer is, for instance, sufficient to install additional software or hardware that allow an attacker to reconstruct encryption keys at a distance. (UHF burst transmitters that can be installed by non-experts inside any PC keyboard within 10–12 minutes are now commercially available, as are eavesdropping drivers that will covertly transmit keystrokes and secret keys via network links.) An entirely different class of tactics focuses on the key holders, who can be threatened with sanctions as long as there remains undecryptable ciphertext on their storage

devices. We are interested in data protection technologies for this latter case, especially considering that attackers can often be more persuasive when the data owner cannot plausibly deny that not all access keys have already been revealed.

Plausible deniability [7] shall refer here to a security property of a mechanism that allows parties to claim to others (e.g., a judge) that some information is not in their possession or that some transaction has not taken place. The one-time pad is a well-known encryption technique with plausible deniability, because for every given ciphertext, a decryption key can be found that leads to a harmless message. However such schemes are only practical for short messages and are not generally suited for data storage.

Anderson, Needham and Shamir [6] outlined first designs for encrypted file stores with a plausible-deniability mechanism, which they called *steganographic file systems*. They aim to provide a secure file system where the risk of users being forced to reveal their keys or other private data is diminished by allowing the users to deny believably that any further encrypted data is located on the disk. Steganographic file systems are designed to give a high degree of protection against compulsion to disclose their contents. A user who knows the password for a set of files can access it. Attackers without this knowledge cannot gain any information as to whether the file exists or not, even if they have full access to the hardware and software.

The authors of [6] outline two different ways of constructing such a system. The first makes the assumption that the attacker has no knowledge of the stored plaintexts and instead of strong cipher algorithms, only linear algebra operations are required. The scheme operates on a set of cover files with initially random content. Then data files are stored by modifying the initially random cover files such that the plaintext can be retrieved as a linear combination of them. The password to access a file corresponds to the subset of cover files that has to be XOR-ed together to reconstruct the hidden file. The number of cover files must be sufficiently large to guarantee that trying all subsets of cover files remains computationally infeasible. We decided not to use this approach in our implementation, because a lot of cover files would have to be read and XOR-ed to ensure computational security. In addition to this prospect of low performance for both read and write access, we felt uncomfortable with the requirement that the attacker must not know any part of the plaintext.

In the second approach outlined in [6], the file system is initially filled completely with blocks of random data. The file blocks are hidden amongst this random data by writing the encrypted blocks to pseudo-random locations using a key derived from the file name and directory password. The file blocks are then indistinguishable from the random data. As blocks are written to the file system, collisions will occur and blocks will be overwritten. This starts to occur frequently after about $\sqrt{n}$ blocks have been used, where n is the number of blocks in the file system, as the birthday paradox suggests. Only a small proportion of the disk space could safely be utilised this way, therefore multiple copies of each

block have to be written and a method is needed to identify when they have been overwritten.

Van Schaik and Smeddle [8] have implemented a file system, which they describe as being inspired by [6], however it falls some way short of meeting the security and plausible deniability aims originally described. They neither hide information by linearly combining password-selected subsets of blocks as suggested in the first method, nor do they replicate blocks as in the second method. Instead they mark blocks as 'might be used by the higher security levels'. Hence, from one security level it can be seen that others exist, although the exact quantity of data stored in this way is obscured. In this way, they avoid open files accidentally overwriting hidden data, but at the same time, they also provide inspectors with a low upper bound on the amount of hidden data.

Other steganographic file storage systems in the past have used the idea of storing files in large amounts of non-random but slightly noisy covertexts such as a graphics or audio file. For example, ScramDisk [13] for MS-Windows allows the ciphertext blocks of its cryptographic file system to be stored in the least-significant bits of an audio file. The Linux encrypting loopback block device can also work in this way.

2 Basic Concept

The design of our hidden file system was inspired by the second construction in [6], but it differs substantially from it and is, in our view, more practical and efficient. We do not require the use of a separate partition of a harddisk, but instead place hidden files into unused blocks of a partition that also contains normal files, managed under a standard file system.

While the second construction in [6] allocates blocks purely based on a hash function of the file name, we use instead a separate block allocation table. It fulfills a similar function as the block allocation bitmap in traditional file systems, but instead of a single bit for each block, it contains an entire 128-bit encrypted entry, which is indistinguishable from random bits unless the key for accessing a security level is available. The additional block allocation table has a length that depends only on the partition size. We ensure that it is always present if the steganographic file system driver has been installed on this system, no matter whether it is being actively used or not. The encryption ensures that inspectors can gain no further information from this table beyond the fact that StegFS has been installed. Therefore, we see no need to especially hide its presence, because we do not properly hide the steganographic file system driver itself anyway.

A plausible explanation might have to be given for the installation of the StegFS driver in any case, usually by providing inspectors access to some of the lower security levels that could be filled with mildly compromising material. The privacy protection of our file system is not provided by giving no indication of whether any hidden files are present or not. It is only impossible to find out how many different security levels of files are actually used. Inspectors could of course force the user to fill the entire remaining disk space with new data,

which is guaranteed to overwrite most hidden data. Our file system provides no protection against this active attack and we envisage that a user typically has backups hidden elsewhere and would rather have his files destroyed than hand them over.

Our construction has over the ones discussed in [6] the advantage of providing a hidden file system that follows very closely all the standard Unix file system semantics. We have subdirectories, inodes, symbolic and hard links, etc. Error conditions such as lost hidden files in our system are identified and signalled via error codes that correspond to closely related conditions in normal file systems, such that standard software will automatically act appropriately. We can use arbitrary positions for blocks, which allows us to coordinate block allocation with the non-hidden file system that shares the same partition and which also allows us to utilize the entire disk.

3   Implementation

File system support in the Linux kernel [9] is structured into a system call interface, a virtual file system (VFS) interface, several file-system specific drivers, a buffer cache and a series of hardware-specific device drivers (Fig. 1). The standard Linux harddisk file system is the Second Extended File System (Ext2fs) [11], but other alternatives such as Minix, DOS, and ISO 9660 can be selected at mount time. Our steganographic file system implementation (StegFS) is installed alongside the normal Ext2fs, Minix, etc. drivers between the VFS interface and the blockbuffer cache.

StegFS partitions are compatible with Ext2fs partitions. The StegFS driver can work on Ext2fs partitions and vice versa. This allows the StegFS driver to be removed entirely from the system and the non-hidden files in the StegFS partition can still be accessed using the standard Ext2fs driver. In such a situation, the StegFS partition will just look like a partition in which unused blocks have recently been overwritten with random bytes using some disk wiping tool.

StegFS contains the full functionality of the Ext2fs driver for compatible access to non-hidden files. In addition, it can store hidden files in the blocks that are currently unused by Ext2fs. As long as no hidden security levels are opened, StegFS behaves almost exactly like Ext2fs. One exception is that when files are deleted, their blocks will immediately be overwritten with random bytes. Another modification is that a small fraction of newly created files is placed at a somewhat more random location on the disk than normal, which will help to plausibly explain to inspectors why sometimes a few unused blocks in the middle of a large range of unmodified free blocks have changed between two inspections. They could either belong to a new hidden file or to one of the non-hidden Ext2fs files that was allocated randomly and later deleted. As soon as hidden security levels are opened, the behaviour of the Ext2fs block allocation routine changes further, such that blocks used by the now visible hidden files will not be overwritten by the Ext2fs section.
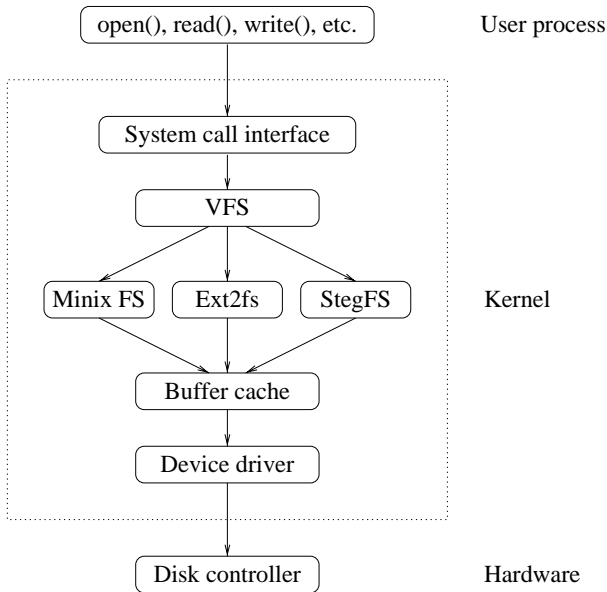
Fig. 1. The StegFS driver in the Linux kernel offers an alternative to the normal Ext2fs driver and includes its full functionality.

We selected Ext2fs, because its widespread use allows us to continue using the non-hidden files in a plausible way even when the StegFS driver has to be removed. Its design is fairly simple and similar to the traditional Unix file system [10]. The use of bitmaps to mark used blocks simplifies block allocation for the steganographic part.

StegFS was built starting with the freely available Ext2fs code. It contains essentially two parallel instances of the Ext2fs functions, an only slightly modified one for access to normal files, and a substantially modified one for access to hidden files. We have effectively two file systems supported in parallel in one driver on one partition. The StegFS implementation adds about 5000 lines of code to the 5400 lines of the Ext2fs driver.

We briefly considered an alternative design in which we would have mounted the same block device multiple times at different security levels. However the VFS interface did not allow this, and the blockbuffer cache was not designed for concurrent access by several drivers to the same block device. We would have had to implement additional synchronization mechanisms between the various drivers and this would have made the design more complex.

When a StegFS partition is mounted as a StegFS file system, it behaves initially almost exactly like a normal Ext2 file system, except for the random overwriting of deleted file blocks and the occasionally more random placement of new files. The user can then use the `stegfsopen` tool in order to open additional security levels in which hidden files become visible and can be stored. Once the

first StegFS level is opened, a directory called 'stegfs' appears in the root directory of the file system. Under this directory additional directories called '1', '2', etc. appear, one for each open security level.

Hidden files are stored in a very similar way to normal Ext2fs files. We also distinguish regular files and directory files. Directories contain file names and inode numbers. Inodes contain file attributes and a list of the blocks assigned to this file. To access a hidden file, we first have to access the root directory of a hidden security level, then traverse the directory path to the inode of the file and finally access the data blocks referenced by the inode. The main difference between normal files and hidden files is that the allocation of new blocks by the Ext2fs driver could overwrite blocks of hidden files when the corresponding security level is not open. In addition, since we cannot plausibly justify sparing any block from being used by the normal file system, we cannot use a fixed location even for the root directory inode of any hidden level.

It is, therefore, necessary that both inodes and data blocks of hidden files are replicated throughout the partition, such that the data can still be recovered after some blocks have been overwritten. The following data structures help the StegFS driver to locate these replicated files.

3.1   Block Table

The block table is the StegFS equivalent of the block allocation bitmap. It contains one entry per block on disk. The main purpose of this table is to store encrypted checksums for each block such that overwritten blocks can be detected. It also stores the inode numbers for blocks containing inodes, such that inodes can be located by searching for their number in the table. The block table is stored in a separate normal non-hidden file.

Each entry in the block table is 128 bits long. It is encrypted under the same key as the data in the corresponding disk block. Each entry consists of the following three 32-bit and two 16-bit variables:

```
struct stegfs_btable {
    uint32_t magic1;
    uint16_t magic2;
    uint16_t iv;
    uint32_t bchecksum;
    uint32_t ino;
}
```

Variable magic1 should always be 0. Variable magic2 is 1 if the corresponding block contains an inode and for a data block it is 0. Variable bchecksum contains the last 32 bits of a block's encrypted contents, which is for our purposes equivalent to a checksum of the block since the blocks are CBC encrypted using the zero-padded value iv as the initial vector. Variable ino contains the inode number of this file. Table entries corresponding to unused blocks contain random data. Note that if we do not know the key for the security level of a block, we

cannot distinguish whether is is used or not. Variables `magic1` and `magic2` together contain 47 bits of redundancy that allow us to determine quickly whether a block is used under a security level for which we know the key. The checksum allows us to test whether a block has been overwritten when files were written under Ext2fs while StegFS was not installed. It also allows a StegFS repair tool to eliminate table entries that are unused but look after decryption accidentally like used ones, which might happen with a probability of 2^{-47} per block.

3.2   Inode Structures

The hidden StegFS inodes resemble those of Ext2fs, but contain in addition the number of replicas that were made of the file's inode and data blocks. The list of data blocks has 12 direct blocks, one indirect, one double indirect, and one triple indirect block, just like in an Ext2fs inode. However, instead of just a reference to a single data block, each hidden StegFS inode contains references to all copies of this block (Figure 2).
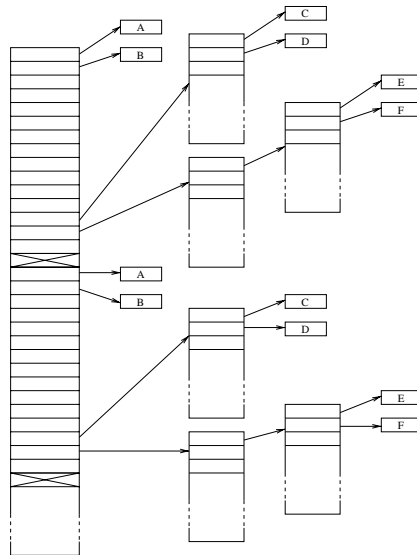

Fig. 2. A StegFS inode contains a sequence of several lists of inodes, each of which points to a different copy of all data blocks for a file. Boxes with the same letter represent blocks containing identical replicated data.

The hidden inodes are 1024 bytes in size, which is the most common size for the blocks in an Ext2 file system. Hence, each inode takes up one disk block. Several copies of each inode are stored to provide redundancy.

In the current version, we can have up to 28 copies of each hidden inode and 14 copies of each hidden file block. The number of copies of the inode and data

blocks is inherited from the directory in which the inode is created. The security levels that are used most often can have fewer copies since they are less likely to be overwritten.

The numbers of copies of inodes and data blocks can be altered by a pair of new `ioctl` request types, which provide access to the replication factors that are stored as new attributes in the inodes.

3.3   Virtual File System

The Linux VFS uses generic inode and superblock structures in memory. Both contain a file-system specific section. The StegFS versions of these extend those used by Ext2fs. We augmented the superblock structure by a number of additional fields for managing security levels (e.g., the keys for the currently opened levels), a pointer to the cipher functions structure and the block-table file. We also extended the VFS inode structure, which has to hold the same information as the hidden on-disk StegFS inodes, namely the replication factors and the locations of all replicated data blocks.

The VFS structures are, in the current StegFS version, larger than those for the standard file systems in Linux. Therefore, the kernel must be recompiled at the moment to install StegFS and it is not yet possible to install it just as a loadable kernel module without any other kernel modification.

3.4   Inode Numbers

An inode number is a 32-bit integer that uniquely identifies a file within a file system. In Ext2fs, the location of a file's inode on the disk can be computed directly from its inode number. For hidden files on a StegFS partition, we have to search for the inode in the decrypted block table and hope to find one that has not been overwritten. We decided to distinguish normal and hidden files by their inode number, such that operations on files can easily be directed to either the Ext2fs or the StegFS part of the StegFS driver. Inode numbers of hidden files also indicate the security level in the range 1 to 15

0	1	level	rest of inode number
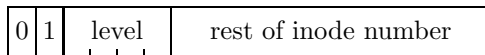
such that the right decryption key can be selected easily, while non-hidden files have inode numbers of the form

0	0	rest of inode number
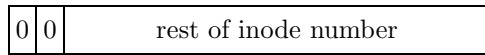
In order to avoid that we have to search the entire blocktable for inodes of hidden files, their block locations are selected by hashing the security level key and the inode number together with a hash sequence number. When we create a new file, the hash sequence number is increased until enough free blocks have been found for all copies of the new inode. We go through the same sequence to locate inodes when they are loaded. The decrypted inodes are cached within the VFS, so this search does not have to be repeated often while a file is in use.

3.5 Block Allocation

Data and inode-indirection blocks in the hidden file system are allocated at random locations on the disk. The Linux kernel provides a random number source in form of the `/dev/urandom` driver, which maintains an entropy pool by hashing timestamps of interrupts [12]. We use this random number generator to select the first free block that we allocate for a file. The first copy of each following block is allocated in the next free block. The additional copies of every block are written starting from completely independent random locations, in order to ensure that the overwrite probability of the various copies remains independent.

Before a block is allocated, we first test whether it is marked as used in the Ext2fs bitmap. If not, we attempt to decrypt the corresponding entry in the block table using each of the known level keys. If for none of the keys the first 47 bits of the decrypted entry are all zero, the block is allocated, otherwise we start the same test on the next free block in the Ext2fs bitmap.

We also tried an alternative method in which each hidden block is assigned completely independently from previous blocks to a random free location. This approach turned out to have both a performance and a security problem. The non-locality of block allocations caused very slow read performance because the harddisk spent most of the time accelerating the read head. In addition, an inspector of the block assignment pattern of the non-hidden Ext2fs files could become suspicious. Ext2fs files would frequently have gaps of single blocks in their assignment pattern, because they had to jump over each single block of the randomly distributed cloud of blocks used by hidden files. Even though the blocks of hidden files cannot be distinguished from deleted files, the large number of supposedly recently deleted single-block files would look rather suspicious and might allow an inspector to estimate the number of blocks occupied by hidden files. It is, therefore, important that the allocation patterns for hidden files are sufficiently similar to the allocation patterns of Ext2fs files with occasionally random placement.

For both normal and hidden files, a block and the corresponding block-table entry are overwritten completely with random bytes when the block is deallocated. This not only ensures that information is really purged when a file is deleted, it also makes deleted normal files indistinguishable from hidden files, as long as the right security level key for a hidden file is not available.

3.6 Block Replication

Multiple copies of both inodes and data blocks are stored on disk, so that if one or more copies are destroyed then hopefully others will remain intact. When reading files, usually only the first copy of any given block will be required. If the checksum for this copy is correct then the block will be decrypted, otherwise further copies will be tried. When writing into a file and only a part of the block is changed, it is first necessary to read and decrypt the corresponding block. After the changes have been made, the corresponding initial vector is modified, the block is encrypted again and written back to disk. The block allocation table

entry has to be updated with the new initial vector and checksum. We then have to go through the list of replicated copies of this block and read each of these in order to check whether it has been overwritten in the meantime or is still valid. If the block is still valid, then we just encrypt the new data block for these locations and write it to disk. If the checksum test indicates that the block has already been overwritten, then we allocate a new block if the overwritten one is still in use by a non-hidden or lower level file. We then encrypt and write the data into this block, and update the inode and block table accordingly.

For hidden files, we can never assume that the content of a block is still correct, and so the checksum has to be verified on any access. Most significantly, the fact that we even have to read every copy of a block for the checksum test before we can overwrite it decreases the write performance. A future revision might, therefore, cache the overwrite status of blocks in memory.

Blocks are replicated when data is written out to disk. A simple method to ensure that the full number of copies exists is, hence, to read and rewrite the file. A new tool `rerpl` does this and should be used each time after the disk has been used in a lower security level. Only a small amount of each block in the file needs to be read and rewritten from a user process to regenerate the full number of copies. Inodes are re-replicated automatically by accessing the file, which updates the `atime` in the inode and causes it to be rewritten.

3.7   Key Management

Each hidden file belongs to one of 15 *security levels*. In order to allow groups of security levels to be opened with just a single passphrase, StegFS manages in addition 15 *security contexts*. Each such context gives access to a subset of all security levels and is protected by its own passphrase. In a typical default configuration, security context C provides access to security levels $1, \ldots, C$. This way, for every security level, there exists a passphrase that gives access to this and also all lower security levels. The user can add and remove any security level in any security context to build more complex hierarchies of security levels.

When a user opens a security context C and enters the corresponding passphrase PP_C, then it is hashed immediately using a secure hash function h to produce

$$HP_C = h(PP_C)$$

At the end of the block-table file, a 15×15-element security matrix M is appended, in which each entry is a 128-bit word. If security context C is to provide access to security level L, then

$$M_{C,L} = \{SK_L\}_{HP_C}$$

contains the level key SK_L encrypted with HP_C. All other matrix elements $M_{C,L}$ contain only a random value. The size of matrix M is hardwired like the block table and is independent of the number of security levels and contexts actually

used, therefore the open presence of this data does not reveal to an inspector any more information than the presence of the StegFS driver itself.

If disk block i belongs to a hidden file under security level L, then it and the related block table entry are encrypted under the security level key XOR-ed with the block number. So the encryption key for block i will be

$$BK_{L,i} = SK_L \oplus i.$$

Each block is then separately encrypted in Cipher Block Chaining (CBC) mode, using an initialisation vector (IV) that is stored in the corresponding block allocation table entry. We use only a 16-bit counter for this IV, because most files modified by applications are usually recreated entirely and therefore end up with a completely different location for every block, which anyway changes the keys used to encrypt the file and reduces the need to add further variability from a full-sized IV. In addition, as long as fewer than 2^8 inspections take place, inspectors will rarely see two modifications of a hidden block with the same 16-bit IV.

The current implementation offers the two Advanced Encryption Standard candidates Serpent and RC6 as block ciphers (using Gladman's implementations [14] with minor modifications) and the architecture allows other ciphers with the AES block size of 128 bits to be added easily later. An interesting alternative would be to use instead of AES-CBC a variable-length block cipher such as Block TEA [15, 16], which would eliminate the need to store an initial vector.

The SK_L for all open levels L are stored in the superblock structure in RAM and these cleartext keys are never written to the disk. The PP_C or HP_C values are overwritten in RAM as soon as the SK_L keys have been decrypted.

4   File System Usage

A StegFS file system has to be prepared as follows. After a partition has been created, we first place a normal Ext2 file system onto it with

```
mke2fs /dev/blockdevice
```

Any existing Ext2 file system can be used as well. We then use the command

```
mkstegfs /dev/blockdevice /path/to/btab
```

to fill all the empty blocks in the Ext2 file system with random data and create the block table file, whose size only depends on the size of the partition.

The block table also contains the space to store enough encrypted keys for 15 different security contexts and levels. The number 15 is hardwired and is deliberately not made user configurable, so as to give users plausible deniability for the number of security levels for which memory was allocated. They can then claim that the software forced them to allocate 15 levels even though they needed only 1 or 2. The program prompts for a passphrase for each context. The StegFS file system is now ready and can be mounted with:

```
mount /dev/blockdevice /mnt/mntpoint -t stegfs \
  -o btab=/path/to/btab
```

In order to open one of the StegFS security levels to access or deposit hidden files and directories, we use the command

```
stegfsopen /mnt/mntpoint contextnum
```

where `contextnum` is either the number of the security context that is to be opened, which by default gives access to security levels 1 to `contextnum`, or zero if all evidence of hidden files shall be removed. When this command is completed, the hidden files of each opened level become accessible in the subdirectories `/mnt/mntpoint/stegfs/1/`, `/mnt/mntpoint/stegfs/2/`, etc. If write operations have taken place while fewer or no security levels were open, the tool `rerpl` should be used to refresh all hidden files and ensure that the required number of replicated blocks is restored. The number of copies of each inode and copies of the blocks of a file can be controlled using the `tunestegfs` utility. The `stegfsctrl` tool allows the user to add and remove security levels from contexts, such that the linear default hierarchy of security levels can be broken up.

StegFS is available under the GNU General Public License from the authors in the form of a patch against Linux kernel 2.2.12 [9] plus user tools and can be downloaded via the Internet from

– `ftp://ftp.kerneli.org/pub/kerneli/net-source/StegFS/`
– `http://ban.joh.cam.ac.uk/~adm36/StegFS/`
– `http://www.cl.cam.ac.uk/~mgk25/stegfs/`

5   Other Design Issues

In our current implementation, the buffer cache keeps hidden blocks only in their encrypted form. This ensures that other file systems never can see plaintext block contents and do things with them that are beyond our control (such as writing parts of them to disk). It also means that decrypted buffers are not retained in memory since the buffer into which a block is decrypted will be overwritten immediately when the next block is re-encrypted. In addition, this helps to retain consistency between functions so that we do not get confused over whether a buffer cache entry is encrypted or not, however we will experience some performance degradation.

The hidden file system implements hard links, but these are only allowed within the same security level. An attempt to create a hard link across security levels will result in an `EXDEV` error code, the same error that occurs when hard links across devices are requested. This way, we prevent users from accidentally moving a file from a higher to a lower level instead of copying and deleting it, or create a link from a lower to a higher security level. Since inode numbers indicate security levels, a higher security level inode number that found its way into a lower level directory would indicate to an inspector that there are higher security levels in use. Users are responsible for not leaving traces of higher security levels (e.g., symbolic links, paths, log files, shell histories) in lower levels.

6   Performance

We have evaluated the performance of StegFS using the Bonnie [17] benchmark tool. The tests ran on an AMD K5 PR150 100 MHz processor, using a 1 GB partition of a Fujitsu 1.2 GB IDE disk. Bonnie attempts to measure real I/O speed by operating on files that are much bigger than the cache to render it ineffective. In real applications, access locality will lead to more cache hits and therefore better performance.

The table below compares the performance of the Ext2fs driver with the performance for normal and hidden files under StegFS, using a replication factor of 5 for both inodes and data blocks of hidden files.

| | Sequential Output | | | Sequential Input | | Random |
	Per Char [kB/s]	Block [kB/s]	Rewrite [kB/s]	Per Char [kB/s]	Block [kB/s]	Seeks [1/s]
Ext2fs	1835	3839	1964	2216	5476	31.4
StegFS normal	1628	2663	1761	2075	4872	31.3
StegFS hidden	44	45	10	374	420	2.6

Access speed to non-hidden files is roughly comparable to that for the standard Ext2 file system. A clear performance penalty is paid however for the high level of security provided for the hidden files.

We also performed a simple transaction write test, which wrote 256 bytes to a file, rewound the file position back to the start and called `fsync()` to commit all inode and data blocks to disk. Ext2fs managed to perform 45.10 of these transactions per second, while StegFS achieved 45.05 for non-hidden and 5.26 for hidden files.

The major reasons for the significantly lower performance of hidden files are the need to write replicated blocks, the encryption and decryption overhead, as well as the need to first check whether a block has been overwritten. Performance has greatly improved compared to earlier versions of the file system that spread block locations across the disk randomly.

We also performed some basic tests to verify the survivability of files while they are hidden. We first created with StegFS 250 hidden files, each 100 kB in size. We then unmounted the StegFS partition, mounted it again under Ext2fs and created another 250 files, each 100 kB long, this time openly visible. We finally remounted the file system under the StegFS driver and checked the integrity of the hidden files. Each test was repeated 100 times. With only a single copy of each hidden inode and data block, an average of 2.43% of the files were lost. With a replication factor of two, this fell to 0.08%. A replication factor of three, meant that only 0.005% of the files were lost. When four or more copies of the inode and data blocks were created, all of the files remained fully intact in this test. A larger number of copies will increase the survival chance of hidden files, but it will also reduce the disk capacity and the write performance.

7 Conclusions

We have created a practical implementation of a steganographic file system. It offers the following functionality:

- Users can plausibly deny the number of files stored on the disk.
- The confidentiality of all hidden file content is guaranteed.
- Deleting hidden or non-hidden files leads automatically to their secure destruction.
- Several layers of plausibly deniable access can be used such that lower layers can be voluntarily compromised without revealing higher ones.
- The existence of higher layers is plausibly deniable.
- The installation of the driver can be plausibly justified by revealing one lower layer and by referring to the additional security advantages provided by the product.
- A moderate amount of write accesses performed while not all hidden layers are opened is unlikely to damage data in hidden files.
- Write access to hidden files between inspections cannot be distinguished from non-hidden files that have been created and deleted.
- Non-hidden files continue to be accessible when the StegFS driver and its block allocation table are temporarily removed.
- The full Unix file system semantics are implemented.

Our project was inspired by [6], however the technique that we eventually chose, namely the use of an openly visible encrypted block-allocation table, differs from the two originally suggested constructions. Our system allows us instead to completely fill the disk safely when all hidden levels are open, and the only storage overhead comes from the adjustable replication of blocks. In addition, our scheme allows us to share a partition with a normal widely used file system, which simplifies installation and provides an additional degree of plausible deniability by making hidden files indistinguishable from unused blocks.

Like [6], we did not attempt to cover in this implementation the following functionality of a steganographic file system:

- Hidden presence of the steganographic file system driver
- High-performance write access to hidden layers
- Integrity protection of files
- Protection against the filtering of the entire disk content

Hiding the presence of the driver would require it to be attached, like a Trojan Horse, to another large and obfuscated application. The performance of write accesses might be somewhat increased by better caching of accesses to the block allocation table, but it will ultimately remain limited by the replication requirement that provides the survivability of hidden data during write accesses to lower layers.

References

1. Matt Blaze: A Cryptographic File System for Unix. In Proceedings of 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, November 1993, pp. 9–16. `ftp://ftp.research.att.com/dist/mab/cfs.ps`
2. Giuseppe Persiano et. al: TCFS – Transparent Cryptographic File System. DIA, Universita' Degli Studi Di Salerno, Italy, `http://tcfs.dia.unisa.it/`.
3. Encrypting File System for Windows 2000, Microsoft Windows 2000 White Paper, Microsoft Corp., 1998, `http://www.microsoft.com/windows/server/Technical/security/encrypt.asp`
4. Linux Kernel International Patches, `http://www.kerneli.org/`.
5. Peter Gutmann: Secure FileSystem (SFS) for DOS/Windows. Internet Web page `http://www.cs.auckland.ac.nz/~pgut001/sfs/`.
6. Ross Anderson, Roger Needham, Adi Shamir: The Steganographic File System. In David Aucsmith (Ed.): Information Hiding, Second International Workshop, IH'98, Portland, Oregon, USA, April 15–17, 1998, Proceedings, LNCS 1525, Springer-Verlag, ISBN 3-540-65386-4, pp. 73–82. `http://www.cl.cam.ac.uk/ftp/users/rja14/sfs3.ps.gz`
7. Michael Roe: Cryptography and Evidence. PhD thesis, University of Cambridge, Computer Laboratory, 1997. `http://www.ccsr.cam.ac.uk/techreports/tr1/`
8. Carl van Schaik, Paul Smeddle: A Steganographic File System Implementation for Linux, University of Cape Town, South Africa, October 1998. Software available on `http://www-users.rwth-aachen.de/Peter.Schneider-Kamp/sources/sfs/`.
9. Linus Torvalds, et al.: Linux 2.2 – Kernel. C source code, `http://www.kernel.org/`, 1991–.
10. Maurice Bach: The Design of the UNIX Operating System. Prentice Hall, 1986.
11. Rémy Card, Theodore Ts'o, Stephen Tweedie: Design and Implementation of the Second Extended Filesystem. In Frank B. Brokken et al. (eds.): Proceedings of the First Dutch International Symposium on Linux. State University of Groningen, 1995, ISBN 90-367-0385-9. `http://www.mit.edu/~tytso/linux/ext2intro.html`
12. Peter Gutmann: Software Generation of Practically Strong Random Numbers. In Seventh USENIX Security Symposium Proceedings, San Antonio, Texas, January 1998, pp. 243–257. `http://www.cs.auckland.ac.nz/~pgut001/pubs/random.pdf`
13. "AMAN" `<scramdisk@hotmail.com>`. ScramDisk – disk encryption software. `http://www.scramdisk.clara.net/`.
14. Brian Gladman. AES algorithm efficiency. `http://www.seven77.demon.co.uk/cryptography_technology/Aes/`.
15. Roger M. Needham, David J. Wheeler: Tea extensions. Draft technical report, Computer Laboratory, University of Cambridge, October 1997, `http://www.ftp.cl.cam.ac.uk/ftp/users/djw3/xtea.ps`.
16. David J. Wheeler, Roger M. Needham: Correction to xtea. Draft technical report, Computer Laboratory, University of Cambridge, October 1998, `http://www.ftp.cl.cam.ac.uk/ftp/users/djw3/xxtea.ps`.
17. Tim Bray: Bonnie file system benchmark, 1990, USENET newsgroup `comp.arch`, `http://www.textuality.com/bonnie/`