A Steganographic File System for the Linux Kernel

by

Ashley Morgan Anderson

The University of Exeter

COM3401 Individual Project

11th August 2007

Abstract

Encryption is all around us these days, yet the security of private data on home computers is overlooked by many. The presence of cipher-text — in some instances — can make things worse, therefore the only way to overcome this is to hide the data. The Ancient Greeks knew about this risk and as such fabricated some of the most ingenious methods of steganography ever used. However, steganography has come along way in the last 3000 years, and is more important today than ever before (due to tightening restrictions governing encryption). This paper looks at how different mediums have been used to ensure messages and data remain private, as well as proposing a new file system which takes care of hiding files for the user. Whilst not the first, it does include many unique features which are intended to provide a more natural interface.

I certify that all material in this project which is not my own work has been identified.

. . . . . .

A.M. Anderson

Contents

List of Figures

1 Introduction

Steganography is the art (or science) of hiding sensitive information in such a way that its existence cannot even be proven[1]. The word steganography is derived from Ancient Greek and literally means "covered writing". Throughout history many different techniques have been used to hide messages, including — but in no way limited to — invisible inks, tattoos hidden under a head of hair, microdots and wax tablets. Now as we entered the digital age, a new opportunity has arisen for hiding data which was previously impossible. Computers enable information to be hidden within documents, data or even programs, using increasingly complex algorithms, thus becoming less susceptible to detection.

The need for steganography came about to prevent the interest which accompanied coded or enciphered messages. For instance, if a censor was to come across a message which was a single, almost meaningless, sentence, or random list of letters, number or other characters, then he can be almost sure that what he has is some form of encrypted message, and until within the last thirty years or so, all forms of encryption have eventually been broken by cryptologists the world over (substitution cipher, Vigenère Cipher, et cetera). It is only that modern encryption techniques are complex enough to ensure that a standard dictionary or brute force attack will take an unfeasibly long time.

§2.1 looks at how steganographic techniques have changed over the last few thousand years — ranging from some of the earliest stories of the Ancient Greek world, all through Medieval Europe right up until the Second World War. Some of the methods are bizarre, and others are still used today, albeit in a digital form. Some early methods of sending a message without incurring the penalty of detection relied upon invisible inks — writing the real message between the lines of an innocent cover message. Obviously sending a blank sheet of paper with your invisible message written on it will arouse just as much suspicion as any encrypted message. Some people even went so far as to tattoo the message onto the messengers head, then wait for the hair to grow back before sending the messenger on his way ($2,000$ or so years ago the Ancient Greeks may not have thought speed to be a huge factor in relying information)[2].

The digital age of steganography can be seen as two different sub-topics: techniques for hiding computer files in other computer files, and how a file system can be used to hide the existence of all the files stored on it. §2.3 looks at existing steganographic file systems and the features they implement. One of the ideas steganography tries to implement, especially with modern techniques, is plausible deniability: any act which leaves little or no evidence of wrongdoing or abuse. The term was first coined in the United States, during the 1950's when "black ops" were carried out without the authorisation of the President, thus when questioned about these operations, he is able to deny his involvement truthfully[3]. When this idea is used within computers it generally refers to data for which the owner can deny its existence. It is this idea of plausible deniability that separates digital encryption from digital steganography. Using encryption on its own is not without its drawbacks: when a message is encrypted, the existence of the message can clearly be seen; whether this is the cipher-text used hundreds of years ago, or the pseudo-random bits which are today's digital "cipher text". Being able to hide the encrypted message adds another layer of security. This can be as simple as a null cipher being used to hide a message encrypted with a Caesar-shift substitution cipher, or as complex as a file first being encrypted using the Blowfish encryption algorithm, and then hidden inside a JPEG[1] image using something akin to least significant bit insertion.

[1] Joint Photographic Experts Group image format. JPEG is a standards committee that designed an image compression format. The compression format they designed is known as a lossy compression, in that it deletes information from an image that it considers unnecessary.

All files in a computer system are stored on disks in a logical and sequential format so that retrieval is easy and as fast as possible. File systems are a fundamental part of any operating system, allowing users to keep data once the system is switched off and different operating systems use different formats for storing files and have varying support for other file systems. Some systems only support a small number of file systems, usually those created by the operating system provider, whereas others are able to read and write to many different file systems formats, typically through an abstract layer known as the Virtual File System[4].

Despite numerous methods for preventing unauthorised access on many home computers, and increased awareness to the threat of viruses, worms and phishing attacks, many people still ignore social engineering and physical attacks where an attacker is able to access data on the machine from the desk were the machine is sitting. Even using a screensaver with a password is not enough to stop somebody who is determined to steal sensitive data. BIOS[2] passwords are seldom effective as they can be reset by temporarily removing the BIOS battery from the motherboard. Passwords are also useless if someone steals the disks, as all the files on the stolen disk will be accessible to whoever subsequently connects the disk to another system, without and restrictions. At this stage of an attack, having used an encrypted partition might seem like the final line of defence, yet a motivated attacker can use many different techniques to convince you that surrendering the password key used to encrypt the partition is in your best interests.

To counteract this threat, the principle idea behind steganographic file systems (steganography as a whole in fact) is plausible deniability. If the existence of files on the file system can be denied without the risk of being uncovered, then a user can reveal unimportant files yet remain silent about those which are more important. Unfortunately for security conscious users, there are only a few steganographic file systems yet they are no longer being actively developed. Therefore this project aims to result in a working steganographic file system which is simple and effective, allowing development of the file system to continue and prosper.

2 Background

2.1 Steganography

History has shown us many different techniques that have been used to ensure that secret messages have reached their intended recipient without being tampered with along the way. Initially secure communication channels were used only by those in power, be it in charge of a country or commanding an army; these days however, more and more people are becoming aware of the threat to their own privacy that they feel the need to use (more commonly) encryption, and (more recently) steganography to overcome this threat[5]. Many corporations are also beginning to value steganography as a means to protect their innovative design ideas. The following example shows how power could be abused by one company seeking to gain information about another's upcoming products: An employee may be arrested on suspicion of a crime and asked to hand over a company laptop as evidence (this laptop may contain one or more encrypted partitions to keep either private details or company secrets safe) and required to provide the password to access the data else the disk will be used as evidence of guilt. This could be the result of the rival company bribing the police to acquire the employee's laptop and expose any trade secrets found in files on the disks.

[2]Basic Input Output System — The set of routines stored in read-only memory that enable a computer to start the operating system.

There are many different techniques for hiding messages that have been used throughout history. The Ancient Greeks were among the first recorded civilisations to use steganography when sending messages. Perhaps the most drastic method they used was the combination of slaves and tattoos: a slave's master would have the message tattooed upon the shaven head of his slave. Once the hair had grown back, and the message was hidden, the slave could be sent on his way — all he had to do at the other end was shave his head again and point it at the intended recipient.[3].

The Ancient Greeks had many other methods for hiding their messages — perhaps the most popular was the use of wax tablets. These tablets were made from small boards of wood which were covered in wax; the idea being that a message could be written on the wax until it was no longer needed and then the wax was scraped off, melted and reapplied. The Greeks eventually realised that they could send hidden messages to each other by writing their intended message on the wood itself before covering it with wax.[4] Eventually the remainder of Europe caught on to the whole "encryption" band-waggon and began developing ciphers of their own. it is unknown exactly when it happened, but at some time in history the idea of null ciphers began: the message to be hidden is spelt out within the body of the cover text. Such as using the first, second... last letter of each word, or perhaps after a punctuation mark. The use of pin holes under the required letters is another technique, which does not require both sender and receiver to agree prior to sending the message which method to use; it essentially allows the sender to disregard any systematic method for choosing which letters of each word/sentence to use.

> Apparently neutral's protest is thoroughly discounted and ignored. Isman hard hit. Blockade issue affects pretext for embargo on by-products, ejecting suets and vegetable oils.

The above passage contains a message which can be found by extracting the second letter from each word. The resulting message was sent by a German spy during World War 2[6], it reads:

> Pershing sails from NY June 1.

The use of null ciphers has not halted as we have entered the digital age: least significant bit insertion (a common method for hiding files inside media) is essentially a digital form of the null cipher. Another tried and tested method for writing a hidden message was to employ the use of an invisible ink. By the end of The First World War, almost every invisible ink known today had then been discovered. It was at that time also noted that inks of this nature were also insecure. There are many liquids that can be used to write an invisible message, most citric juices for example. These, plus milk or liquids high in sugar content (honey, cola or similar, or just sugar water), can easily be developed with the use of heat. Others require the use of a specific chemical to reveal the hidden message — such as red cabbage water to reveal a message written in vinegar, or a message written with copper sulphate can be exposed using ammonium hydroxide.

During the years between WWI and WWII Germany developed microdot technology. This process involved photographing the document to send and decreasing the size of the photograph until the final photograph was the size of a typographical dot, such as the tittle of a lower case

[3]Herodotus (an Ancient Greek scholar) recounts the story of Histaiaeus, who wanted to encourage Aristagoras of Miletus to revolt against the king of Persia

[4]Again, this is a story documented by Herodotus involving Greece and Persia: Xerxes of Persepolis was amassing an army to expand his empire by conquering Greece; this was witnessed by Demaratus who used the wax tablet method to warn the cities of Athens and Sparta.

i or j. Whilst all references to microdots infer that the microdot was of a document, it should be noted that the document could well be encrypted beforehand to hinder the extraction of information once discovered[7].

2.2 File Systems

There are many different file systems in use today, each offering a different set of features — some are suitable for larger capacity disks, whilst others can be used to access files over a network yet cause the files to appear to be local to the user. Different systems implement various file systems as default. The Microsoft Windows line of operating systems only support file systems written by Microsoft. However, systems which have their roots in UNIX such as the Apple range of operating systems, the Linux Kernel and BSD (Berkeley Software Distribution) — among other — use an abstraction layer to interface with files on disks, this layer is known as the Virtual File System[8]. The VFS allows support for any file system which adheres to the File System Switch Functions[5]. Figure 1 shows how a user's request for file system information (such as reading from a file) is processed by the kernel.

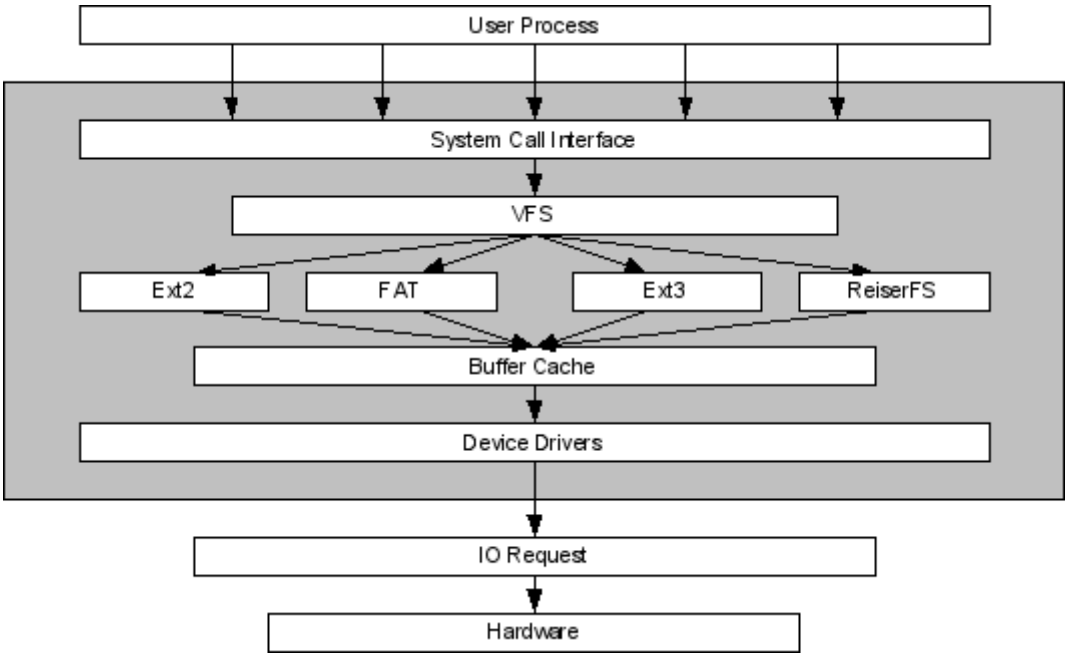

Figure 1: The Virtual File System[9]

The two file systems which are used by Microsoft are FAT (File Allocation Table)[10] and NTFS (New Technology File System)[11]. Each of these file systems have had numerous enhancements over the years; FAT became FAT12 then FAT16, finally stopping once it was known as FAT32, or more commonly VFAT. The values refer to the number of bits used to address the data blocks, and it is clear that when using more bits, more data blocks can be referenced, thus the capacity of the partition can be larger. In theory, the initial FAT implementation was

[5]A list of these functions can be found in Appendix A.1.

limited to 4,077 files within a 32 MiB partition and file names could only contain 8 characters plus a 3 character extension, over a period of 20 years, Microsoft improved and updated FAT to support increases in disk capacity.[6] Figure 2 shows how simple the series of FAT file systems are, with the file allocation bitmap towards to head of the partition. It is worth noting that the table itself only stores whether or not a block is in use, not which file is occupying it, as these details are extracted from directory entries with the data block region. When Microsoft

Figure 2: FAT File System Structure[9]

Windows NT was released, Microsoft had designed a new file system called NTFS. They have slowly made it the default file system for all of their operating systems since Microsoft Windows 2000. NTFS has many advantages over FAT, not limited to increases in file capacity ($2^{32}-1$ files and 16 TiB[7] maximum volume size). NTFS supports user quota's[8] as well as transparent volume compression and encryption allowing available storage space to be better utilised and protected against prying eyes. With the release of Microsoft Windows Vista, Microsoft introduced transaction functionality, whereby writing data to multiple files is treated as a single thread; thus if one write process fails, the entire transaction fails. As with transactions in database environments, other processes will only see the changes after they have all been committed.

Non Microsoft operating systems have, for the most part, drastically different file system designs; with many more file attributes and permissions. Many have their roots set in UNIX, with the advent of the UNIX File System (UFS)[12]. Typically, files stored on UNIX based file systems have permissions governing who can read, write or execute them. Unlike FAT or NTFS, not only are directories also considered as files, but devices and interprocess communication pipes are also treated as files. Obviously it makes no sense to open a device in any kind of editor, but permissions can be set in the same way to allow or disallow access.

Each of the major UNIX based operating systems uses a different default file system: Apple's Mac OS X uses Apple's Hierarchical File System Plus (HFS+), where as the BSD forks[9] and Linux distributions allow the user to select any supported file system during the installation. The Linux kernel natively supports the Second Extended File System (Ext2)[13] and its descendants: the Third Extended File System[14] and the Forth Extended File System (referred to as Ext3 and Ext4 respectively). There is also support for many other file systems built into Linux, including MinixFS, ReiserFS (and Reiser4)[10], JFS[11], and XFS[15][12].

Many of the UNIX-like file systems follow a similar structure (as did FAT and NTFS), however because of the extended set of permission and file attributes, UNIX file systems and Microsoft file

[6] Appendix A.2 compares some of the standard file systems referenced throughout this report.

[7] For details on volume capacity and SI equivalent units, refer to appendix A.3.

[8] Quota's limit the volume of disk space a single user can use, to stop disks becoming full and inconveniencing others.

[9] FreeBSD, NetBSD and OpenBSD — each aims to achieve a different set goals: FreeBSD focuses on high performance, NetBSD aims to be as portable as possible (running on many different architectures and chip-sets), whereas OpenBSD focuses on security, and aims to be the most secure operating system "right out of the box". (Check http://www.freebsd.org http://www.netbsd.org or http://netbsd.org for more details.)

[10] Details about the different versions of Hans Reisers' file systems can be found at NameSys (http://www.namesys.com)

[11] JFS — or Journaled File System — is a journaling filesystem created by IBM

[12] Additional details, including the original XFS design specification are available from Silicon Graphics Inc. (http://oss.sgi.com/projects/xfs/)

systems are very different from each other. Figure 3 is a simplified example of how UNIX based file systems structure the data. Each file has its own inode which contains necessary metadata

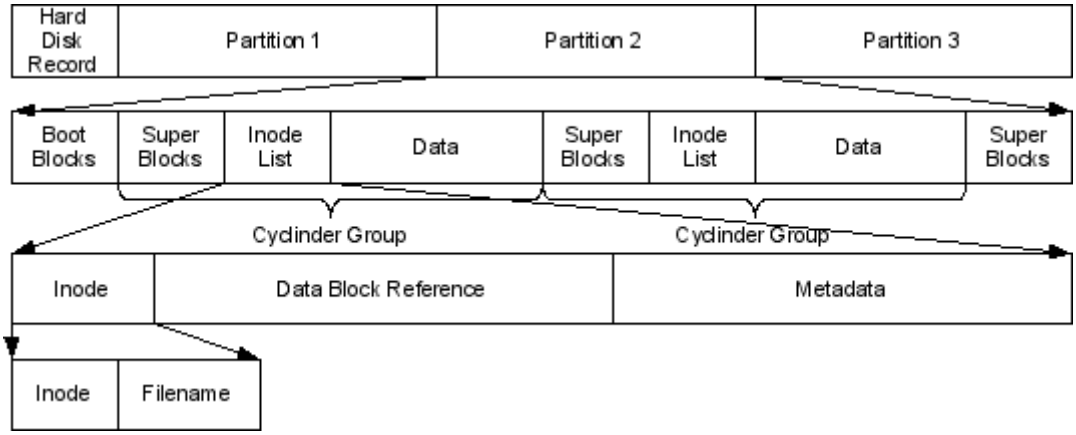

Figure 3: UNIX File System Structure[9]

to identify who owns the file and who else has permission to access it, as well as where the file resides on the hardware.

Many file systems in widespread use today are what are known as a journaled file systems. The birth of journaled file systems came about because of the risks and problems involved with file system recovery after a system or power failure. Such an event can leave data structures in an invalid intermediate state. For example, deleting a file from a UNIX based file system involves two steps:

1. Removing the files directory entry
2. Marking the files inode as free space

If step 1 occurs just before a crash, there will be an orphaned inode and hence a storage leak. On the other hand, if only step 2 is performed first before the crash, the not-yet-deleted inode will be marked free and possibly be overwritten by something else. There are two methods to ensure data integrity: the first is to perform a complete walk of the file system's data structures when it is next mounted. This is similar to how `fsck` works on UNIX-like systems, unfortunately this can be very time consuming on large volumes. The second is to keep a journal of all changes made on the file system. Recovery is now as simple as reading the journal and checking the entries match the state of the file system. Popular journaled file systems include NTFS, Ext3, ReiserFS and HFS+.

2.3 Steganographic File Systems

Initially, digital steganography involved hiding files inside of other files, which will always have its limitations, the obvious restriction is size. As larger files need to be hidden, there is only so much compression that can be achieved, and only so big cover files can be before it becomes inconvenient. The evident solution to hiding files in files is to alter the way files are stored on any given medium; the file system. There are many different file system implementations, and

all have their advantages and disadvantages. There are even encrypted file systems — such as TrueCrypt[13] — yet again though, the presence of cipher-text remains. (With cipher-text yet to be decrypted, an attacker can continue to demand passwords from a user, and with sufficient motivation, is likely to do so.) File systems have many similar basic file related features: besides the name of the file and its size, there are file permissions, who owns the file and who else can read or modify it. All UNIX based file systems have these basic attributes (as shown by a simple '`ls -l`' from the terminal in figure 4): type of file, access permissions, number of hard links, owner, group, size, creation time/date, modification time/date, and name. A steganographic file system needs to store all of this information without giving away the existence of any of the files, as well as where to look on the device itself for the data without actually knowing itself. Successfully storing all of this metadata allows the steganographic file system to integrate seamlessly with the host operating system and all applications which look for the metadata to determine how to act on a particular file when encountered.

```
brw-r--r--   1 user group 64,  64 Jan 27 05:52 block
crw-r--r--   1 user group 64, 255 Jan 26 13:57 character
-rw-r--r--   1 user group     290 Jan 26 14:08 compressed.gz
-rw-r--r--   1 user group  331836 Jan 26 14:06 data.ppm
drwxrwx--x   2 user group      48 Jan 26 11:28 dir
-rwxrwx--x   1 user group      29 Jan 26 14:03 executable
prw-r--r--   1 user group       0 Jan 26 11:50 fifo
lrwxrwxrwx   1 user group       3 Jan 26 11:44 link -> dir
-rw-rw----   1 user group     217 Jan 26 14:08 regularfile
```

Figure 4: An example of the types of file stored on a file system

There are two recognised methods for a steganographic file system, both proposed by Anderson, Needham and Shamir in their paper[16], and they have one major similarity in their design: the idea of different levels of security. Files are stored in levels much like files are stored in directories on standard file systems, with each level requiring a password to access it in some way. The first of these methods makes no assumption about the use of a strong encryption algorithm to obscure the data. It aims to provide protection using linear algebra by requiring anyone who wishes to access a file to know the name of the file exists, and the password used when the file was encrypted and written to disk. Without satisfying both of these preconditions an attacker can gather no information about whether such a file is present — unless he already knows some of the contents of the file or tries all possible file name/password combinations.

The idea is to have a number of cover files already on the file system that are initialised as random data, and then embed the user's files as the exclusive or of a subset of the cover files (the subset is chosen by the user supplied password). If there are m cover files ($C_0, \ldots, C_{m-1}$) and they are all of the same length, and the user inserts a file F with password P. Then select the bits from the cover files such that C_n for the n^{th} bit of P is 1 and perform a bitwise exclusive or; which is — in turn — XOR'd with the file to be hidden, F, the results of which is again XOR'd with one of C_n. This process results in the user's file F now being the exclusive or of the subset of the C_n selected by the non-zero bits of P. Symbolically expressed[17] as:

$$F = \bigoplus C_j$$

$$P_j = 1$$

[13] TrueCrypt is available from SourceForge.net (http://sourceforge.net/projects/truecrypt)

One important aspect of this technique is that if access is linearly hierarchical (storing a file at a given security level requires knowledge of all passwords for all files stored at lower levels) then files can be added in such a way that they do not disturb existing hidden files.

To ensure that the implementation of this design provides the plausible deniability that is sought the user needs to use a number of levels with a number of passwords, ideally more than two or three. As of this stage in the design, the data on the file system is not encrypted, only masked with a cover file. This is essentially the same as using a simple mono-alphabetic substitution cipher, or at best a Vigenère cipher, which these days provide no protection against even a frequency analysis attack. Linear algebra also demonstrates that if the password is n bits long, and an attacker knows more than n bits of plain text, then after obtaining all of the files he can write n linear equations in the n unknown bits of the key. For example, a tax inspector might know where to look in a file to find a tax reference number, and this could be enough to break the system[16]. Hence it is assumed that an attacked has no prior knowledge of what could be stored in the system. However, the damage from this kind of attack can be limited by restricting knowledge of what exists on the file system. There is a simple solution to this problem: to pre-encrypt all plain text data before being written to disk, using a key derived from the password. This method's implementation of levelling files allows a user to reveal x number of levels (on tax payment manipulation, affairs, et cetera) and thus the files under them, yet remain quiet about the remaining y levels (on industrial trade secrets).

This construct contains one more major flaw: performance. Reading or writing a file would involve reading or writing fifty times, and if this were to access a 10 MiB file then it is the equivalent of accessing a 500 MiB file on a standard file system. This issue has arisen because the design, in its initial state, has a complexity of $2,100$ in guessing linear combination. This complexity also provides the file system with a hierarchy with 100 levels in which to hide data. As this is likely to be too large it is propositioned that the complexity is reduced to 28 (or 251 — being the largest single byte prime), therefore using sixteen cover files and providing eight levels in the hierarchy. Ultimately this still results in a sixteen-fold read/write penalty.

The second construct makes a different initial assumption: the existence of a good block cipher, for which an attacker can not distinguish cipher-text from pseudo-random data. This infers that the presence or absence of a block at any location cannot be determined without knowledge of the encryption key. Unfortunately if the usage of a block cannot be ascertained then the concept of implementing numerous security levels does not hold. Even if files are written to pseudo-random blocks — calculated from a one way hashing function derived from the password used during encryption — eventually after very few blocks have been written "collisions" will occur (where data written begins to overwrite previously written data). This behaviour is because of the birthday theorem[14]. After only $\sqrt{m}$ blocks are written to a file system, with a total of m blocks, existing files will be corrupted by subsequent writes. An obvious solution to this problem is to write a file to multiple pseudo-random blocks. Again, there's a catch: imagine that m blocks are written, and each block is written n times. Now consider the probability that any given block will be overwritten on a subsequent write; if there are M of these then the total number of unique blocks being written n_i times from all blocks N is[17]:

$$\pi(n_1, n_2, \ldots, n_M) = 1/N^M \left(\begin{array}{c} N \\ n_1, n_2, \ldots, n_M \end{array} \right) \frac{M!}{\prod_{i=1}^{M} (i!)^{n_i}}$$

[14]In probability theory, the birthday theorem states that in a group of 23 randomly chosen people, the probability is more than 50% that at least two of them will have the same birthday. For 60 or more people, the probability is greater than 99%, although it cannot actually be 100% until there are over 366 people.

which can be approximated for computational purposes as:

$$\pi(n_1, n_2, \ldots, n_M) \simeq 1/(N^M e^{M^2/2N}) \frac{M^{M-n_1}}{\prod_{i=1}^{M}(i!)^{n_i} n_1!}$$

so the total number of unique blocks being overwritten is:

$$\Phi(M, N) = \sum_{1}^{M} \pi(n_1, n_2, \ldots, n_M)$$

where the sum is over all possible combinations of the n_j. Therefore the probability that block k_j is not overwritten is calculated from:

$$1 - [\Phi((j-1)m, N)]^k$$

Thus the probability of a file K, being overwritten is:

$$p = \sum_{j=1}^{K} [\Phi((j-1)m, N)]^k$$

There exists no analytical solution for the above equations, nevertheless in practice if the disk is considered to be "full" the first time a corrupt file is read back — all locations reveal overwritten data — then the load factor[15] of the system will be about 7%. This result is somewhat tentative, and can — in practice — be improved upon by allowing a greater percentage of blocks to become corrupt — 10% corruption provides a 20% load factor.

As with the first method there is a read/write penalty, but it is now only five times, instead of from sixteen upwards. However this design idea utilises a Larson table[18] which indicates which block to seek out first. Larson table's were designed to allow only one disk access to read from a database. In this construct, the use of a Larson table enables — at any given security level — the first uncorrupted location of each file. Likewise when the file is written back to the file system, this block can be the first to be written to. This list can be generated when a user first enters a new security level and can reside in memory for the duration; when the user leaves the level for a lower level the table in memory is discarded. Therefore there is only a small performance penalty for files which the user accesses immediately upon entering the new security level, and if for a particular file all blocks are corrupt then a warning message can be returned to the user that the file system is (nearly) full. Experiments by Larson and Kajla showed that the disk would appear full once $80 - 90\%$ of the blocks were occupied. When files are written to the disk the block stored in the Larson table is the first to be written, with the remainder being written in the background, unless the user immediately descends to a lower security level.

Figure 5[19] shows how this second method for a steganographic file system implements the idea of security levels. Each white rectangle is an inode pointing to a file — where some of the files are in fact directories and therefore deeper security levels which point to more files, and so on. Blocks of the same colour represent the same file being replicated at different hardware addresses.

There are already numerous existing steganographic file systems, all inspired by the paper from Anderson et al. and the initial implementation by McDonald and Kuhn; they all implement the latter of the two design ideas in their own way. The first implementation of the scheme

[15]The ratio of total unique blocks stored against the total space allocated.

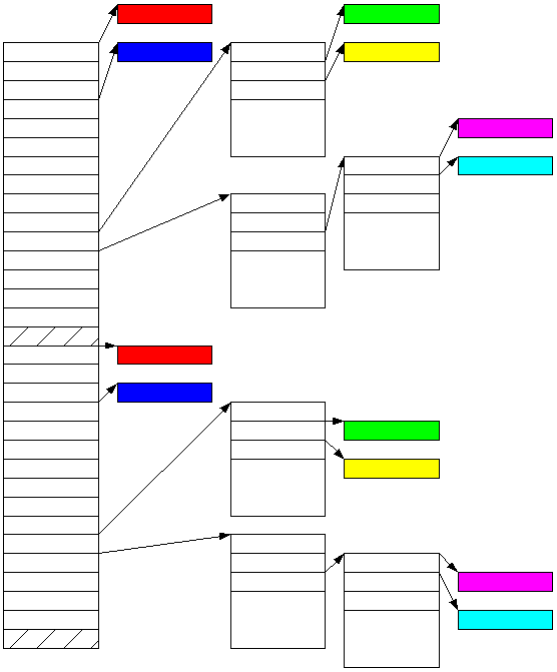

Figure 5: An inode structure showing different files and varying security levels

was StegFS[19]. The second NSteg[20], attempts to eliminate the need to replicate the same file multiple times. It uses a bitmap to set which blocks have been allocated, however this then removes the plausible deniability by proving that there are files in such blocks. There is a second StegFS, which is based on the two previous StegFS implementations[16]. It aims to build upon the work by McDonald and Kuhn. Magikfs[17] is a fourth steganographic file system, based on the work from[20].

Unfortunately, many of these projects appear to be no longer under active development. StegFS[19] was last updated in 2004, and NSteg is even older (2003); lastly, authors of StegFS number two are appealing to anyone who wishes to take over project development. All of the proposed projects that have released a file system driver have written kernel modules to work with the virtual file system (VFS), alongside other drivers such as Ext2fs. (Magikfs is a FUSE[18] module.) It is also unanimous that the resulting steganographic file system resemble a standard UNIX as much as possible: with support for multiple users, fully supported file permissions, et cetera. This ambition has driven project authors to start with Ext2 and add the steganographic functionality. Ext2 has advantages over Ext3 for use in a steganographic system as file operations could be journalled thus hindering the claim of plausible deniability therefore compromising security with journalled data becoming accidentally stored at a different level to the file it refers to.

The one major difference between each proposal was how to determine whether a block is being used by a file. To fulfil the plausible deniability aspect, looking at the file system bitmap

[16]StegFS (the second) is available from SourceForge.net (http://sourceforge.net/projects/stegfs)
[17]Magikfs is available from SourceForge.net (http://sourceforge.net/projects/magikfs)
[18]FUSE is a Kernel module which allows file system code to be executed in user space instead of kernel space — this is explained in greater detail in §3.2

should not suggest whether a particular block is being used, however if a block is defined as either not used or "maybe used" then this adds some ambiguity. NSteg does just this: when the file system is first placed on a device a number of random files are stored, and further inodes are marked as occupied, in a attempt to allow the user to claim that the file system did it and no real files have been hidden. Magikfs uses a similar technique: users are allowed to write files without setting the block as used. This does add the possibility that files could be overwritten without there being a replicated copy elsewhere on the system. There is a trade-off to be made: increased plausible deniability on one side, increased file integrity on the other.

3 Design

3.1 Implemented Design

Before any of the coding and implementation could being, a solid design foundation was necessary to allow me to progress in a definitive direction. Over the course of many weeks the initial design was modified and eventually evolved to provide a suitable level of steganography whilst at the same time simple enough that writing the code would not have become an infeasible task. Like the steganographic file systems which have preceded my own, it is a file system driver — or module — for the kernel. This allows support for the file system to be loaded and unloaded as necessary using the same techniques as any standard file system, as well as allowing for standard libraries and functions to interface with files on my steganographic file system as they would with files on any "normal" file system. By having the file system supported in this way, users notice no difference with how they store and access data.

Where as all of the existing steganographic file system implementations have been built on top of the Second Extended File System to further emphasise the likeness between themselves and standard file systems, I choose to use the heavily simplified file system, *Uxfs*[19][21], as I deemed many of the features of Ext2, specifically those which optimise read/write performance, to be unnecessary in a file system where performance usually comes at the cost of security. This factor is more true whilst observing writing data to a file, where several copies of the data need to be written to account for possible corruption, yet equally visible when the current copy of a file *is* corrupt and thus a non-corrupt copy needs to be found.

The number of copies of each file was also a decision I needed to make — to few would mean that the risk of loosing data would be increased, whilst too many would cause an unnecessary decrease in read/write speeds — a trade-off between data integrity and I/O performance. Following the rational in [20] the number of duplicates should be within the range $8 - 16$, therefore I decided to make 7 additional copies of any data written to disk (totalling 8 instances of each file); allowing for a combination of security, performance and simplicity. Each additional copy is placed at a calculated offset from the first, in a similar way to the first inode: the first inode is the combination of a simple one-way hashing functions on the key used for encryption and the name of the file. This gives a numerical value which can be scaled down to within the range of inodes and blocks available on the file system. Thus if there are 256 possible inode locations, and the hash value of the key and name is 711, then 256 is subtracted from 711 until the value is lower than 256:

$$711 - 256 = 455$$

[19] Uxfs is a simple file system which allows basic file system commands: making/reading/writing and deleting of files, in addition to manipulation of a hierarchical tree structure.

$$455 - 256 = 199$$

Therefore the first instance of the inode data would be placed in inode number 199. Subsequent inode locations are calculated by adding the initial value (in this case 199) to itself minus 1:

$$199 + 198 = 397$$

$$397 - 256 = 141$$

This keeps going until a distinct inode locations have been written to:

$$141 + 198 = 339$$

$$339 - 256 = 83$$

For this particular example the inodes which contain the file information are (in order of calculation): 199, 141, 83, 25, 223, 165, 107, and 49. Block locations for the file data itself are calculated in a similar fashion.

All of the previous steganographic file systems make use of hidden levels to protect increasingly sensitive data. (Likewise, I too have implemented this, otherwise all I would really have is a overly-redundant encrypted file system.) My method for creating new levels differs from all of the other techniques — such as a `mklevel` command, or using a pre-constructed level hierarchy — by furthering the notion of hidden files. Traditionally on UNIX-based file systems any file which starts with a dot (".") is considered to be hidden. These files are usually configuration files and directories within a users home directory, and are not really hidden just ignored by the standard `ls` command; using `ls -a` will show them. My intention was to turn hidden directories into the hidden levels, whereby allowing the existing `mkdir` command to create the next hidden level of the file system.

Prior to accessing my steganographic file system, the underlying structure needs to be place, just like any other: using `mkfs`. The Uxfs example came with its own `mkfs`, however this obviously needed to be modified to account for the newly added enhancements. As all of the data written to the file system would be stored as cipher-text, any blocks which were not occupied needed to appear as pseudo-random, to hide the the presence of any encrypted data. During the `mkfs` stage this would require that instead of zero's and necessary header information (magic number[20], et cetera), all bytes needed to be randomised. Once data is written to the file system the cipher-text will overwrite the originally generated "noise". This then highlighted a further point: the need to ensure that when a user deletes a file that the data is actually removed from the blocks. In all standard file systems the process of deleting a file does not actually remove the data, but in fact only the link to the data. In non-encrypted, as well as non-steganographic, file systems this can have devastating consequences as the data remains on the disk until another file is written to the same location (as the location is marked as free), causing the data to be recoverable for anywhere from days (if the user defragments their partition regularly) to potentially years if the data existed on a remote backup server. Within the realm of encrypted and steganographic file systems this threat is essentially eliminated as only cipher-text will be recoverable, however if an attack on the encryption algorithm that was used is discovered then the cipher-text can easily be decrypted. In an attempt to prevent this from becoming a problem, all data which is set from allocated to unallocated will be "securely" deleted, by overwriting the inode and data blocks with random data once again. The patterns for this deletion technique will be sampled from the proposed method in [22]:

[20] a magic number is a constant used to identify the file or data type found in the header area

Pass	Binary Representation
1	`01010101`
2	`10101010`
3	`10010010`
4	`01001001`
5	`00100100`
6	`11111111`
7	`00000000`
8	*Random*

Therefore the final specifications for my Steganographic File System are as follows: a module for the Linux Kernel, which before writing file or metadata to disk, it encrypts the date using the XTEA algorithm and a key derived from both the file name and a user supplied password (or pass-phrase). Finally the encrypted inode and file data is then written 8 times; to an initial pseudo-random location, followed by 7 further locations offset from the first. The concept of creating hidden levels is an extension of hidden files from UNIX-based systems ("dot" files), where any directory name which starts with a dot is considered the next steganographic level. This allows for multiple levels to be spawned from any starting level, adding to security and the notion of plausible deniability.

3.2 Potential Driver Types

Choosing to write a module for the Linux Kernel was a decision which followed much deliberating. The nature of the VFS allows file systems to be implemented in many different ways. Although most standard file systems are either compiled into the kernel, or loaded at run-time as modules, there are many highly specific file systems for which it would be unnecessary to have the code executed in kernel space[21]. Some such file systems use the File System in User Space (FUSE[22]) module which enables normal users to manipulate the file systems in the same ways as the super user can under normal circumstances. These file systems which rely on FUSE are usually virtual file systems: they do not store the data themselves, but act as a view or translation of an existing file system. Figure 6 shows how a simple system call — in this case the command `ls -l` — is passed though the VFS to the FUSE module, then using the file system module written for FUSE, the file system can be accessed and the desired information returned to the user. SSHFS[23] is a FUSE-based file system for Linux (and other FUSE-compatible systems, such as Mac OS X) and is capable of operating on files on a remote computer using just a secure shell login on the remote machine. The practical effect of this is that the end user can seamlessly interact with remote files being securely served over SSH just as if they were local files on their own system.

Another possibility for the driver was to use the loopback device. Use of the loop back device would allow the entire file to exist on another file system as a normal file. This is a common practice with encrypted file systems for Linux, where the data is encrypted as it is "looped" back, enabling common file systems such as Ext2 and Ext3 (which do not support encryption natively) to act as encrypted file systems. Due to loopback devices seeing standard files as devices, they allow file system images — such as CD ISO[24] images or floppy disc images — to be mounted

[21] System memory in Linux can be divided into two distinct regions: kernel space and user space. Kernel space is where the kernel (i.e., the core of the operating system) executes (i.e., runs) and provides its services, and User space is that set of memory locations in which user processes (i.e., everything other than the kernel) run.

[22] FUSE is available from SourceForge.net (http://sourceforge.net/projects/fuse)

[23] More information about SSHFS can be found at their website: http://fuse.sourceforge.net/sshfs.html

[24] The International Organization for Standardization (ISO) defines a file system for CD-ROM media as ISO 9660.

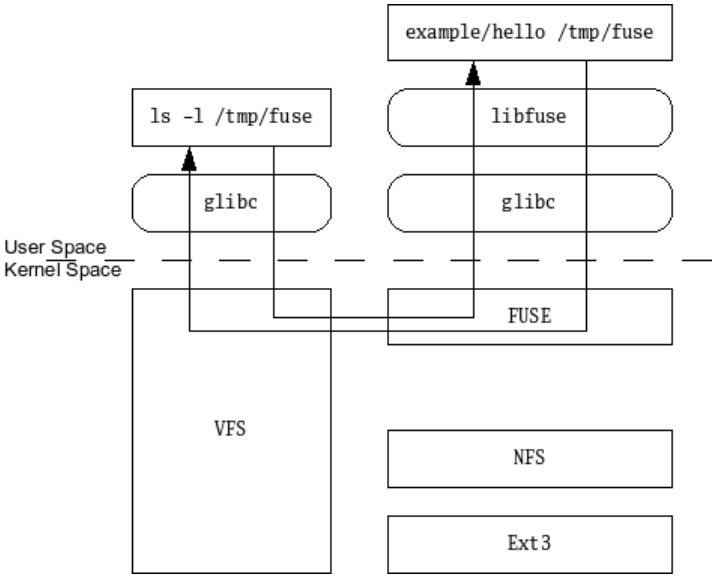

Figure 6: A file system call being passed through the VFS to the FUSE module

without the need for additional hardware. Mounting a file system image contained within a file using the loopback device involves two processes:

1. The file is associated with a loop device node (a special file)
2. The loop device is mounted on a directory as would be a normal device

These operations can be performed using two commands:

```
losetup /dev/loop0 example.img
mount /dev/loop0 /mnt/dir
```

The first command (`losetup`) associates the image `example.img` with the loop device node `loop0`, and the second is as any other mount command for mounting a file system hierarchy at the given directory. The `mount` command itself allows for these two commands to be combined as such:

```
mount -o loop example.img /mnt/dir
```

Using the loopback device has one major draw back: it can only be accessed by the super-user. Added to this is the requirement to already have a steganographic file system module for the kernel anyway, albeit without the encryption. Thus I could have chosen this route and forgone the encryption routine in my steganographic file system, this however would have forced access to the file system to the root user only, thus not fully imitating normal file system access properties.

Due to the nature of where I intended to start building the file system (§3.3) using either FUSE or the loopback device would have been, in some aspect, counter-productive. If I had chosen to build a FUSE module I would be writing code which would be almost identical to a kernel module — the names of the functions are all that would differ. If I had chosen to write

a file system which made use of the loopback device then I would have had a file system which could have acted as a standard file system given a block device instead of a loopback device. Moreover, writing the encryption function into my steganographic file system did not seem like that huge of a task to omit, just to be able to have the encryption handled by the loopback device.

3.3 Initial File System

The possibilities for using a file system which already existed was high: there are many file systems which are used for a variety of reasons, including those discussed in §2.2 and §2.3. All of the previous steganographic file system attempts have started with an Ext2 file system and added the necessary functions to allow files to be hidden and later retrieved. However, the initial proposal by Anderson, et al. made no assumption about the actual data structures of the file system, thus I was open to either follow the previous designs, or try something different.

I quickly concluded that any file system which was journaled could potentially cause security-related issues because of metadata being accidentally cached, and stored at a less secure level than the file it represents. I was not the first to conclude this either; McDonald and Kuhn[19] realised this when designing their file system, StegFS. This limited me to using either Ext2, UFS or VFAT (from the list of common file systems mentioned in §2.2) and I soon reduced this list to Ext2 — deeming that the available support for UFS in the Linux source tree to be inadequate (UFS is only support as a read-only file system), and VFAT does not support full UNIX permissions and is not suited to a multiuser environment.

The alternative option to modifying an existing file system was to create everything from scratch: my own inode and data structures, as well as implementing all the functions as I intended. This would allow my steganographic file system to do exactly what I wanted, and I would know exactly how it would do it — unfortunately, this would have been a monumental task on my part as at that time I had very little knowledge regarding Linux, and more specifically Virtual File System, function calls. As an aid to my learning about the VFS, and UNIX-like file systems, I obtained a copy of *UNIX Filesystems* by Steve Pate, and coincidentally that was where I found the Uxfs File System[25] as well as an invitation to use it as a starting block. This was exactly what I was looking for: a simple file system skeleton, on which I could add my own steganographic capabilities.

3.4 Encryption

Prior to being written to disk all of the data has to be encrypted to ensure that even through the scanning of raw sectors on the disk the information cannot be recovered and subsequently decrypted. Where encrypted file systems encrypt everything using the same key, steganographic file systems use a different key for each file, usually deduced from a combination of the file name and some user supplied password — this is essentially the same technique as calculating where on the disk the file will be. Choosing from any of the possible encryption algorithms which have no patent restrictions, gave a similar trade-off dilemma as with previous design choices among performance, security and simplicity. The eXtended Tiny Encryption Algorithm (XTEA)[23] is at one end of the spectrum, it is very simple and thus reasonably fast, whilst at the same time being moderately secure, whereas Rijndael[24], the Advanced Encryption Standard (AES)[26], is

[25]The original source code for Uxfs can be found at http://www.wiley.com/compbooks/pate/

[26]In 2003 the United States Government announced that AES may be used for classified information:

regarded as secure enough for the National Security Agency (NSA), yet is reasonably simple and therefore moderately fast when implemented in either software or hardware. In the end — for me — it came down to simplicity, thus XTEA is the algorithm used to encrypt and decrypt data in my file system. Figure 7 shows how the XTEA algorithm generates cipher-text from the original data, using the key.

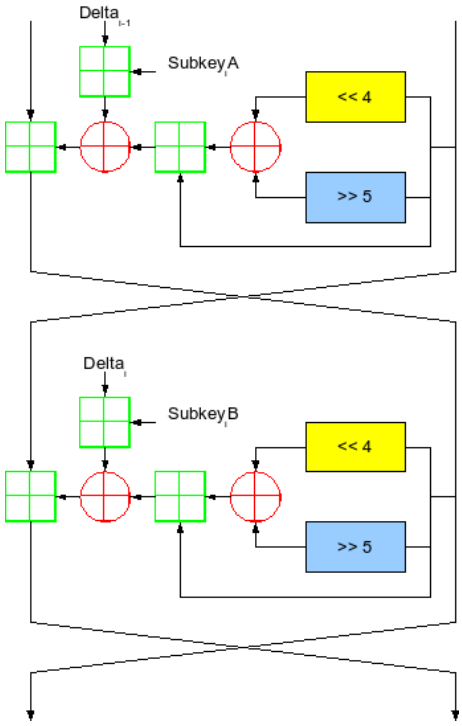

Figure 7: Two Feistel rounds (one cycle) of XTEA

3.5 A Discarded Design

Aside from the above mentioned tweaks to how my file system came together, there was my initial design. It was — sadly — too advanced for this project, yet noteworthy enough to deserve a mention on its own. In many ways it was similar to the FUSE module: it acted as a framework interface for file systems, almost like another VFS, yet it was steganographic. It was to allow generic support for any file system, much like the VFS, however, all of the data would be encrypted before being passed to the file system driver itself. This is similar to the loopback device with one exception: it would tell the file system driver where to write the data, thus providing the steganography. This idea was to allow the user to add steganographic capability to any file system (which could interface with the VFS) without the need for "converting" any existing partitions — similar to the reasons for using Ext2 by [19] and Magikfs.

The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256 bits) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 bit key lengths. The implementation of AES in products intended to protect national security systems and/or information must be reviewed and certified by NSA prior to their acquisition and use.[25]

In addition to allowing support for any file system compatible with the VFS, I had intended to use a plugin system for the encryption; enabling the user to choose when they mount the file system which algorithm they wish to have the data encrypted with. Theoretically the user could "stack" multiple steganographic file systems on top of a single original file system, each using a different encryption algorithm each time. Obviously this risks corruption in each of the steganographic file systems, but would add an unrivalled level of plausible deniability.

The only major problem with this proposal was telling the file system driver where to read or write the hidden files. To obtain maximum compatibility with standard file systems the steganographic VFS would not be allowed to modify the how the original drivers were implemented, otherwise all it would really have achieved was an over-bloated steganographic file system. Initially I was convinced that this could be overcome by "patching" the original file system source code, but, shortly after, realised that this would either result in multiple copies of almost identical file system drivers, or even worse: a single copy of a file system driver which only works when used with the steganographic extension and not under normal circumstances.

So whilst this design is ultimately a success theoretically, it has some severe disabilities which need to be addressed before it is able to become a success practically. Maybe, at some future time the Linux VFS will have a similar ability as standard, requiring all file systems for Linux to allow for the possibility of steganography.

4 Development

4.1 The Tools

Upon finalising my design I was able to start the coding portion of the project. I had my starting place, with the Uxfs skeleton, but I at the time was not sure how to begin; where the physical code would be written. I obviously needed a machine for which I had unlimited access as the super user to allow me to load and unload my file system module to and from the running kernel, but had realised that using the own machine hardware could cause delays when, undoubtedly, I would write erroneous code and be required to restart my machine. Fortunately, the quality of virtualisation software has increased rapidly in the last few years, and there are were many available virtualisation suits to choose from: VMware[27], Qemu[28], and Bochs[29], to name the three most popular. Each of these packages are able to emulate popular hardware components allowing for entire operating systems to be installed and executed as a guest on what is known at the host operating system. This makes testing and previewing operating systems available without the need to format hard disks and start from scratch — not even rebooting the host system is needed to set up a guest. This type, of configuration would allow me to code the file system in an emulated environment, with no risk of adversely affecting my own machine and potentially loosing unsaved work.

Eventually I chose to use Qemu for two reasons: firstly it is free, and secondly it is provided

[27] VMware is the commercial world leader in virtualisation software, with the largest support for all standard x86 based hardware components. More information about VMware can be found at http://www.vmware.com

[28] Qemu is the biggest *Free Software (Appendix A.5 gives the official definition)* alternative to VMware. Its hardware device support is somewhat more limited, however it does have the ability to emulate many different types of processor architecture from a single host chip — a feature VMware lacks. Qemu is available from http://fabrice.bellard.free.fr/qemu/

[29] Bochs is virtualisation software for more experienced users: it focuses more on tweaking hardware parameters and functionality and less on look-and-feel; it is available from http://bochs.sourceforge.net

by my GNU/Linux[30] distribution for easy installation. Next I needed a light-weight GNU/Linux distribution which would not be hindered by the tight "system requirements" within the virtual machine (VM), as well as with easy access to the source code for the kernel version it used. Finding a distribution which fitted the profile I required was not an easy task, as there any hundreds freely available for almost any purpose, yet I was able to reduce the potentials to just three: Knoppix[31], Slax[32] and Damn Small Linux[33]. All three are live distributions, thus they are designed to be booted from CD or USB stick, and because of this it makes them prime candidates for development within a VM. After researching all three live CD's: testing the ability to access the Linux source for the kernel provided, as well as the ability to run smoothly when performing demanding tasks with the smallest amount of available resources, I concluded that Damn Small Linux was the distribution to use.

4.2 The Process

So by now I had code for a file system on which I could build steganographic capabilities, as well as a safe environment in which to develop and test my modifications; I could begin.

Due to the simplicity of Uxfs it was not too difficult to understand what was going on in the various functions, and I was quick to establish where I needed to make modifications to randomise to which inode number the initial inode data would be written. Following this enhancement, I continued to trace the path of execution through the original code and was able to flag whether an entry was needed in the current directory or if the inode in question referred to a hidden level and needed to be omitted. Many of the functions that were modified during this process also needed their list of parameters and return values altered to allow my modifications to know whether or not the inode being manipulated was to be hidden, therefore I was not just changing how the functions were acting, but also how they were being called and what return values the calling function should expect to receive. It is worth noting that at this point there is still no concept of hidden levels — the directories are just inaccessible without know what their called. Encryption is also missing to enable me to view where the files are being placed on the device, using various file system forensic tools, discussed further in §5.

Once I was satisfied that inode structures were being written and re-read successfully I moved on to the actual file data blocks. As the process for writing the data to the file system for both inodes and data are similar, they are almost identical and as such altering the code and adding my randomiser to determine where to place the data was easily ported from the inode functions to the data block equivalents.

[30]The first Linux systems were completed in 1992 by combining system libraries from the GNU Project with the Linux kernel, this led to the coining of the term GNU/Linux. The GNU Project, started in 1983 by Richard Stallman, aimed to create an entirely free operating system. GNU is a recursive acronym that stands for *GNU's not UNIX*. More information about the Free Software Foundation and GNU can be found at http://www.fsf.org and http://www.gnu.org respectively.

[31]Knoppix is a Debian-based live CD distribution (with the option to become a fully-installed Debian distribution if one wishes), developed by Linux consultant Klaus Knopper. It is available from http://knopper.net/knoppix/

[32]Slax is a plugin orientated Slackware-based distribution, with a live CD size of approximately 192 MiB. Its sole author is Tomas Matejicek, and can be found at http://www.slax.org/

[33]DSLinux is the smallest of all distributions: John Andrews (the author) states that the overall size of DSL will never exceed 50 MiB:

> The whole idea behind DSL is trying to fit a complete, fully functional desktop into a small footprint. If we were to raise the size, we would become just another distro. And the fun of working more and more functionality out of 50mb would be gone. DSL will NEVER go over 50mb for the base iso.[26]

DSL home page: http://www.damnsmalllinux.org

As I was able to "gain" time with the previous task, the subsequent stall seemed less of a threat. So far I was able to write a file and its inode (be it hidden or otherwise) to a pseudo-random location. Reading back non-hidden inodes and files was not an issue either, unfortunately hidden inodes tended to be lost upon unmounting of the file system. This was obviously a rather large problem, because if a user cannot read back their hidden data then the entire purpose of the steganographic file system is flawed. The solution to this problem continued to elude me for much longer than I would have liked, and became much more of a issue before being fixed. As all of the entries with a directory are stored as data within the inode itself, trying to read from an inode which was regarded as unused by the file system resulted in a buffer overflow, as the process of reading the directory entries would continue passed the last entry, until either the end of the file system or a buffer overflow error occurred. Neither effects were desired.

After the effects of the buffer overflow were resolved, development halted to being the first phase of testing. Details of this series of events is covered in §5.2.

With the first testing phase completed, implementation of the XTEA encryption algorithm could begin. Unlike the previous code, where I enhanced the inode structures and functions, this time I started with the data blocks because I believed this approach to be easier. The file data could be encrypted before being written, all as a single block, where as the data within the inode structure could only be encrypted individually, as it was written. Encrypting the data blocks proved to be a trivial process; a pointer to the data could be passed to the encryption function, which would return a pointer to the encrypted data, which could subsequently be written to disk. Unfortunately writing encrypted inode data structures was more complicated, as the structure were a collection of pointers and therefore, as a whole, could not be encrypted. Upon successful resolution of this problem, the implementation of my steganographic file system would be complete.

5 Testing

5.1 Progressive

Much of the test occurred during the implementation — using `printk` statements at key points within functions to instruct the kernel to record in its log file what was happening. This technique was especially useful in the early stages of development as it enabled me to confirm that various actions were functioning correctly and that pseudo-random inode and block values were being calculated and passed between the necessary functions. This technique was also helpful when attempting to construct the level model, by informing me as to what type of file had just been created: normal file, normal directory, or hidden directory (next level). Most of the `printk` statements will have been removed (or at the least commented out) for the functioning file system as it is not necessary, nor secure, to log that a new hidden level has been created, however some information might need to be logged for general debugging purposes. This information includes whether the file system was successfully mounted, and if not, possible reasons for the failure. Likewise with regards to unmounting the file system.

Aside from writing to the kernel log during execution, the testing phase also included attempts to understand exactly how the data was being stored on the disk, and whether the secure file deletion routine as functioning as expected. Being able to view the contents of the file system from the point of view of an attacker enabled me to understand how they might go about searching for the data they want. As devices are essentially a special type of file, they are susceptible to standard file manipulation techniques. Provided an attacker has full access to the

disks themselves, they have a wealth of tools which will allow them to extract any data available. Even using the simple `dd` command can copy a file system on a device to a regular image file:

```
dd if=/dev/hdc of=cdimage.iso bs=2352
```

Following this, the use of standard UNIX commands such as `sed` and `grep` will allow for the searching of known strings or regular expressions[34]. Using these utilities an attacker is able to easily search file systems for known strings of text, but for a more thorough analysis, various forensic tools would be necessary.

5.2 Forensic Analysis

The use of simple command line tools, however, is unable to provide a comprehensive report of recent file system activity. There are many complete kits available for this very task: The Sleuth Kit (TSK) and Autopsy[35][27] are a collection of command line tools for UNIX-like operating systems with optional graphical interface. TSK and Autopsy have the ability to analyse all of the popular file systems which are mentioned throughout this report, with the ability to list files and directories, recover deleted files, make timelines of file activity, perform keyword search and use the hash database. Many of the tools included with TSK are Free Software alternatives to The Coroner's Kit[36].

The Sleuth Kit contains over 20 commands which are organised by group: disk, volume, file system and search command tools. The group for which a particular command belongs to, can be identified by the name of the command. The first letter indicates which group the command is within, and the remainder hints at what the command does. On a sample Ext3 file system, the command `fsstat -f linux-ext3 ext3.dd` results with:

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name:
Volume ID: 6ce83217d46137a824e4fdcb6ab62df

Last Written at: Fri Apr 27 02:44:42 2007
Last Checked at: Fri Apr 27 02:44:42 2007

Last Mounted at: emptyUnmounted properly
Last mounted on:

Source OS: Linux
Dynamic Structure
Compat Features: Resize Inode, Dir Index
InCompat Features: Filetype,
Read Only Compat Features: Sparse Super,

METADATA INFORMATION
```

[34] A regular expression is a string that is used to describe or match a set of strings, according to certain syntax rules.

[35] The Sleuth Kit/Autopsy are available from http://www.sleuthkit.org

[36] TCK is available under the IBM Public License, from http://www.porcupine.org

```
--------------------------------------------
Inode Range: 1 - 128
Root Directory: 2
Free Inodes: 117

CONTENT INFORMATION
--------------------------------------------
Block Range: 0 - 1023
Block Size: 1024
Reserved Blocks Before Block Groups: 1
Free Blocks: 986

BLOCK GROUP INFORMATION
--------------------------------------------
Number of Block Groups: 1
Inodes per group: 128
Blocks per group: 8192

Group: 0:
  Inode Range: 1 - 128
  Block Range: 1 - 1023
  Layout:
    Super Block: 1 - 1
    Group Descriptor Table: 2 - 2
    Data bitmap: 6 - 6
    Inode bitmap: 7 - 7
    Inode Table: 8 - 23
    Data Blocks: 24 - 1023
  Free Inodes: 117 (91\%)
  Free Blocks: 986 (96\%)
  Total Directories: 2
```

This information shown is extracted from the superblock and can be used to deduce the total number of inodes and blocks, as well as the size of the blocks. Other commands are capable of listing all unused block values, or echoing to the screen the raw contents of the file system.

Where TSK provides the command line tools for the forensic analysis of file systems, Autopsy provides a graphical web front end to assist with organising the process of analysing the file system. In addition, it provides case management so that multiple web clients can connect to the Autopsy server and work in a similar way to many source code tree servers (such as CVS and SVN)[37] by keeping track of changes and logging all activity of analysis. Notes can also be kept about what data is found.

Autopsy also allows for creating of lists of search results with thumbnails of the file they expose; e.g. if a directory of deleted images is recovered, then thumbnails of the images can be displayed within the search results. Further structuring of results is based upon what type of files where searched for initially. Using these forensic tools I was able to verify that data blocks and inodes where correctly written to, and that any multiple copies were in fact correctly

[37]Both, Concurrent Versions System (CVS) and Subversion (SVN), are source code version control systems allowing for multiple users to edit files within the same source tree without conflicting with each other. SVN (being the replacement to CVS) also allows users to simultaneously edit the same file.

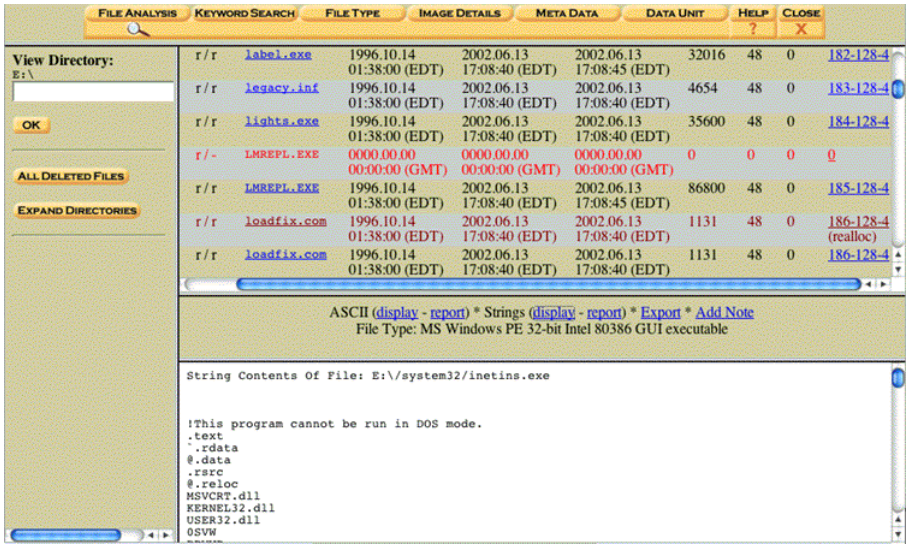

Figure 8: Autopsy file listing of deleted files

replicated. These tools also gave me to opportunity to experiment with the file systems ability to recognise when the inode or block is corrupt. This was an additional element of testing which I had originally not considered, but after presenting itself, proved to have been extremely useful.

Being satisfied with how the files were being positioned on the file system I was able to return to the code, ready to implement the final feature: encryption. This, along with the pseudo-random data in unused sectors of the disk resulted in a file system which looked entirely pseudo-random, and as I was not about to try to show one way or another how secure the XTEA algorithm is, the file system should now provide steganographic capabilities with the much sought-after plausible deniability.

6 Conclusion

6.1 Project Evaluation

From my personal viewpoint, this project has been a success: I have learnt a great deal about how file systems work and are structured, as well as how the operating system interfaces with the hardware through the file system module. Whilst not necessarily as integrated with traditional file systems, as is the case with the previous attempts at steganographic file systems, this project illustrates that the motivation behind wanting — maybe even needed — a file system which offers plausible deniability is increasing.

The techniques I employed for creating the notion of levels within the steganographic file system are unique, and because they require no additional tools, provide the simplest interface for day-to-day actions associated with files. Providing a `mkfs` tools does nothing to invalidate this claim, as every file system requires a tool for laying the foundations for use. I made the choice to keep everything simple, not only to ease myself into writing a Linux module for the first time, but to help others who have the desire to further this steganographic file system; it provides a starting point with very little “clutter” and optimisation code. During my search

for a base file system I was overwhelmed by the complexity of Ext2, even browsing the source for VFAT in the kernel tree was an enormous task because of all of the code which deals with simulating the POSIX user permissions whilst the file system is mounted.

Much of the time, the project was progressing smoothly, at a steady pace, and aside from the occasional problems — which were mentioned in §4.2 — everything was okay. Despite the continued progress, much of the code accompanying this report is incomplete due to the need to initially learn how Uxfs worked. I am sure, that without the Uxfs file system available as a building block, that I would still be trying to get my head round the inner workings of Ext2.

6.2 Future Enhancements

Aside from the major design I proposed in §3.5, any future attempts at a steganographic file system which I undertake, would include a range of techniques and and modifications which were just not feasible with a one-man team. Although all steganographic file systems which exist are based on the second proposal by Anderson, Needham and Shamir, it would have been novel to use the first, which would further secure the data it concealed, as the entire method does not rely on encryption, but linear algebra. This would also reduce the redundancy that is present in all other steganographic file systems (for the reasons in [16]) due to duplicating files in an attempt to ensure their integrity. This method also allows users access to a steganographic file system even if the use of encryption is prohibited by law; by obscuring the data the linear algebra, theoretically, should be enough to proved suitable security.

Using the first proposal would also be reason enough to design the entire file structure from the beginning — where it is not based on any existing file systems, such as Ext2. This would allow for various optimisations with regards to how the data is actually stored in inodes and blocks. In this instance, I would likely place any possible inode data at the head of the block, before any file data, reducing the number of structures that are needed to be read from the file system when accessing a file. With the knowledge of both linear algrbra, the inner working of the Linux Kernel, and an understanding of file systems, this plan, in conjunction with my discarded design, would provide a fully comprehensive steganographic file system which would be the envy of government agencies throughout the world.

6.3 Summary

Throughout history people have needed to communicate with each other, and now an ever increasing number of people are using electronic means to send their messages. These same people are also becoming more aware of their right to privacy, and how many governments are introducing new laws to combat terrorism without thinking about the right to privacy of the average Joe. A thousand years ago, if a private conversation was needed it was sufficient to walk to the middle of a clearing or locked room, and with a quick check, ensure that no one was eavesdropping. Even twenty years ago messages could be sent with the strong belief that they were not going to be read by anyone along their path. It is far from economical to open, read and reseal all mail which passes through a sorting office (except in a dictatorship where citizens are employed to spy on fellow citizens), yet intercepting thousands of emails and searching for key words and phrases can be performed in minutes, if not seconds.

In 1993 Philip Zimmermann took the first step to providing people with a secure way to communicate using email with Pretty Good Privacy[28]. Friends and colleagues could now send and receive emails safe in the knowledge that their messages could not be read by anyone else

and guarantee emails were from who they claimed. Now, just encrypting emails is not sufficient, and with more computer viruses in circulation and the advent of spyware being used to steal personal information and documents from home computers, it has become necessary to ensure that a users personal files are secure from attack. Encrypting individual files solved this problem but in doing so introduced another: if it is encrypted it must be worth encrypting. The presence of cipher-text, to some, may seem like fun or a challenge, to others it could imply guilt, therefore to overcome this, hiding the data, not being able to prove that there is any data in the first place, is the solution.

Steganography, the art and science of hiding messages, is this solution.

As far as encryption goes, the code makers have won by devising a method of encryption based on quantum physics and the uncertainty theory[29], which essentially provides a one-time pad[38] key. As such, provide absolute plausible deniability (even against quantum computers) because for any one-time pad resulting cipher text, an infinite number of keys can be used to derive an infinite number of plain text possibilities. Unfortunately, quantum cryptography uses the polarisation of light to determine the value of each bit, and until disks are able to store bit values in a similar fashion this will remain an infeasible method of data storage. Therefore steganographic techniques will have to suffice, and whether files are hidden in other files or the file system driver itself is modified, digital steganography (for now) guarantees that private data stays secure.

[38]The one-time pad is an encryption algorithm where the plain text is combined with a random key or "pad" that is as long as the plain text and used only once. A modular addition (for example XOR) is used to combine the plain text with the pad. If the key is truly random, never reused, and kept secret, the one-time pad can be proven to be unbreakable.

A Appendix

A.1 File System Switch Functions[21]

This table shows the main structures used in the FSS architecture. It is worth noting that the structures are independent of the type of file system.

FSS Function	Description
`fs_init`	Each filesystem can specify a function that is called during kernel initialization allowing the filesystem to perform any initialization tasks prior to the first `mount` call
`fs_iread`	Read the inode (during pathname resolution)
`fs_iput`	Release the inode
`fs_iupdat`	Update the inode timestamps
`fs_readi`	Called to read data from a file
`fs_writei`	Called to write data to a file
`fs_itrunc`	Truncate a file
`fs_statf`	Return file information required by `stat ()`
`fs_namei`	Called during pathname traversal
`fs_mount`	Called to mount a filesystem
`fs_umount`	Called to unmount a filesystem
`fs_getinode`	Allocate a file for a pipe
`fs_openi`	Call the device open routine
`fs_closei`	Call the device close routine
`fs_update`	Sync the superblock to disk
`fs_statfs`	Used by `statfs ()` and `ustat ()`
`fs_access`	Check access permissions
`fs_getdents`	Read directory entries
`fs_allocmap`	Build a block list map for demand paging
`fs_freemap`	Frees the demand paging block list map
`fs_readmap`	Read a page using the block list map
`fs_setattr`	Set file attributes
`fs_notify`	Notify the filesystem when file attributes change
`fs_fcntl`	Handle the `fcntl ()` system call
`fs_fsinfo`	Return filesystem-specific information
`fs_ioctl`	Called in response to a `ioctl ()` system call

A.2 File System Comparison

The following table compare general and technical information for a number of standard file systems mentioned in this document.

Year	File System	File Limit	Partition Size	POSIX	Journaled	Encrypted
1977	FAT12	4,077	32 MiB			
1985	HFS	65,535	2 TiB			
1987	FAT16	65,517	4 GiB			
1990	JFS	*Unknown*	4 PiB	Yes	Yes	
1993	Ext2	2^{64}	32 TiB	Yes		
1993	NTFS	2^{32}	16 EiB		Yes	Yes
1994	UFS	*Unknown*	256 TiB	Yes		
1994	XFS	*Unknown*	8 EiB	Yes	Yes	
1996	FAT32	2^{28}	2 TiB			
1998	HFS+	*Unlimited*	16 EiB	Yes	Yes	
1999	Ext3	*Variable*	32 TiB	Yes	Yes	
2001	ReiserFS	2^{32}	16 TiB	Yes	Yes	
2004	Reiser4	*Unlimited*	*Untested*	Yes	Yes	Yes
2006	Ext4	*Unlimited*	1024 PiB	Yes	Yes	

A.3 IEC Standard Prefixes[30]

The new prefixes and symbols for binary multiples standardized in IEC 60027-2 are not part of the SI metric system of units. But as the table below shows, they are related to the SI prefixes and symbols for positive powers of ten in a simple way so that they are easy to remember and use.

Name	Symbol	Base 2	Base 10	SI Derivation	SI Factor
kibi	Ki	$(2^{10})^1$	1,024	Kilo	$(10^3)^1$
mebi	Mi	$(2^{10})^2$	1,048,576	Mega	$(10^3)^2$
gibi	Gi	$(2^{10})^3$	1,073,741,824	Giga	$(10^3)^3$
tebi	Ti	$(2^{10})^4$	1,099,511,627,776	Tera	$(10^3)^4$
pebi	Pi	$(2^{10})^4$	1,125,899,906,842,624	Peta	$(10^3)^5$
exbi	Ei	$(2^{10})^5$	1,152,921,504,606,846,976	Exa	$(10^3)^6$
zebi	Zi	$(2^{10})^6$	1,180,591,620,717,411,303,424	Zetta	$(10^3)^7$
yobi	Yi	$(2^{10})^8$	1,208,925,819,614,629,174,706,176	Yotta	$(10^3)^8$

A.4 Secure Data Removal Patterns[22]

With a set of 22 overwrite patterns, and 8 random passes, everything should erase everything, regardless of raw encoding scheme. The the sequence of 35 consecutive writes shown below can be performed in any order to to make it more difficult to guess which of the known data passes were made at which point.

Pass No.	Data Written	Encoding Scheme Targeted		
		(1,7) RLL	(2,7) RLL	MFM
1	*Random*			
2	*Random*			
3	*Random*			
4	*Random*			
5	01010101 01010101 01010101	Yes		Yes
6	10101010 10101010 10101010	Yes		Yes
7	10010010 01001001 00100100		Yes	Yes
8	01001001 00100100 10010010		Yes	Yes
9	00100100 10010010 01001001		Yes	Yes
10	00000000 00000000 00000000	Yes	Yes	
11	00010001 00010001 00010001	Yes		
12	00100010 00100010 00100010	Yes		
13	00110011 00110011 00110011	Yes	Yes	
14	01000100 01000100 01000100	Yes		
15	01010101 01010101 01010101	Yes		Yes
16	01100110 01100110 01100110	Yes	Yes	
17	01110111 01110111 01110111	Yes		
18	10001000 10001000 10001000	Yes		
19	10011001 10011001 10011001	Yes	Yes	
20	10101010 10101010 10101010	Yes		Yes
21	10111011 10111011 10111011	Yes		
22	11001100 11001100 11001100	Yes	Yes	
23	11011101 11011101 11011101	Yes		
24	11101110 11101110 11101110	Yes		
25	11111111 11111111 11111111	Yes	Yes	
26	10010010 01001001 00100100		Yes	Yes
27	01001001 00100100 10010010		Yes	Yes
28	00100100 10010010 01001001		Yes	Yes
29	01101101 10110110 11011011		Yes	
30	10110110 11011011 01101101		Yes	
31	11011011 01101101 10110110		Yes	
32	*Random*			
33	*Random*			
34	*Random*			
35	*Random*			

A.5 Free Software Definition[31]

Free software is a matter of liberty, not price. To understand the concept, you should think of free as in free speech, not as in free beer.

Free software is a matter of the users' freedom to run, copy, distribute, study, change and improve the software. More precisely, it refers to four kinds of freedom, for the users of the software:

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

A program is free software if users have all of these freedoms. Thus, you should be free to redistribute copies, either with or without modifications, either gratis or charging a fee for distribution, to anyone anywhere. Being free to do these things means (among other things) that you do not have to ask or pay for permission.

You should also have the freedom to make modifications and use them privately in your own work or play, without even mentioning that they exist. If you do publish your changes, you should not be required to notify anyone in particular, or in any particular way.

The freedom to run the program means the freedom for any kind of person or organization to use it on any kind of computer system, for any kind of overall job and purpose, without being required to communicate about it with the developer or any other specific entity. In this freedom, it is the user's purpose that matters, not the developer's purpose; you as a user are free to run a program for your purposes, and if you distribute it to someone else, she is then free to run it for her purposes, but you are not entitled to impose your purposes on her.

The freedom to redistribute copies must include binary or executable forms of the program, as well as source code, for both modified and unmodified versions. (Distributing programs in runnable form is necessary for conveniently installable free operating systems.) It is ok if there is no way to produce a binary or executable form for a certain program (since some languages don't support that feature), but you must have the freedom to redistribute such forms should you find or develop a way to make them.

In order for the freedoms to make changes, and to publish improved versions, to be meaningful, you must have access to the source code of the program. Therefore, accessibility of source code is a necessary condition for free software.

One important way to modify a program is by merging in available free subroutines and modules. If the program's license says that you cannot merge in an existing module, such as if it requires you to be the copyright holder of any code you add, then the license is too restrictive to qualify as free.

In order for these freedoms to be real, they must be irrevocable as long as you do nothing wrong; if the developer of the software has the power to revoke the license, without your doing anything to give cause, the software is not free.

A.6 Additional Autopsy Screenshots[32]

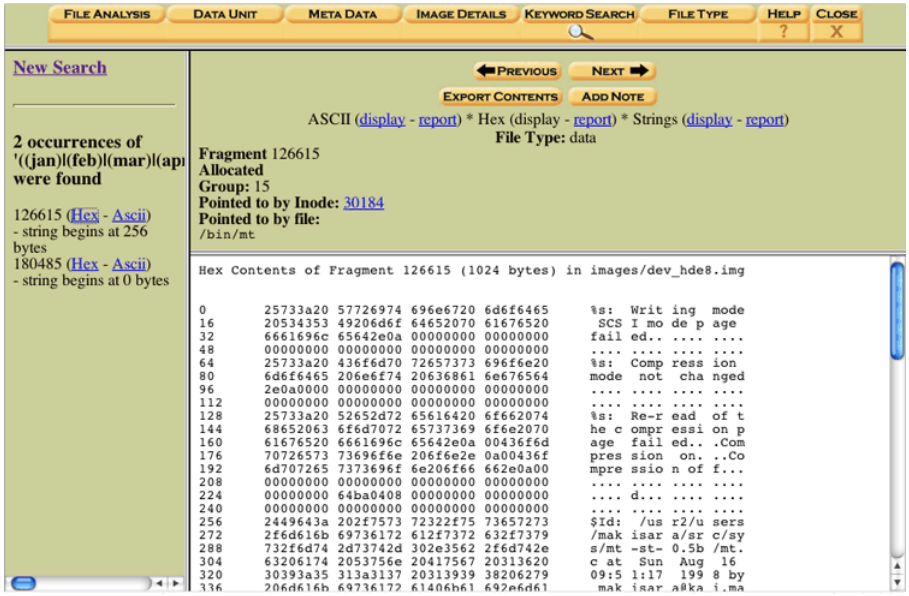

Figure 9: Autopsy keyword search results

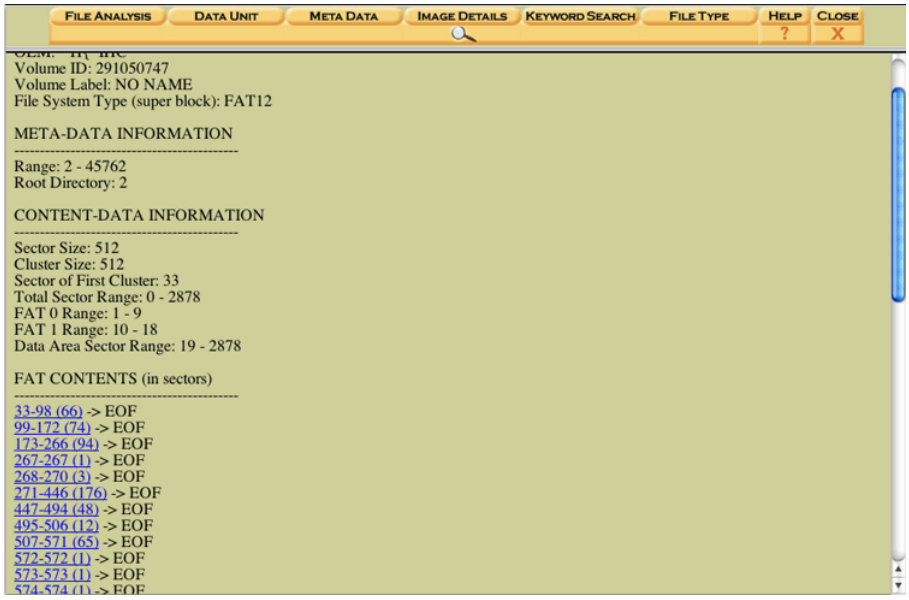

Figure 10: Autopsy volume information

References

[1] C. Cachin: *An Information-Theoretic Model for Steganography*, D. Aucsmith (ed.) *Proceedings of* 2^{nd} *Workshop of Information Hiding*, Springer-Verlag (1998)

[2] S. Singh: *The Code Book: The Secret History of Codes and Code-Breaking*, p.5, Fourth Estate (1999)

[3] R.P. Carlisle: *The Complete Idiot's Guide to Spies and Espionage*, p.213, Alpha Books (2003)

[4] R. Hyde, M. Koehler and R. Rodriguez: *The Generic File System*, *Proceedings of the USENIX Summer Technical Conference*, USENIX Association, pp.260–269 (1986)

[5] H.X. Mel and Doris Decker: *Cryptography Decrypted*, Addison-Wesley (2001)

[6] D. Kahn: *The Codebreakers*, Macmillan New York (1967)

[7] W. White: *The Microdot: History and Application*, Phillips Publications (1992)

[8] S.R. Kleiman: *Vnodes: An Architecture for Multiple File System Types in Sun UNIX*, preprint (1985)

[9] S. Demblon and S. Spitzner: *Linux Internal*, http://learnlinux.tsf.org.za/courses/build/internals/internals-all.html (2004), (accessed 07.04.2007)

[10] R. Duncan: *Design goals and implementation of the new High Performance File System*, Microsoft Systems Journal 4 (1989)

[11] H. Custer: *Inside the Windows NT File System*, Microsoft Press (1994)

[12] M.K. McKusick, W. Joy, S. Leffler and R. Fabry: *A Fast File System for UNIX*, *Communications of the ACM*, pp.181–197 (1984)

[13] R. Card, T. Ts'o and S. Tweedie: *Design and Implementation of the Second Extended Filesystem* (1993), F. Brokken (ed.) *Proceedings of the First Dutch International Symposium on Linux*, State University of Groningen (1995)

[14] S. Tweedie: *Journaling the Linux ext2fs Filesystem*, Linux Expo (1998)

[15] Silicon Graphhics Inc.: *Getting Started With XFS Filesystems*, Silicon Graphics Inc. (1994)

[16] R. Anderson, R. Needham and A. Shamir: *The Steganographic File System*, D. Aucsmith (ed.) *Information Hiding, Second International Workshop*, pp.73–82, Springer-Verlag (1998)

[17] R. Anderson, R. Gibbens, C. Jagger, F. Kelly and M. Roe: *Measuring the Diversity of Random Number Generators*, preprint (1992)

[18] P.Å. Larson and A. Kajla: *File Organisation: Implementation of a Method Guaranteeing Retrieval in One Access*, *Communications of the ACM* pp.670–677 (1984)

[19] A. McDonald and M. Kuhn: *StegFS: A Steganographic File System for Linux* (1999), A. Pfitzmann (ed.) *Information Hiding*, pp.463–477, Springer-Verlag (2000)

[20] H. Pang, K.L. Tan, X. Zhou: *StegFS: A Steganographic File System*, *Proceedings of the* 19^{th} *International Conference on Data Engineering*, pp.657–668 (2003)

[21] S.D. Pate: *UNIX Filesystems: Evolution, Design, and Implementation*, Wiley Publishing, p.124 (2003)

[22] P. Gutmann: *Secure Deletion of Data from Magnetic and Solid-State Memory*, Sixth USENIX Security Symposium Proceedings (1996)

[23] R. Needham and D. Wheeler: *Tea extensions*, preprint (1996)

[24] S. Borg, J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer-Verlag, (2002)

[25] N. Ferguson, R. Schroeppel and D. Whiting: *A simple algebraic representation of Rijndael*, *Proceedings of Selected Areas in Cryptography*, pp.103–111, Springer-Verlag (2001)

[26] Damn Samll Linux: *Frequently Asked Questions*, http://www.damnsmalllinux.org/wiki/index.php/FAQ (2007), (accessed 26.04.2007)

[27] B. Carrier: *File System Forensic Analysis*, Addison-Wesley (2005)

[28] P. Zimmermann: *The Official PGP Users Guide*, MIT Press (1996)

[29] C. Bennett, G. Brassard, F. Bessette, L. Salvail, J. Smolin, *Experimental Quantum Cryptography*, (1991), *Scientific America*, pp. 26-33 (1992)

[30] International Electrotechnical Commission: *Prefixes for Binary Multiples*, http://www.iec.ch/zone/si/si_bytes.htm (2007), (accessed 26.04.2007)

[31] The Free Software Foundation: *The Free Software Definition*, http://www.gnu.org/philosophy/free-sw.html (2007), (accessed 26.04.2007)

[32] B. Carrier: *Autopsy Forensic Browser*, http://www.sleuthkit.org/autopsy/desc.php (2003), (accessed 26.04.2007)