Abstract

This article is a summary of three different papers, which have in themself very good ideas which we merge into our project, thereby improving them (IMNSHO) ...

1 Shortcuts

Ross Anderson, Roger Needham, Adi Shamir	ANS (Theoretic Paper)
Andrew D. McDonald, Markus G. Kuhn	MK (Theoretic Paper)
HweeHwa Pang, Kian-Lee Tan, Xuan Zhou	PTZ (Theortic Paper)
Andreas C. Petter	AP (Developer)
Sebastian Urbach	SU (Developer)
John Williams	JW (Developer)

2 The Steganographic File System, ANS

2.1 Introduction

1. some scenarios which might make use plausible are enumerated
2. crypto keys may be kept temper resistant, but this is relative
3. documents may be shredderd, but remain intact by some other way
4. dual control for bank managers is done to protect their own families
5. victims might get rid of their attackers by plausibly simulate non-existance of data
6. ⇒ steganographic file systems
7. concept example:
   - victim keeps several records
   - e.g. email, tax, love letters
   - but doesn't want to know possible attackers about the existence of his trade secrets
   - so he keeps quiet about his trade secrets and attacker has no means of getting or even prove that they exist
8. assumption: opponent competent, hard- and software not tamper resistant

2.2 A Simple Construction

This chapter is left open, since it has no relevance for us

2.3 An Alternative Approach

1. **idea:** write random bits into all blocks of the harddisk and write the real data with some encryption key $\Rightarrow$ unused blocks and real data indistinguishable
2. collisions will behave as calculated by the birthday problem, meaning collisions will probable if more than $\sqrt{n}$ blocks are written
3. solution: replicate blocks and use redundancy to blocks to detect overwrites
4. then a formula for the probability for overwriting and replicating blocks is given (see the paper for details)
5. but practice might improve the results in 3 ways:
   (a) the file system might be organized as sort of a hash table, called Larson's table, therefore reading is fast (one disk access per block), but writing is slow (replicas have to be made)
   (b) there should be a linear hirarchy, where the higher levels know about the existance (and the passwords) of the lower levels, but (of course) not the other way round ... and the lowest level may be implemented more efficiently than the others, since the password may be revealed easily under compulsion
   (c) most of the user's file content will be recoverable
6. many other optimations are possible (orthogonal codes)

2.4 Relevance to Classical Steganography

1. classical steganography (writing to files, such as images, audio and video) has problem when it comes to big amounts of data and isn't transparently done (normally).
2. linear digital watermarks have a similar problem as stegfs: an attacker could mark several times, thereby destroying the watermark.
3. ANS suggest that the watermark is brought in redundantly

2.5 Conclusions

1. steganographic filesystem was introduced
2. 2 different were presented

3 McDonald/Kuhn (MK) Paper

3.1 Introduction

1. various crypto filesystems exist
2. attackers more persuasive if data owners data may not plausibly deny existence of data, so StegFS hides even the existence of data: “plausible deniability”
3. first design (using linear algebra) by ANS lacks the existence of many files and “it felt uncomfortable to MK, that the attacker must not know any plaintext”
4. the second design by ANS:
   - first wipe the n disc blocks with random data
   - write files to pseudo-random locations using some encryption function
   - Problem: files may be overwritten frequently, after $\sqrt{n}$ blocks have been used
   - therefore multiple copies have to be stored and a detection method for overwritten files has to be implemented
5. filesystem by Schaik and Smeddle lacks part of plausible deniability by marking blocks to be pssibly used by higher security levels, but files may not be overwritten
6. other Steganographic filesystems have been implemented

3.2 Basic Concept

1. approach by MK does not need separate partition, use of unused blocks within partition with normal files
2. MK use block allocation table
3. PROBLEM: installation of stegfs module visible $\Rightarrow$ good explanation must be given for what reason it has been installed
4. focus is on providing no signs for the existence of higher security levels not on the existence of the steganographic filesystem

5. overwriting all unused and therefore espc. hidden blocks will result in data data loss, but user should have backups placed elsewhere and normally would not wnat to hand data over
6. MK's stegfs has advantage that all standard unix filesystem features exist

3.3 Implementation

1. stegfs standalone driver between VFS and blockbuffer cache
2. stegfs partiotions look like ext2fs partitions where blocks have been overwritten
3. stegfs contains ext2fs driver for access to non-hidden files
4. when deleting normal files they get immidiatly overwritten by random data
5. small portion of normal files are positioned more randomly to plausibly explain block changes
6. stegfs is a modified ext2fs driver
7. user can use stegfsopen to open security levels
8. hidden files are accessed by traversing through root dir of security level (and subsequent subdirs) and then the data blocks
9. however, files may be overwritten, so even root dir has to moveable throughout the entire disc
10. and files and inodes have to be replicated to provide somewhat secure access

3.4 Block Table

1. block table stores encrypted checksums for each block to detect block overwrites
2. each entry 128bits long, encrypted with same key as corresponding blocks
3. `bchecksum` contains 32bits of CBC encrypted data
4. unused blocks get random values, so they cannot be distinguished from used ones

3.5 Inode Structures

1. the hidden stegfs inodes resemble those of ext2 but contain in addition the number of copys that were made of the inodes and data blocks.

2. the list of data blocks has 12 direct blocks, 1 indirect, 1 double indirect and 1 triple indirect block, like ext2.
3. ext2 has just a reference to a single data block, stegfs hidden inodes contains references to all copies of this block
4. stegfs inodes contains a sequence of several lists of inodes, each of which points to a different copy of all data blocks for a file.
5. boxes with the same letter represent blocks containing identical replicated data.
6. hidden inodes are 1024 bytes in size, equal to the ext2 default size.
7. each inode stores one disk block
8. Several copies of each inode are stored to provide redundancy
9. in the moment max 28 copies of each hidden inode and 14 copies of ech hidden file block can be stored.
10. the maximum number of copies of inode and data blocks is inherited from the directory in which the inode is created
11. the security leves which are used most often can have fewer copies since they are less likely to be overwritten
12. number of inode and data block copies 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

3.6 Virtual File System

1. Linux VFS uses generic inode and superblock structures in memory
2. both contain a file system specific section
3. StegFS versions of these extend those used by ext2
4. StegFS is using some additional fields for managing the security levels (e.g. keys for the current opened levels)
5. there is also a pointer to the cipher functions structure and the block table file
6. the VFS inode structur was extended, which has to hold the same informations as the hidden on-disk StegFS inodes like the replication factors and the locations of all replicated data blocks
7. the StegFS VFS is largen than the standard linux file system.
8. Kernel must be recompiled to install StegFS and it's not possible to install it as a loadable kernel module without any other kernel modifications.

3.7 Inode Numbers

1. An inode number is a 32-bit integer that uniquely identifies a file within a file system
2. in ext2, the location of a file's inode can be computed directly from its inode number
3. for hidden files on a StegFS partition, we habe to search for the inode in the decrypted block table and finde one which is not overwritten
4. in StegFS normal and hidden files are distinguished by their inode number so that operations ca be directed to either the ext2 or the the StegFS part of the StegFS driver.
5. inode numbers of hidden files indicate the security level from 1 to 15 (eg. — 0 — 1 — level rest of inode number—) such that the right descryption key can be selected easily while non-hidden files have inode numbers of the form : — 0 — 0 — rest of inode number —
6. to ovoid 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 togehter with a hash sequence number
7. when a new file is created the hash sequence number is increased until enough free blocks have beend found for all copies of the new inode
8. we search through the same sequence to locate inodes when they are loaded
9. decrypted inodes are cached within the VFS, so this search does not have to be repeated so often while a file is in use

3.8 Block Allocation

1. data and inode-indirection blocks in the hidden VFS are located at random disc positions
2. linux kernel supports random number source in form of the /dev/urandom driver, which maintains an entropy pool by hashing timestamps of interrupts
3. /dev/urandom is used to select the first free block that we allocate for a file
4. the first copy of ech following block is allocated in the next free block

5. additional copies ov every block are written starting from completely independent random locations, overwrite probability of the various copies remains independent
6. before a block is allocated we first test whether it is marked as used in the ext2fs bitmap, if not we are going to decrypt the entry in the block table using each of the known level keys
7. if for none of the keys are 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

3.9 Block Replication

1. multiple copies of inodes and data blocks are stored on disk to have redundance
2. for reading files we just need the first copy of any given block
3. when the checksum is correct, the file will be decrypted otherwise further copies will be tried
4. 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
5. when changes are made the corresponding initial vector is modified, the block is encrypted again and written to disk
6. the block allocation table has to be updated with the new initial vector and checksum
7. then we 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 or is still valid
8. if the block is valid, the we are going to encrypt the new data block for these locations and write it
9. 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
10. the we are going to encrypt and write the data into the block and update the inode and block table
11. 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
12. because we have to read every copy for the checksum test before we can overwrite something, it decreases the write performance. A future revision might cache the overwrite status of blocks in memory to get besser performance

13. blocks are replicated when data is written to disk
14. a simple method to check if all copies are written is to read and rewrite the file
15. new tool rerpl does this and should be used after the disk has been used in a lower security level
16. only a small amount of each block in the file needs to be read an rewritten to regenerate the number of copies
17. inodes are automatically re-replicated by accessing the file which updates the atime in the inode and causes to be rewritten

3.10 Key Management

1. each hidden file belongs to one of 15 security levels
2. to allow groups of security levels to be opened with a single passphrase, StegFS manages in addition 15 security contexts
3. each context gives access to a subset of security levels and is protected with his own passphrase
4. for every security level exists a passphrase, that gives access to this and all lower security levels
5. the user can add and remove any security level in any security context to build more complex hierarchies
6. when a user opens security context C and enters the corresponding passphrase PPC, then it is hashed immediately using a secure hast function h to produce HPC = h(PPC)
7. at the end of the block-table file, a 15 x 15 element security matrix M is appended, in which entry is a 128 bit word. If security context C is to provide access to security level L, then MCL = (SKL)HPC contains the level key SKL, encrypted with HPC.
8. all other matrix elements MC,L contain only a random value. The size of Matrix M is hardwired like the block table and is independent ot the number of security levels and contexts actually used, therefore the open presence of this date does not reveal to an inspector any more information than the presence of the stegfs driver itself
9. 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 a block i will be BKL,i = SKL+i

10. each block then is 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
11. in addition as long as fewer than 2 hoch 8 inspections take place, inspectors will rarely see two modifications of a hidden block with the same 16-bit IV.
12. current implementation offers two the 2 AES candidates Serpent and RC6 as block ciphers (Gladman's implementation) with minor modifications
13. architecture allow other ciphers to be added easily later
14. interesting alternative would be to use instead of AES-CBC a variable-length block cipher such as Block TEA, which would eleminate the need to store an initial vector
15. The SKL for all open levels L are stored in the superblock structure in RAM and these cleartext keys are never written to the disk
16. PPC or HPC values are overwritten in RAM as soon as the SKL keys have been decrypted

3.11 File System Usage

1. create a partition with a ext2 on it
2. then we have to fill the empty blocks with random data and create the block table file
3. the block table contains enough space to store enough encrypted keys for 15 different security contexts and levels
4. the 15'th is hardwired and not user configurable, so as to give users plausible deniability for the number of security levels for which memory was allocated
5. they can then claim that the software forced them to allocate 15 levels even though they needed 1 or 2
6. each context need a passphrase
7. you can now mount StegFS and you can also open security levels

8. 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
9. number of copies of each inode and copies of the blocks of a file can be controlled with tunestegfs
10. stegfsctrl allows to add and remove security levels from contexts
11. StegFS is under GNU GPL

3.12 Other Design Issues

1. buffer cache keeps hidden blocks only in their encrypted form, this ensures that other file systems never can see plaintext block contents
2. 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
3. this helps to retain consistency between funktions so that we do not get confused over whether a buffer cache entry is encrypted or not
4. the hidden file system contains hard links, but these are only allowed within the same security level
5. hard links (creation and requests) across security levels will result in an EXDEV error code
6. since inode numbers indicate security levels, a higher security inode number that founds it way into a lower level directory would indicate to an inspector that there are higher security levels in use

3.13 Performance

1. performance tested with bonnie benchmark software
2. access speed to non-hidden files is roughly comparable to that for the standard Ext2 file system. Hidden file access is much slower.
3. the major reasons for the significantly lower performance of hidden files are :
   - write replicated blocks
   - encryption and decryption overhead

    - first check whether a block has been overwritten

4. performance has greatly improved compared ro earlier versions of the file system that spread block locations across the disk randomly
5. file integrity test was done 100 times with every file
6. with only a single copy of each hidden inode and data block, an average of 2,43

3.14 Conclusions

1. this implementation of a a steganographic file system 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 amoutn of write access 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 temprarily removed
    - the full Unix file system semantics are implemented
2. we did not attempt to cover in this implementation the following functionality :
    - 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

4 StegFS: A Steganographic File System

4.1 Introduction

1. stegano blubber common to all papers, except that their design concept is directly mass market application
2. their steganographic file system should meet 3 major criterias:
   (a) no data loss
   (b) provide plausible deniability
   (c) minimize processing and space overhead
3. file information isn't stored in central place

4.2 Related Work

1. crypto file systems, steganography and steganographic file systems (ANS) are summarized, AK is mentioned as an implementation, new one mentioned by Hand ans Roscoe

4.3 StegFS: Steganographic file system

1. inode table for file object is saved in file itself

4.3.1 File System Construction

1. standard size blocks, bitmap as table for used blocks
2. hidden files not registered in central dir, but space allocated in bitmap
3. on creation disk is overwritten with random data, some percentage of blocks is abandoned, by marking the blocks as used (but in fact they are not used)
4. dummy hidden files to prevent successful examination of bitmap snapshots
5. hidden file consists of three components
   (a) link to inode table of the file
   (b) signature that uniquily identifies the file
   (c) linked list of pointers to free blocks held by the file
6. file is allocated using a hash value consisting of the access key and the file name + uid and searching for the first free block thereafter

7. file is found equivalently by searching for the signature of the file, which is computed equivalently to the block number (e.g. using SHA or MD5)
8. free blocks are allocated with every hidden file to be more secure against file system comparisons / block comparisons

4.3.2 Directory Support for File Sharing

1. integrates multi-user functionality
2. files shared by multiple users get random key (FAK), distributed to all users
3. several user access keys supported, for which each a directory of files and FAKs
4. does not have linear hirarchy (does not need it in fact, beacuse of bitmap marking)
5. FAKs enable file sharing, since the users might share them with otther users, enabling them to find and decrypt them

4.3.3 File System Backuo and Recovery

1. saving whole disk image would be too time and space consuming
2. saves only blocks, that are marked used but still saves other (publically readable) files normally
3. recovering first restores hidden files (because they contain encrypted inode information) and then writes the other files

4.3.4 Potential Limitations to StegFS

1. all hidden files must be backupped and restored together ⇒ restore to temporary volume and let users copy their files
2. filesystem unable to defragment hidden files
3. dead files by dead accounts may not be removed without the cooperation of the corresponding users

4.4 System Implementation

1. based on Kernel 2.4 an API was created
2. some relevant API calls are then enumerated

4.5 Performance evaluation

4.5.1 Experiment Setup

1. their StegFS filesystem will be compared to
   - StegFS
   - Steganographic filesystem with cover files (ANS)
   - Steganographic filesystem with hidden blocks (ANS)
   - freshly defragmented ext2fs
   - fragmented ext2fs

4.5.2 Effective Space Utilization

1. their StegFS performs very well when it comes to space utilization, since they don't need to replicate the data

4.5.3 Performace Analysis

1. their StegFS performs almost as fast as an fragmented disk, when it comes to write operations
2. it performs better than all other approaches when it comes to read operations

4.5.4 Sensitivity to File Access Patterns

1. their StegFS doesn't perform well in lightly loaded file servers (but this is common to all other steganographic filesystems)

4.5.5 Conclusion

1. will be extended to DBMS and P2P Networks

5 Ideas/Requirements

5.1 Sebastian Urbach

1. optimisation of stegfs based on the following list, commencing with security as top priority :
   - security
   - performance
   - functionality
2. easy to use graphical frontend to setup stegfs (based on Java?)

3. steganographic support :
    - Using standard steganographic methods or generate a complete new mehod ?
    - should stegfs support different methods ? Remember that we will support different ciphers as well ...
4. upgrade possibility from original StegFS ? (McDonald/Kuhn)
5. support different operating systems for Future ?
6. compatibility to original StegFS ? (McDonald/Kuhn)
7. availability of different package formats? (Eample .deb) I think its cool to support Gnu/Linux like Debian, but not commercial distributions like suse ...
8. We should explain in the final paper why we dont take the chinese system and why we take the principle from Andrew, remember the priority list above which demonstrate that the security is our first mission goal. Just explain why we made this fundamential decision.
9. We should think about the default stegfs settings when the user dont want to adjust them by himself :
    - number of file copies to write per procedure. I think we have to check out whats the default in Andrew's code. I thought it's 4 times or something.
    - crypto cipher (AES / Rijndael, maybe its besser to choose another one in the moment 'till its clear that theres no trouble anymore ?)
    - steganographic system (ample ambiguous, maybe some multimedia file stuff ? We have to check out which are well supported standard systems)
10. we should check every attack possibility against stegfs, especially unlikely ones like :
    - direct hardware ram access from outside the system
    - decrypted data which is plain readable in the system memory
11. whe should take a special care over the users an their security level passwords. Maybe it's better to say directly the user should take a passphrase instead of a password ? Every password (phrase) they'll use should have the following attributes :
    - it should be alphanumeric
    - it should have at least 8 characters

- it should contain at least one special character

12. we should take a look that users dont leave traces of higher security levels like symbolic links, paths, log files, shell histories etc.

5.2 Andreas Petter

1. security is highest goal / everything else is nice to have feature, only
2. basic concept ANS's second idea is good
3. it should not be provable that a stegfs filesystem has been installed
    - this includes, that no mapping file may be installed
    - since it is not normal that blocks with true random content exist, we need to simulate deleted files by simly copying random blocks
    - high fluctuation should be simulated by removing unused blocks from time to time by altering inodes and copying random, possibly unopened blocks and overwriting some other blocks
    - high storage integrety by saving many, many times... better ideas (10x ?)
    - perhaps sort of stegfs disseption feature; espc. useful would be digital watermarking steganographic means (many people have mp3s today) to avoid diskette or CDROM installed modules.
    - perhaps user level based module loading? How can this be done?
4. compress data before writing to disc to increase entropy and decrease number of known plaintext/ciphertext pairs
5. is it possible to use blind signatures to get provable deniability without overwriting files? Hm. Don't think so, but i'll check.
6. Proof sketch for highest security/invisibility (= pratical plausible deniability: meaning filesystem below has no knowledge about hidden files, even no knowledge if files are possibly written to this block), that file overwrites are unavoidable:
    - assume overwriting is avoidable
    - ⇒ filesystem must know where files/blocks are, that might store information to get 100% secure filewrites (therefore not overwriting files)

- ⇒ results in knowledge of very probable data existence (who installs some stegfs without the will of covering some data)
- ⇒ therefore security is not high enough for pratical plausible denieability

7. Another scenario for plausible deniability of entire harddisks might be when crypto is not allowed in a country e.g. old France and someone wants to encrypt.
8. give formula for probability that blocks get overwritten. This will be a multidimensional function. Print the result within the mkfs.stegfs program using the disk values from the user.
9. use of orthogonal codes (left to further improvements)
10. using CDMA techniques for file blocks may result in perfect security while retaining unavailability to detect which blocks have been occupied by data

5.3 John Williams

Usability: Some comments on the existing (ADM/MK) version from the user's point of view

1. Key management.

   The existing system is cumbersome. By default, security context N (accessed by a single password) gives access to security levels 1,2,..N.

   In practice, I think the usage will be something like this: I have some secrets that I want to keep from the police. So I set up security levels 1, 2 and 3. In level 1 I put my household account files: I give the password for this to my wife, and write it on a piece of paper. In level 2 I keep some pornography. In level 3 I keep the records of my arms sales to Iraq. When the police call, I explain that I was using stegfs to keep my accounts safe, and to stop my wife seeing my porno collection. I give them the keys to levels 1 and 2. I deny the existence of any more levels.

   When I'm using the system, I want level 3 to be open. There's no harm in having levels 1 and 2 open also. The original default system would allow me to do this by just entering the context 3 password. But this means that I don't usually enter the context 1 or context 2 passwords, which are the ones I intend to give to the police. So, when they are beating me, I may find it hard to remember these passwords. This may arouse suspicion.

   So my proposal is that the default key management system should work like this: to gain access to security level 1, you need password 1 to gain access to level 2, you need to open level 1 first, and then give password 2 to gain access to level 3, you need to open level 2 first, and then give password 3 ... and so on

Then, in normal use, I will enter all 3 passwords in order to access my files; when the police call, I will find it easy to give them the first 2 passwords, and I just have to deny the existence of the 3rd.

I've no objection to more complicated schemes for expert users, but I think this should be the default (of course, with up to 15 levels). I hope that this can be achieved simply by a small change to the pasword management code, without any major change to the main encryption code.

2. Replication of blocks and inodes.

   The default in the existing code is to keep several duplicate copies of each hidden data block and inode. This is very wasteful of disk space, and it gives no guarantee that hidden files will not be destroyed.

   If (in the above example) the police get access to levels 1 and 2, and find that there are damaged files, it is a sure sign that some other level exists. So I believe that this is a security threat.

   The PTZ paper proposes that this should be solved by marking the hidden blocks as 'used' in the free block list, but also randomly marking some 'abandoned' blocks as 'used', so that the hidden blocks appear to have been 'abandoned'.

   The problem here is that the proportion of blocks that appear to be abandoned (and so might possibly be used for hidden files) be fixed independently of the number of hidden levels; otherwise the police could deduce how many levels there were. Suppose that we set this propotion at 50for normal files, and 512M appears to be abandoned. Once we enter the level 1 password, we now see that 50actually available, so we have 256M for files in level 1, and 256M abandoned. When we enter the level 2 password, we see that actually there is 128M for files in level 2, and 128M abandoned, and so on. By the time we reach level 15, there is not much space for files.

   Of course the figures could be massaged by using a ratio other than 50a different ratio for each level. An alternative system of ratios would have given us 64M available for each of the 15 levels. But the ratios must be chosen without regard to how many levels are in use. So the disk space usage is always going to be very inefficient.

   I suggest that the original ADM/MK block allocation mechanism is adequate in practice. Going back to my original example, in normal use there is no reason why levels 1, 2 and 3 should not be open all the time, even when I am just doing my household accounts. So long as all the hidden levels are open, no files should be over-written even if the block replication factor is set to 1. Of course I would need to be careful that my wife does not edit the account files, as she only knows one of the passwords.

3. The block table file.

   The extra 'btab' file used by the stegfs filesystem is a nuisance. If it is needed, it should be a hidden file in the root of the filesystem itself, not a separate file outsize the filesystem (c.f. the journal file in ext3 filesystem, which is a hidden file in the root named '.journal').

4. Hiding the existence of stegfs.

   The proposal is that we should do some work to hide the factt that stegfs is installed on the machine; for example, by disguising the stegfs.o module as something else.

   This is the sort of thing that viruses and root kits do well. I don't believe that we should try to re-invent this from scratch. I think we should investigate the possibility of modifying some existing code in order to do this. I believe this could be done as a separate project, or as phase 2 of the stegfs project once we have got the file system code working.

Target systems and code portability:
I think our target systems should be linux 2.4 and 2.6, and Windows 2000 and XP.

We need to do a Windows version, because security should be available to the masses, and Linux is still a minority operating system. Windows 98/ME have different device drivers, but I think that we can ignore these as Micro$oft is phasing them out.

We need to do a Linux version first, because that is the starting point that we are coming from (and I, for one, do not want to use Windows). By the time we have got anythng working, linux 2.6 will be ready (or nearly so), and we don't want to get locked into an obsolete version for a second time; on the other hand 2.4 will not disappear straight away, so we should support that too.

I'm assuming that both the Linux and Windows version will use the same filesystem structure on the disk, and so will have the same code base. To make a Windows version based on the FAT or NTFS filesystem instead of ext2 would be a much bigger task.

For maximum portability, the code design should look something like this:

```
+----------------------------------------------------------------------------+
|                  Operating system call compatibility layer                 |
+----------------------------------------------------------------------------+
                                      |
                                      V
+----------------------------------------------------------------------------+
|                    Operating-system-independent interface                  |
+----------------------------------------------------------------------------+
|                                Main stegfs code                            |
+----------------------------------------------------------------------------+
                  |                                          |
                  V                                          V
+---------------------------------------+    +-------------------------------+
|  OS services call compatibility layer |    |   Non-OS compatibility layer  |
+---------------------------------------+    +-------------------------------+
                                                             |
                                                             V
                                             +-------------------------------+
                                             |    Non-OS-supplied services   |
                                             +-------------------------------+
```

Explanation of the diagram:

The main part of the code is written in an operating-system-independent way. Above this is a thin laye of code that converts the operating-system-specific calls into a common form. For example, Linux and Windows will both make calls into the stegfs code in order to open a file, but these calls will have different names and different parameters. The interface above the main code converts these different calls into a common form. Of course we may choose to make this common interface to be the same as the Linux interface, but this might not always be possible (e.g. if Windows has some features that are missing in Linux, or vice versa).

To do its work, the common code must then call back into other operating-system services in order to (for example) read and write disk sectors. Again, we write the common code to use a standard interface, and use a thin layer of code that is different between Linux and Windows, in order to convert this common interface to the different interfaces that Linux and Windows will supply.

There may be some services that Linux supplies that Windows does not (or vice versa); for example there may be no Windows equivalent of the Linux kernel module for AES encryption. In this case we would have to supply our own AES encryption code for Windows; we would make this fit the same common API that the Linux code supplies. This is the job of the box marked "non-OS-supplied services".