Differential Attacks on Deniably Encrypted File Systems

Matthew Joyce

Project Supervisor:
Prof. Michael Goldsmith

Undergraduate 4th Year Project

Abstract

Deniable encryption is a form of encryption which allows the user either to deny that they created some data, or that the data exists at all. It is used by file systems to hide certain files, whose existence can then be plausibly denied. Several implementations of a deniable file system exist, but they can be broken using a differential attack which compares two or more snapshots of the same file system. The differences between the snapshots reveal writes to supposedly unallocated areas, which therefore actually contain hidden data. This report analyses three existing file system implementations which support deniability, and their susceptibility to differential attacks. The report then uses the lessons learnt to design a new file system that is provably resistant to differential attacks. This file system is then implemented and tested, showing that the new file system is both secure and usable enough for general use.

Contents

1 Introduction

Encryption, the process of reversibly scrambling messages to prevent attackers from reading the contents, has greatly revolutionised the information industry, enabling applications such as online commerce and secure file transfer. However, encryption has one major problem: the contents of an encrypted message are hidden, but the existence of the message is not. When an attacker can coerce a victim to reveal the message or the encryption key, encryption can be broken despite the strength of the encryption algorithm. This coercion can be applied using a legal threat such as key disclosure laws[*], a psychological threat such as blackmail, or physical threat such as violence[†]. Examples of such a situation include activists vs an authoritarian government, or a hostage vs their abductors.

One way to hide the existence of a message is steganography. Steganography attempts to hide the message inside a cover object, usually an image. For small messages, it can be quite successful. However, larger messages distort the cover object too much, and can be detected using statistical tests [12]. This is because no reasonable cover material is truly random, and small correlations in the cover material can be used to detect modification. Steganographic methods also lack any way of proving that stored information cannot be discovered, which makes it hard to trust them with sensitive data. This is especially true when the original, unmodified cover material can be recovered, since comparing any two versions of the cover material will highlight the differences caused by the steganographic algorithm.

Another method is deniable encryption. Deniable encryption is an application of normal encryption which creates a message that can be decrypted to an innocuous cover message, or a separate hidden message which does not have to be present. A strong deniable encryption protocol has a proof that the existence of the hidden message cannot be detected, even when the attacker can read the cover message by extracting the key from the author. Since a deniable message has a reason for its existence (the cover message), it can reasonably be claimed that there is no hidden message, a property known as plausible deniability. Deniable encryption is used as part of many applications, such as messaging using the OTR protocol [6] used by Signal [17] or the focus of this report which is secure hidden data storage in a file system.

There are multiple implementations of deniably encrypted file systems, including VeraCrypt [19] and stegfs [1]. However, these implementations suffer from a fatal flaw. As demonstrated by Hargreaves and Chivers in [11], a differential attack can defeat simple deniability implementations, such as those found in VeraCrypt and stegfs. A differential attack is an attack in which the attacker compares two or more snapshots of the same file system and analyses the difference between them. These changes can betray the presence of hidden data, by revealing writes to areas which supposedly contain no data. In some situations, such as secure temporary storage, the file system is short-lived and no snapshots are expected to exist, and so such an attack may not be a concern. However, most forms of long term storage have the potential to be copied, potentially without the users knowledge. Copies can be created by automatic backups [11], sharing copies of the file system with other users, or even retrieved from copies of data made by border control agents [13]. There is therefore the need for a new deniable file system, one which is resistant to differential attacks.

In this report I analyse three existing file systems that support plausible deniability, and show how differential attacks could be used against them. This report goes on to design a new file system from the ground up, with an emphasis on enforcing good usage habits and differential-attack

[*]In the UK this would be the Regulation of Investigatory Powers Act 2000.

[†]Sometimes referred to as rubber-hose cryptanalysis.

resistance, by applying the lessons learnt from the vulnerable file systems. This new file system is then implemented and tested in the same way as the existing file systems, to demonstrate its security properties. The file system is also tested for efficiency and usability, showing that it can therefore support normal workloads and can be used in place of existing solutions.

2 Analysis of Existing Implementations

In this section, three existing file systems which attempt to provide plausible deniability are analysed. The first, VeraCrypt, is a popular encrypted file system that includes a simple implementation of deniable encryption. The second, stegfs, is a more complex implementation that is based on steganography. And the final and most advanced implementation, Rubberhose, uses multiple layers of randomness to hide the data it contains. VeraCrypt and Rubberhose perform transparent disk encryption, exposing a set of virtual disk partitions inside of which any file system can be stored. In contrast, stegfs implements its own file system, including file structures and directory trees.

2.1 VeraCrypt

VeraCrypt (a fork of TrueCrypt) is free open source disk encryption software that supports multiple operating systems [18]. It can be used to encrypt disk partitions, or used to create a virtual partition stored inside a file. VeraCrypt does not implement a file system, but rather acts as a proxy to the backing partition, allowing normal file systems to be mounted inside its volumes.

VeraCrypt can provide plausible deniability though the use of a hidden volume. A VeraCrypt partition starts with a pair of headers. The first header is used by the main volume, while the second header is used by the hidden volume. Since both headers are encrypted, it is impossible to distinguish whether a hidden volume is in use from a single snapshot. The main volume is set up to use the entire partition, while the hidden volume is offset from the start of the partition. This means that the hidden volume's data is stored in the free space of the main volume. Figure 1 shows these differences as a diagram.

Given a single snapshot of a VeraCrypt partition and the key to the main volume, it is not in general possible to prove that a hidden volume exists: The main volume is configured to use the whole partition, and the header for the hidden volume is indistinguishable from randomness. However, this is not always enough to avoid detection. Since the main volume could accidentally overwrite data in the hidden volume, VeraCrypt implements a protected mode that switches the entire partition to read-only whenever a write to the main volume would modify the hidden volume [20]. However, this incentivises users to fill the main volume with data initially, and then only minimally change it afterwards to avoid errors. A lack of changes in the main volume makes explaining its presence harder, especially when there is evidence of recent use.

When an additional snapshot is available, hidden volumes are quite easy to find. Previously demonstrated by Hargreaves and Chivers in [11], the difference between the two snapshots can be computed, as shown in Figure 2. If a change is made to an area that is not marked as allocated by the main volume, the change was almost certainly made by a hidden volume. In addition, when a write occurs VeraCrypt simply encrypts the data and writes it to the specified location, so that there is a direct mapping between locations in the partition and locations in the virtual partition VeraCrypt presents to the file system. This means that the type of file system used for the hidden volume can be deduced, by looking for characteristic modification patterns. The hidden volume shown in Figure 3 contains a FAT file system, which can be deduced by looking at the start of the changes to the hidden volume. Clearly visible are the changes to the file allocation table (FAT*), as well as some file modifications.

*Not to be confused with the file system type named FAT, which uses a file allocation table as part of its implementation.

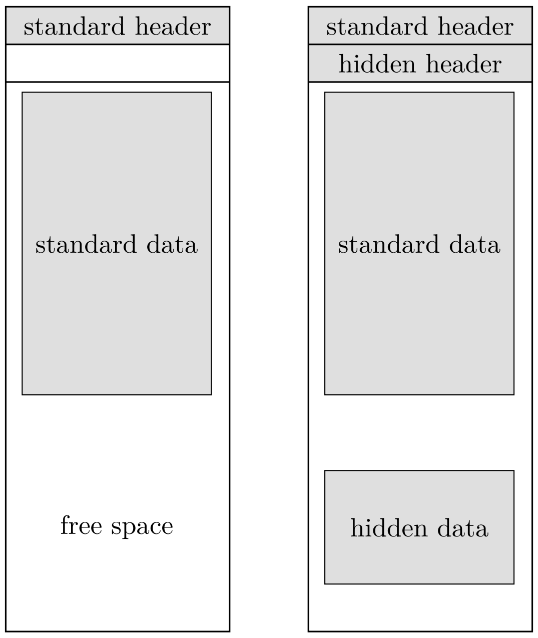

Figure 1: Header and data locations in a VeraCrypt container without a hidden volume (left) and with a hidden volume (right) [19].

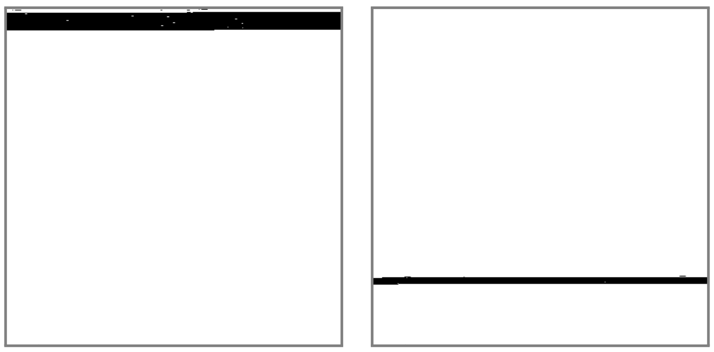

Figure 2: Difference between a Veracrypt container before and after writing to a normal standard partition (left) and the inner hidden partition (right). Each square is 512 bytes, and is coloured based on how much it has changed: black is fully changed while white is fully unchanged.

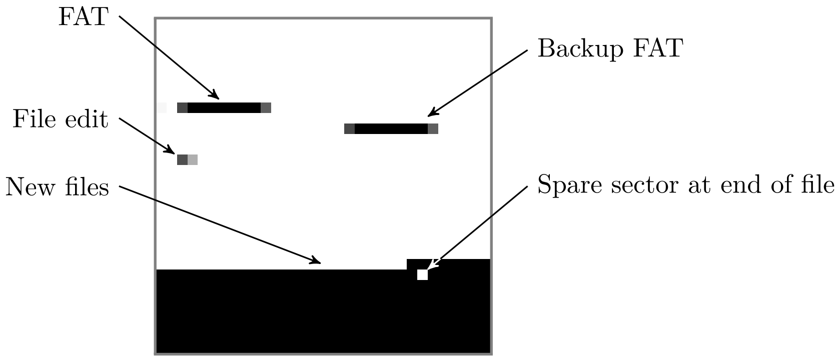

Figure 3: An enlarged view of the start of the difference caused when writing to the hidden inner partition, with interesting features labelled. As in Figure 2, each square is 512 bytes, and is coloured based on how much it has changed: black is fully changed while white is fully unchanged.

VeraCrypt does have one advantage over the other implementations, and that is its popularity. VeraCrypt is widely used for normal disk encryption, and so an overwhelming majority of VeraCrypt containers do not contain any hidden data. Therefore, an attacker who does not suspect the presence of hidden data is unlikely to become suspicious when VeraCrypt software is found on the victim's machine. This is an advantage over the other implementations analysed in this section, since they are only utilised by users interested in deniable encryption, making it much more likely that hidden data exists.

2.2 Stegfs

Stegfs is a steganographic file system for Linux that is implemented using FUSE (File System in Userspace). There are several loosely related versions of stegfs, and the version* covered here is the version by GitHub user 'albinoloverats' [1]. Stegfs provides plausible deniability in the following way: Each file is encrypted by a key derived from the file's name and then the file header is written to a pseudorandom location that is also based on the file's name. The rest of the file is written to random locations in the partition. This ensures that it is hard to find a file or even prove that it exists without knowing the file's full path, which can optionally contain a key to increase its randomness. Directories are not stored at all, and must be recreated each time the partition is mounted. A diagram of a file's structure can be found in Figure 4.

Due to the randomness, writing a new file to a stegfs file system can overwrite an existing file. To combat this the file system writes 8 copies of the same file, each using a different location generation seed. However, the Birthday Paradox means that even on a large partition, collisions will occur quickly. This behaviour is problematic, as a file system that randomly destroys its own files cannot be used for reliably storing sensitive or important information. The author of stegfs suggests using the file system with temporary storage devices such as USB sticks and SD cards, to hide documents that are being transferred from one system to another [1]. This avoids the problems with stegfs that makes long-term storage risky.

When only a single snapshot of a stegfs file system is available, the file system's contents are generally secure. Using directories can be an issue, since the file header includes an unencrypted hash of the file's directory path, which could be used to prove that a directory exists if a significant

*Tests done on stegfs-2015.08.1-27-g8ddd255, which was the latest public version at the time of writing.

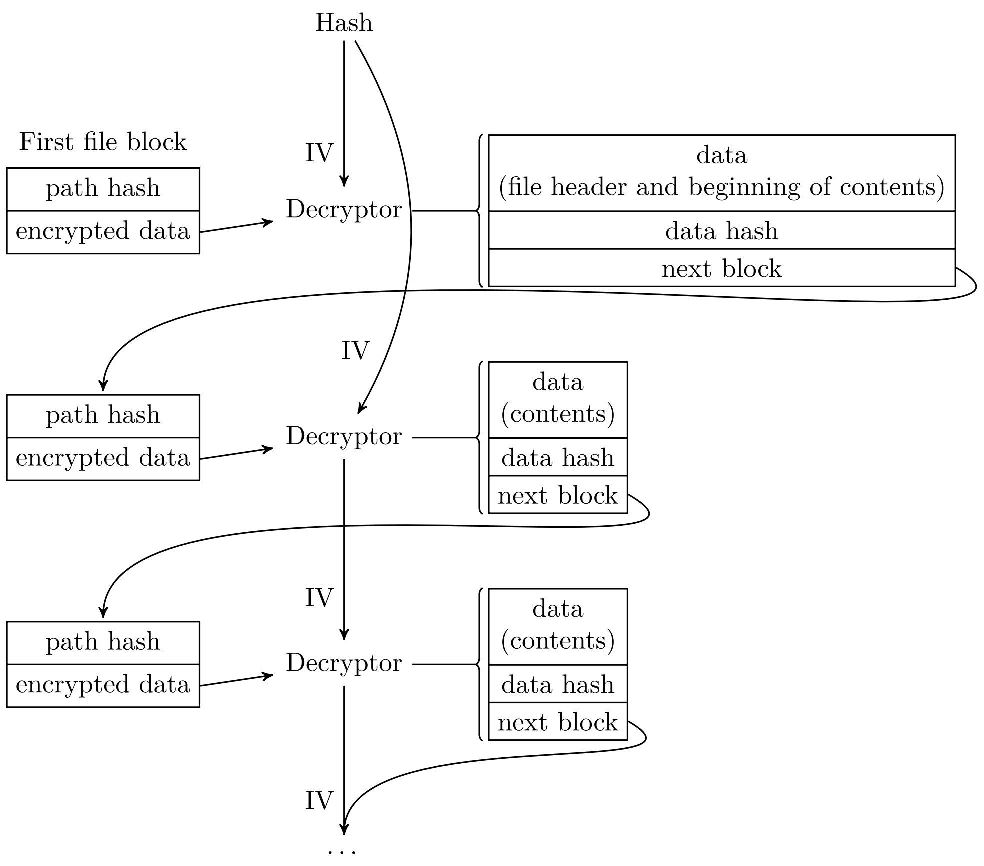

Figure 4: Structure of a file chain in stegfs. There are by default 8 copies of the chain for each file.

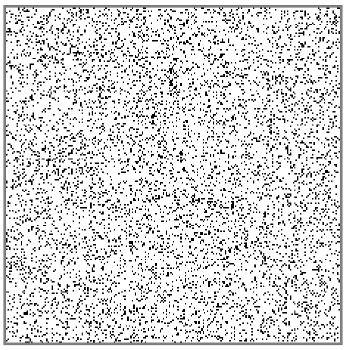

Figure 5: Difference between a stegfs container before and after writing to it. Each square is 2 kilobytes.

number of blocks have the same bytes in that position. This weakness can be easily remedied, by moving the hash into the encrypted part of the file header.

When two or more snapshots are available, however, similar problems to those experienced by VeraCrypt emerge. The differences between two snapshots are random, as shown by Figure 5, but each difference is always caused by a file modification. Specifically, given two snapshots, it is possible to work out the approximate amount of data that has been changed, and to decrypt those files when provided with the file name. An attacker can simply keep demanding file names from the victim and try decrypting all of the blocks that have changed, until none remain. The way stegfs overwrites files makes this more difficult, since the chain of file blocks can be broken, leaving orphaned blocks that cannot be reached. However, due to the simplicity of stegfs's file structure, orphaned blocks can be found by using intact chains to find the missing initialisation vector, and then simply trying to decrypt every block in the file system. Since every change corresponds to a user action, the attacker can prove that data exists that the victim has not revealed, breaking the deniability.

2.3 Rubberhose

Rubberhose (also known by its development name, Marutukku) is a proxy layer implementation for Linux and BSD [4]. Like VeraCrypt, it presents multiple virtual partitions which are mapped to a single encrypted partition. Unlike VeraCrypt, it supports an arbitrary number of virtual partitions, which are referred to as aspects. Each aspect has a translation table which randomises the storage location of data in the aspect, and this table is randomly altered each time a write occurs. In addition, random writes are performed while the partition is mounted [9, page 10], creating changes without relying on the user*. Provided that users use Rubberhose correctly and provide convincing cover material, Rubberhose should provide reasonable protection from differential attacks on its plausible deniability.

Unfortunately Rubberhose has not been updated since 2001, and so it is not compatible with modern Linux distributions. It can be compiled and mounted on a Linux 2.2 VM, such as Debian Potato, and used as a basic encryption layer. However, the lack of documentation meant that its deniable encryption facilities could not be tested, and so attacks against it may exist. The most obvious attack is to measure how long a file system was mounted and how much data was written it that time that was not written to the cover aspect(s). Some of these changes will be due to

*This could not be verified when tested.

random writes, even when the file system is used as a simple encrypted file system. However, if the amount of data changed does not scale linearly with the mount duration, hidden data has been written. It is possible that a random write distribution could be designed to provide enough anomalous behaviour to cover up writes to hidden aspects, but Rubberhose appears to use a simple uniform distribution [2].

3 Requirements

In this report I will design and implement a file system that fulfils the following criteria:

1. **Plausible deniability.** An attacker must not be able to prove or even detect with any confidence the presence of hidden data, regardless of what the user does, as long as the user does not deliberately betray the existence of the hidden data.
2. **Usability.** The file system must store the files, both hidden and cover, without corruption. The exception to this is when a user is forced to modify the file system under duress by an attacker. In this situation, corruption of hidden files is expected, since to the file system they do not exist. Lack of corruption is required if the file system is to be used by normal users in normal circumstances.
3. **Efficiency.** The file system must be fast, as well as efficient in CPU and disk usage. The file system needs to be fast enough to support normal workloads, such as watching HD videos and copying large images. The file system should also not use more disk space than needed for deniability. Setting a numeric limit is difficult and the acceptable amount of wasted space will vary between users. The efficiency must be above 50%, which is comparable with the efficiency provided by some RAID levels [16]. For more general usage the efficiency would have to be 90% or greater.

The attack model used for these requirements is the following, which should represent the file system running locally on an uncompromised system:

- The attacker can observe the initial file system state, and all subsequent changes on a byte level. The attacker can also observe the time that each change occurs.
- The attacker has a key that can decrypt any cover data present in the file system, which is obtained from the user.
- The attacker cannot observe reads, user interaction or gain any other information about the file system, the machine it is mounted on, or the user.
- The attacker cannot modify or interact with the file system. It is assumed that the user can store a hash of the file system on some out-of-band medium, such as a notebook, which can be used to verify the integrity of the file system.
- Cryptographic primitives are considered to be perfect, and the output of an encryption function is assumed to be indistinguishable from randomness when the attacker does not posses the encryption key.

4 Design

This section first describes the key design decisions made while designing the file system, and then walks though the completed design.

4.1 Design decisions

Under the attack model shown previously, the most important part of a deniable file system is how data is written back to disk. Since writes are often cached by file systems until enough changes have accumulated, and then written back to disk in one go, this section will consider this flushing operation to be an atomic unit. Section 4.2.1 shows how to implement the flush so that an attacker that observes the flush in more detail does not learn any additional information.

The first problem that must be considered is why a flush happens. If an attacker can find a flush that can only have been caused by a user writing hidden data, then plausible deniability has been broken. Each flush must be triggered by some event, either a user interaction or an environment change. An example of triggering a flush using a environment change would be to create flushes periodically using the system clock. However, a periodic flushing strategy either does not flush often enough to support high-bandwidth operations (such as copying videos), or flushes at a high frequency which is inefficient when only small changes are being made. Therefore for efficiency, flushes should be triggered when required by user interaction. The most common approach is to flush when the file system's cache becomes full, and when the file system is unmounted. Additionally, a flush may occur during quiet periods as an optimisation, but such flushes function in the same way as the other types, and so will not be considered. For a flush that is caused by user interaction to be deniable, it must contain some changes to the cover files, to provide a reason for its existence. The rest of the flush can either be random dummy data, or encrypted hidden data. This rule is not followed by VeraCrypt or stegfs, which is why they can be broken by a differential attack. Rubberhose can write data randomly without user input which can help provide some cover-like changes, but these changes may not be enough to cover up large quantities of changes to hidden aspects.

This leads to the second problem, how much data to flush. Several methods can be considered, based on different parameters:

Flush a constant amount of data, based on the size of the file system. This approach will provide deniability for a single flush, even when that flush contains a very small amount of cover data. However, if multiple such flushes occur in quick succession, an attacker can deduce that hidden data is being written, as fewer flushes could have been used if only cover data was being written. This problem can be overcome by limiting the number of flushes, either temporal rate-limiting or only allowing a constant number of flushes. However, this runs into the same limitations as environment-triggered flushes, namely bandwidth issues.

Flush a random amount of data. This approach has the same issues as the previous one, but adds an element of unpredictability which will confuse the user.

Flush an amount of data proportional to the amount of changed cover data. This approach overcomes the bandwidth issue, but requires that user changes large amounts of cover data whenever they want to change large amounts of hidden data. While this may be annoying for some users, it is good practice which should be enforced anyway. The other useful property of this approach is that the amount of data written is linear in the amount of cover data changed, unlike the other approaches, which is good for efficiency. This approach allows the ratio between the cover data and the space for hidden data to be set arbitrarily. However, this ratio must be constant and the same for all users of a particular file system. If users could adjust the ratio, users that do not store hidden data would favour ratios that add a minimum amount of padding, while users who do store hidden data would pick more balanced ratios. This would potentially provide attackers with a way of discerning which users are likely to have hidden data and which do not, breaking plausible deniability.

The last approach, flushing a proportional amount of data, is the most efficient and provides the most resistance to attacks, and so it is the one this report will implement. For simplicity, flushes will contains equal amounts of cover data and possible hidden data.

The next problem is where to write the data contained in a flush. For most file systems, the partition that backs the file system is divided into blocks, which are the smallest unit that the file system can write. In order to avoid any correlation between the data stored in particular blocks and that block's location, the data should be written to blocks at random locations. The simplest way of doing this is to select the required number of block locations at random. The problem with this is that some of the blocks may already contain data. These blocks would need to be written back at a different random position. While a flush generally has some padding blocks that could be replaced by the blocks that have to be moved, in general the flush size either needs to be increased or the block selection redone. If the flush size increases, the ratio property can be broken, betraying the presence of a block containing hidden data. Therefore block selection has to be repeated. However, this can also lead to problems. If the attacker is observing a mostly full file system, and measures the frequency that each block is overwritten, the attacker can deduce that certain blocks are less likely to picked and so must be in use. If that block cannot be decrypted by the cover key, then it must contain hidden data. There are several solutions for this, but each introduces extra restrictions:

Restrict the maximum space usage. In this approach, the maximum number of blocks in use is restricted, to prevent retries when selecting the new block locations. Let s be total file system size, u be number of blocks in use, f be the number of blocks to flush and w be the number of blocks to flush that contain data. The probability that block selection succeeds on the first try is ($\binom{n}{r}$ is the binomial coefficient):

$$\begin{aligned}
P(success) &= \frac{\sum_{i=0}^{f-w} \binom{u}{i}\binom{s-u}{f-i}}{\sum_{i=0}^{u} \binom{u}{i}\binom{s-u}{f-i}} \\
&= \frac{\sum_{i=0}^{f-w} \binom{u}{i}\binom{s-u}{f-i}}{\sum_{i=0}^{f} \binom{u}{i}\binom{s-u}{f-i}} && \text{since } \binom{u}{i} = 0 \text{ when } i > u \\
&= \frac{\sum_{i=0}^{f} \binom{u}{i}\binom{s-u}{f-i}}{\binom{s}{f}} && \text{by the Chu–Vandermonde identity} \qquad (1)
\end{aligned}$$

This function is plotted in Figure 6. It shows that for low usage values, the probability that block selection is restarted is quite low. However, Figure 6 shows an ideal situation, where a flush contains no hidden blocks. Clearly, the restrictions needed would make most of the disk space unused, which would contradict requirement 3.

Ask the user to delete some files when block selection fails. This approach is similar to the previous one, but allows the disk space usage restriction to adapt to the particular conditions for that instance of the file system. This may help, but does not fix the underlying efficiency

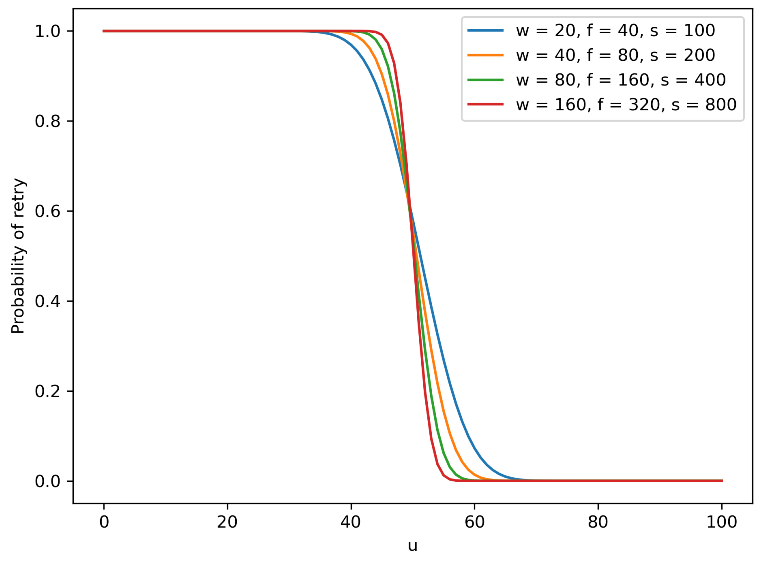

Figure 6: The probability that block selection is retried. As before, s is total file system size, u is number of blocks in use, f is the number of blocks to flush and w is the number of blocks in the flush that contain data.

problems. In addition, asking the user to delete things will annoy the user, and users are likely to change a minimal amount and then retry until the flush succeeds. This exposes them to the attacks this approach is meant to prevent. And overall, full space utilisation will not be achieved, contrary to requirement 3.

Discard used blocks when too many are found. This is the approach stegfs uses, and with the flushing techniques described above this should lead to a secure file system. The only problem is that data will be corrupted over time, which is contrary to requirement 2.

Add dummy hidden blocks so that overwrite frequency anomalies are normal. Instead of trying to avoid differences in the frequencies of block selection, this method simply makes the differences meaningless. In this approach, additional hidden blocks are created and filled with dummy data, so that the number of cover blocks and hidden blocks is always equal. This means that there are always hidden blocks present in the file system, even if the user does not have any hidden data. As a result, being able to deduce the locations of hidden blocks does not help the attacker. This solution means that a user that does not store any hidden data can only use half of the available disk space, which does cause problems for requirement 3. However, this solution only halves the usable disk space, unlike the first two solutions. Therefore this is the solution this report will use.

4.2 A complete design

With the main problems solved, a concrete design can now be presented. This design will expose a complete file system, not a set of virtual partitions, so that errors caused by the tight limits required by the flush ratio can be handled properly. In order to keep the design simple, the file system is divided into two layers: the lower layer which handles encryption, flushing and block allocation, and the upper layer which implements files and directories.

4.2.1 Block layer

The block layer transforms the backing storage partition into a pair of virtual partitions, which are exposed to the file layer. The file layer can perform block reads and writes though this interface, as well as allocating and deallocating blocks. The block layer handles the flushing operations automatically when its cache is full.

The backing partition is divided up into blocks, which are then used as one of three things: a cover mapping table, a hidden mapping table and the data area. The mapping tables translate between the physical blocks in the backing partition and the virtual blocks in the cover and hidden virtual partitions. The tables are simple lists of integer identifiers, each identifier corresponding to a single block in the data area. The identifiers allow the block layer to locate which physical block holds the data for a particular virtual block. For example, to locate a cover block with the identifier x, the cover table is searched until an entry equal to x is found. The index of that entry is then the position of the physical block in the data area. There are two special identifier values: -1 indicates that a block is unused, and is free to be allocated. -2 only occurs in the cover table, and indicates that the block is used by the hidden partition. If the block is marked as used in the hidden table, then a hidden block is stored there. If it is marked as unused in the hidden table, then that block is a padding block. An example of this mapping is shown in Figure 7.

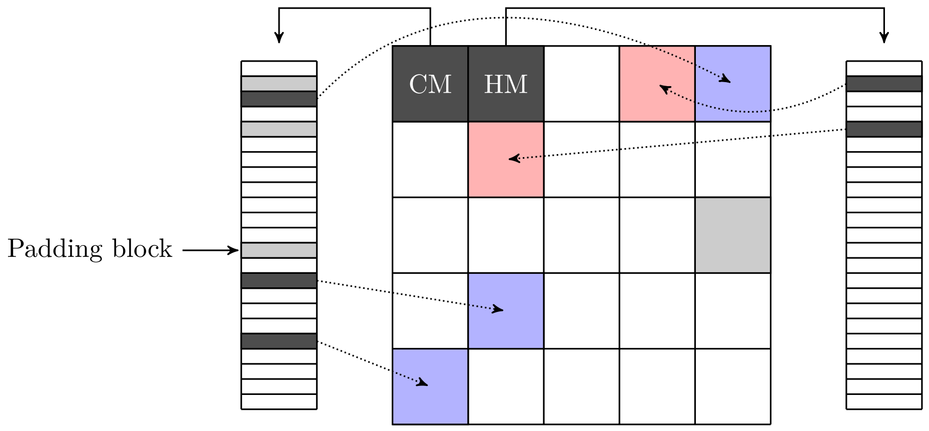

Figure 7: The block mapping tables. CM is the block that stores the cover mapping, while HM is the block that stores the hidden mapping. In a larger file system, each mapping would occupy multiple blocks. Lighter entries in the cover table are the hidden blocks. In the data area, blue blocks are blocks storing cover data, red blocks are blocks storing hidden data and light gray blocks are padding blocks.

Blocks are allocated by picking a random unused block from the data area and assigning to that block a unique identifier for the virtual partition that it has been allocated to. However, this is not written back until a flush occurs. A block is deallocated by removing it from the corresponding mapping table.

The block mapping tables are cached by the block layer, as well as a number of the data blocks. When the cache become full or the file system is unmounted, this data is written back in a flush operation. First, the number of hidden padding blocks is adjusted, adding or removing padding blocks as required, so that the total number of hidden blocks is equal to the total number of cover blocks. The number of modified blocks is then calculated, with additional hidden blocks being marked as modified in order to achieve the 1:1 change ratio required by the flush. Each of the modified blocks is reallocated, as assigning a new physical location prevents attackers from correlating which hidden blocks have changed. The data blocks are then written back in a random order, and finally the mapping tables are written back. The random order of the data block writing ensures that an attack that can view individual blocks being written does not learn anything about the flush that they could not learn from viewing the flush atomically.

Each block is encrypted using AES [15] with cypher-block chaining (CBC) mode, with a random 128 bit IV. CBC was chosen for its simplicity, and good security when used properly. CBC has several weaknesses that can be exploited by an attacker who can modify the file system, such as truncating messages or flipping bits in known positions. However, since the attack model used for this report does not permit the attacker to modify the file system, these weaknesses cannot be exploited. In order to prevent attackers viewing byte-level differences, a new IV is generated randomly each time the block is written. The block is encrypted using one of two 128 bit keys, one for cover blocks and one for hidden blocks. When the user does not want to store any hidden blocks, the hidden key is randomly generated. This prevents attackers demanding the hidden key as proof that no hidden data exists. This is also why hidden blocks are listed in the cover mapping. Since both users with and without a hidden key can view where the hidden blocks are, they will act in the same way when allocating new blocks and flushing, and so cannot be distinguished by the attacker.

An improvement that could be made here is marking unmodified cover blocks as modified. This can be done in every flush to increase the amount of cover changes, so that more hidden data can be changed in each flush. As long as the amount of additional cover changes does not depend on the presence or absence of hidden data, the security properties of the file system are unaffected. However, this will affect efficiency in most cases, since more data is written for the same amount of cover changes. Since there will usually be more cover data than hidden data, this improvement is not used in this report.

4.2.2 File layer

The file layer is implemented in the same way as any other file system, since the deniability and security is handled by the block layer. There are two types of objects that the file system needs to handle: files and directories. Files and directories are differentiated by the first byte of the header, which is F for files and D for directories. Each virtual partition has a root directory, and its header is stored in block 0. Since all file system paths start from the root directory, this allows all of the objects to be located.

Directories are implemented using a B-tree. B-trees were first invented by Bayer and McCreight [5] for use in databases, and provide a tree data structure that supports logarithmic search and insertion that is optimised for storage as blocks. Items in the directory are stored using the file name, which is a null-terminated string with a maximum length of 255, and a pointer to the block containing the item's header. Directories can also be traversed, allowing directory listings. The root node of the B-tree is stored in the directory header, while the other nodes each occupy a single block.

Files are implemented as a tree of block identifiers, which will be referred to as a block tree. The block tree allows for compact storage of a sequence of integers, and provides logarithmic finding and insertion. It is similar to a B-tree, except that no keys are stored and the only insertion operation is an append. Figure 8 shows an example of a block tree. Each file stores the number of blocks in the block tree, the length of the file, the root node of the block tree, and the start of the file's contents. This means that small files only occupy one block. To read or write to a file, the block containing the relevant data is looked up using the block tree, and then read from or written to the corresponding block.

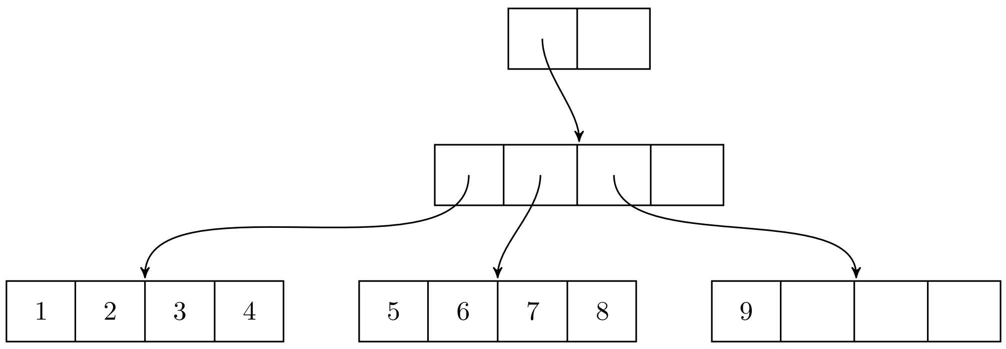

Figure 8: An example block tree. The root node is part of the file header. The leaf nodes contain the block identifiers that store the file contents.

5 Implementation

The file system is implemented in C++20 using a modular programming style. A diagram of the modules and how data flows between them is shown in Figure 9.

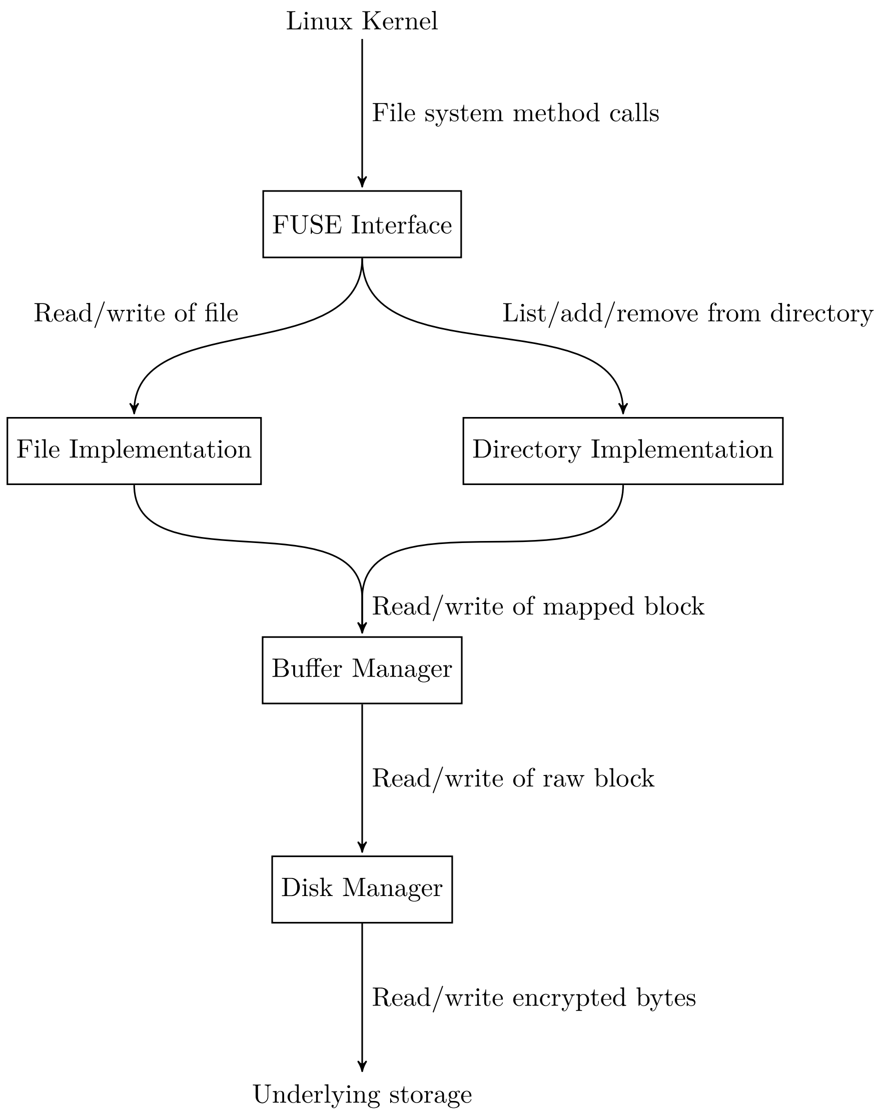

Figure 9: Overview of the modular structure of the file system implementation, as well as the data flow between those modules.

In this section, the implementation of each module will be described, with a focus on the details that are implementation-specific and not covered in section 4.

5.1 Disk manager

The disk manager is responsible for reading and writing the encrypted physical blocks. The file system uses a block size of 4KiB (4096 bytes). Since the IV uses the first 16 bytes, this leaves 4080 bytes in each block for data storage. Encryption is done using Crypto++ [7], an open-source implementation of many common cryptographic algorithms. IVs are generated using the cryptographically secure pseudorandom number generator (CSPRNG) provided by Crypto++.

5.2 Buffer manager

The buffer manager is responsible for providing access to cover and hidden blocks. It performs translation between physical blocks and virtual blocks, and handles caching, concurrency management, and flushing. Blocks are identified using 32-bit integer identifiers, which restricts the maximum file system size to about 16 TiB.

When the file system is mounted, the buffer manager loads the mapping tables from the start of the partition and verifies that the tables pass a sanity check. The manager then allows other modules to access blocks from the virtual partitions using `BlockAccessor` objects. These objects lock the relevant cache location while alive, preventing other threads from modifying that block. When a block is requested, it is loaded into the cache. The cache uses a least-recently-used (LRU) replacement policy, and can evict cache entries which are not in use and have not been modified. The buffer also has a set of methods which allow operations to specify how much cache space they require. If the cache is too full, these operations are blocked until all concurrent operations have finished, and then the cache is flushed. These methods also prevent operations that could allocate or modify too many hidden blocks from proceeding. This ensures that flushing is always possible.

Random numbers used by the buffer manager are generated using the CSPRNG provided by Crypto++.

5.3 File implementation

As described in section 4.2.2, files are implemented using a block tree. The file header includes the block tree header, which contains 8 pointers. The header also includes the number of blocks in the block tree and the size of the file. These values are all 32-bit integers, so the maximum size of a file is 4 GiB. Each node of the block tree except the root node contains up to 1020 pointers, which give a very high fan-out. This header structure is shown in Figure 10.

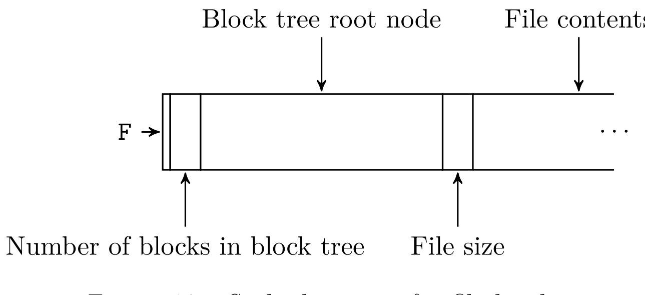

Figure 10: Scale diagram of a file header.

5.4 Directory implementation

Directories are slightly simpler than files, since the B-tree is the only thing that they contain. A directory header contains the number of blocks in the B-tree as well as the B-tree's height, and then the root node of the B-tree. This is shown in Figure 11. Each node of the B-tree can hold 15 entries, which gives a decent fan-out. More advanced techniques such as variable key sizes and key compression could increase fan-out, but were not implemented for this report.

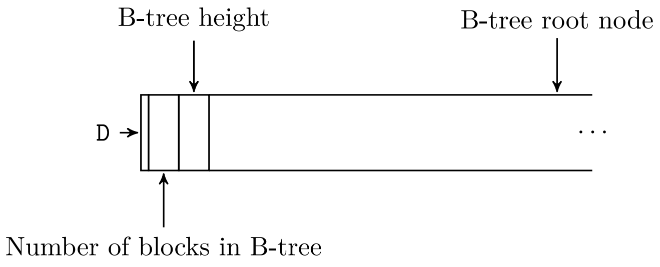

Figure 11: Scale diagram of a directory header.

5.5 FUSE interface

The last module of the file system is the FUSE interface. FUSE (File System in Userspace) is a kernel module [10] and userspace library [14] that allows file systems to be implemented in userspace instead of as a kernel module. This library is used to make the file system accessible as part of the normal Linux file system. Normal programs can then natively use this new file system without any modifications.

The FUSE interface exposes a pair of directories, named `hidden` and `cover`. When the user does not provide a key for the hidden part, the hidden directory simply mirrors the cover directory. This means that an attacker who finds a hidden path referenced by some other source, such as an applications "recently used" list, cannot prove that the hidden path is indeed hidden. All of the basic system calls are implemented, allowing mostly normal usage by external programs. Each operation starts by calling the operation methods on the buffer with the predicted maximum cache usage and amount of changes. When cache space is not available, the operation is delayed until a flush can be performed. If the operation would exceed the restrictions for a flush, the FUSE interface returns the `EPERM` error code to the kernel.

5.6 Mounting the file system

The final part to cover is the user interface. The file system compiles to a single executable, which supports two run modes: `mount` and `init`.

When `init` is selected, the executable creates a new file to hold the file system's backing partition. The user can specify how many blocks the file system should contain. The executable then writes randomness over the entire file system to initialise it, and then creates the root directories for each virtual partition. The randomness used comes from the Linux kernel's non-blocking random pool, which is exposed at `/dev/urandom`.

When `mount` is selected, the executable mounts the file system at the provided mount point. The `cover` and `hidden` folders appear inside that mount point. The executable stays running in the foreground until the file system is unmounted.

```
Usage:
    fs mount <fname> <path> [--debug] [--cache-size=<cache-size>] [--no-hidden]
    fs init <fname> <numBlocks> [--debug] [--cache-size=<cache-size>] [--no-hidden]
    fs (-h | --help)
    fs --version

Options:
    -h --help                       Show this screen.
    --version                       Show version.
    -c, --cache-size=<cache-size>   Size of file system cache in blocks [default: 1024].
```

Figure 12: Help message provided by the file system executable. Argument parsing is done by docopt [8].

6 Evaluation

This section evaluates whether the requirements detailed in section 3 have been fulfilled. Requirement 1 is covered by section 6.1, while requirement 3 is covered by 6.2. Requirement 2 was evaluated as part of testing, and so is not covered explicitly.

6.1 Security

The security properties of the file system derive quite straightforwardly from the statements made in section 4. Given a single snapshot, an attacker who knows the key for the cover data can learn the following information:

- How many cover blocks there are, where they are and their contents.
- How many hidden blocks there are, and where in the file system they are. However, the number of hidden blocks is always the same as the number of cover blocks, and the locations are random.

When the attacker has additional snapshots, they can find the set of flushes that occurred. Figure 13 shows an example of a series of snapshots. Each flush allows them to deduce the following information:

- How many cover blocks were changed / allocated, where those blocks are, which blocks were deallocated and the contents of those allocated and deallocated blocks.
- How many hidden blocks were changed / allocated, where those blocks are and which blocks were deallocated. However, the number of changed hidden blocks is the same as the number of changed cover blocks, and as before the locations are random.

The attacker does not know the identifiers used by the hidden partition, and this prevents them analysing the frequency at which each virtual block changes. If the attackers could deduce the identifiers, they could analyse the change patterns and show that they are not uniform, and therefore break plausible deniability. However, since the hidden identifiers are stored in a separate mapping table encrypted with the hidden key, this is not possible.

In addition, a user that does not store any hidden data in the file system does not know the key to the hidden partition. Therefore they cannot prove that they have no hidden data, unless they deliberately use a hidden key and turn that over. Since all the information the attacker can gather is either random or mirrors cover data that they already know, plausible deniability cannot be broken by a differential attack.

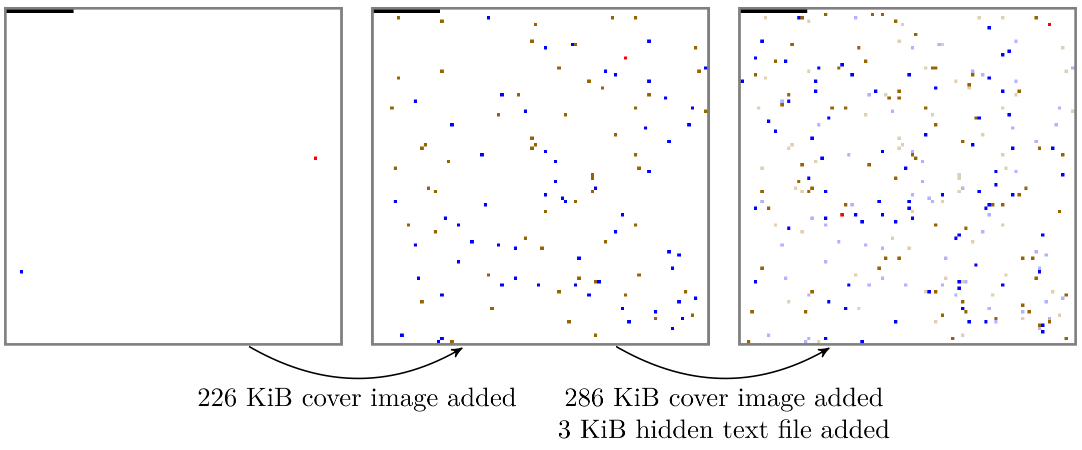
226 KiB cover image added
286 KiB cover image added
3 KiB hidden text file added

Figure 13: A series of snapshots of a file system instance with 10,000 blocks (38.9 MiB). Blue blocks are cover blocks, red blocks are hidden blocks, and brown blocks are padding blocks. An attacker would not be able to distinguish between hidden and padding blocks. Faded blocks are blocks that have not changed since the previous snapshot. The black blocks at the top of each snapshot are the blocks that contain the mapping tables.

6.2 Efficiency

While efficiency is not the main goal of this report, a reasonable level of performance is required if normal users are to adopt this file system. The following tests were executed on a modern laptop with a 1 TB SSD. Each test was executed 20 times and an average taken.

Test	Deniable File System	Normal File System
Creation of 1000 empty files (files/second)	5147	38390
Write speed of a 10 MiB file (MiB/second)	53.76	1145
Read speed of a 10 MiB file (MiB/second)	249.3	771.7

The performance tests show that while the deniable file system is slower, it is still fast enough to satisfy requirement 3. The file system has not been profiled or optimised in any way, and so it is likely that improvements could be made if required. Additionally, the deniable file system was backed by a file on the normal file system in these tests. If the deniable file system was instead backed directly by a partition on a storage device, speeds should increase. However on Linux, writing directly to storage devices requires root privileges, and so the setup used for these tests reflects the setup most users would use.

The other part of requirement 3 is the efficiency of disk usage. When both cover data and hidden data are stored, the disk efficiency approaches 100%. For example, the large file created in the write speed test (which has a size of 10 MiB) used 2575 blocks (which is 10.06 MiB), which is an efficiency of 99.4%. However, a normal user will store far more cover data than hidden data, and may not store any hidden data at all. For these users, disk usage efficiency will typically be about 50%, since the other 50% of the file system is filled with hidden padding blocks. This is suboptimal, and both Rubberhose and VeraCrypt achieve better storage densities, although this comes at the cost of security. However, this is better than stegfs, which stores each file 8 times. Thus stegfs will have a disk usage efficiency of at most 12.5%, which is further reduced due to the Birthday Paradox.

6.3 Usability

While not an explicit requirement, the file system must be easy to use and not provide too many hindrances, otherwise users would simply switch to another, less secure file system. When cover data is being manipulated, the file system acts like a normal file system, and so usability is not impacted at all. However, when hidden data is changed and there is not a sufficient amount of cover changes, an error is returned. This interrupts the user's workflow, as they must change tasks and generate some more cover changes to be able to continue. This problem is exacerbated by the file system's implementation, which uses overly large estimates when deciding whether an operation should proceed. The unexpected errors may cause some programs to simply crash, although most will simply ask the user to retry. The errors do present a usability challenge, and a better workaround has not been found yet. One potential solution was to hang on writes and wait for enough cover changes to accumulate before resuming the hidden write. However, when tested with the Plasma desktop, a write caused by pasting a file into the cover area could cause the file manager to freeze, which would in turn freeze the desktop as well. Therefore the compromise reached was to return an error. VeraCrypt acts in a similar way when writing to the cover volume, when hidden volume protection is activated. The file system implemented in this report only returns errors when writing to hidden files, which is a slight improvement.

7 Conclusions

This report demonstrates that differential attacks are effective in finding hidden files in existing file systems that provide plausible deniability. Using the lessons learnt in performing these attacks, I have successfully designed and implemented a new file system that allows users to store hidden files, even when an attacker can access multiple different copies of the same file system. The new file system can store both cover and hidden files in a 1:1 ratio, with reasonable disk usage efficiency. The file system is fast enough to be used in place of existing solutions, and during normal use it does not impact a user's workflow.

7.1 Limitations

While the file system implemented in this report is ready for use, it does have some limitations that could be remedied in future iterations:

Disk usage. As detailed in section 6.2, a user that does not store any hidden data will only fill at most half of their disk with useful data. This makes the general adoption required for convincing deniability hard, since most users will not use a file system that wastes half of their disk space. However, increasing the cover to hidden ratio used by the file system from 1:1 to a larger ratio such as 100:1 may be enough to encourage general usage.

Usability. While the file system is very usable when cover files are being modified, writing hidden files is more difficult. Since the file system returns errors when an operation would create too many hidden changes for a flush to occur, user actions such as saving a file can be met with a cryptic error dialog. This is exacerbated by the overly cautious estimates made by the file system when evaluating whether an operation can proceed. Since these errors only occur with hidden files, this does not affect the overall usability of the file system. However, using a friendlier approach or just improving the estimates used would improve the experience for users who wish to store hidden files.

Lack of multiple partition support. Like most implementations of deniable file systems, the file system implemented in this report only supports two types of files: cover files and hidden files. While this is secure and usable, some benefits can be achieved by allowing multiple hidden partitions, so that some hidden files can be revealed without revealing the most sensitive files.

7.2 Future work

There are multiple avenues for future work, which are focused on fixing or at least alleviating the limitations of the file system:

Reduce number of errors when writing to hidden files by using transactions. The file system currently estimates how many blocks will be changed, which is fragile and is tightly coupled to the implementation used by other parts of the file system. In addition, the estimates currently used are based on the worst case, and so grossly overestimate when small files are modified. If the file system used a transaction-based approach, operations would not need estimates. Rather, an operation would initially be allowed to proceed and, when a change constraint is violated, the transaction could be paused or rolled back. This would require changes to the buffer manager, but would simplify the operation code significantly.

Research alternative ways of handling constraint violations. Currently, an operation that would change too many hidden blocks returns an error to the operating system. It would be more user friendly is the operation was simply paused, and resumed once enough cover material has been changed. This would also allow large writes to hidden files to be automatically broken across multiple flushes, each bit being written as cover changes become available. The simple approach of simply blocking when an operation cannot proceed currently causes file managers such as KDE's Dolphin to freeze, along with the desktop itself. It is possible that certain flags could be set to indicate that the file system may freeze, and so avoid these issues.

Support multiple aspects. The file system, as described in this report, only has one hidden partition, which means that revealing the key to the hidden partition reveals all of the hidden data at once. As described by Assange in [3], there are advantages to having multiple hidden aspects, since they can slowly be revealed, tricking the attacker into thinking that all of the hidden data has been revealed when some still remains. It also allows interesting use cases, such as sharing a single file system containing many aspects with a group of people, each of which only knows the key to a subset of the aspects. This allows the group to compartmentalise information between themselves, reducing the risk presented by an insider or defector.

This could be done in two ways: The hidden blocks could be divided into several aspects, and a flush simply ensures that the correct number of blocks is changed in each aspect. This solution is similar to the approach used by Rubberhose. An alternative design could be a recursive implementation, where each hidden aspect may contain its own hidden aspect. This establishes a hierarchy of hidden aspects, requiring the keys of all of the parent aspects to access a deeply nested one. This approach would hide the number of hidden aspects, unlike the first approach.

Research the best cover to hidden ratio for general use. The file system design and implementation presented in this report uses a 1:1 ratio for simplicity. However, this is not the best ratio and a cover-heavy ratio will be more useful in practice. As mentioned in section 4.1, the ratio must be not be user modifiable, and so a general sample of the target user population would have to be analysed to find the best compromise between users who do not want to store any hidden files and those who do.

Extend the file system to protect against stronger attackers. The attack model used in this report does not permit the attacker to observe reads. This restriction makes sense when applied to a normal file system, which is backed by local storage. However, situations exist where the backing storage is controlled by a potential attacker, such as cloud storage. In these situations, an attacker who observes a read can look up which block was read, and check if that block was a cover block. If it was not, then there must be hidden data present, since a file system without hidden data does not normally read unallocated blocks. In order to remain secure, the file system implemented in this report would have to read and write from a local copy of the file system, which is then uploaded to the cloud storage. A better design would remove the need for a local copy, and would allow a system with constrained storage to read and write to file systems far too large to store locally.

Applying the rules detailed in the design section, a read-observable file system would have to batch reads in a similar way to flushes, pairing each cover read with a possible hidden read. This would exacerbate the freezing problem, since every hidden read could result in a hang, while the file system waits for a cover read. This design would also allow attackers to record the frequency that each hidden block is read. A non-uniform access pattern would indicate the presence of hidden data, and such a pattern will occur since directories are accessed more than files. Therefore the techniques used in this report are not sufficient, and further research needs to be done.

7.3 Reflections

The initial idea for this project, creating a more resistant file system, came about when I was implementing a simple encrypted file system in Python. As such, this project has allowed me to delve into the details behind deniable encryption and the issues and design challenges faced when creating a file system that supports plausible deniability. This project also allowed me to implement some of the high performance data structures, such as B-trees, that had been covered as part of the Database Systems Implementation course in functional code, instead of implementing the data structures in standalone, proof of concept code. The analysis part of the project allowed experimentation in using VMs, as well as some digital archeology, especially when running Rubberhose on a 20 year old version of Linux. The cryptographic element of this project allowed me to put the knowledge gained during the Security course to use.

The actual implementation of the file system went smoothly, and took about a week and a half. It is implemented in C++, a language which I am familiar with, and consists of around 3,500 lines of code, which is quite compact for a full file system implementation*. No major design flaws were encountered during implementation, although the design did evolve during the implementation phase. These changes were mainly to simplify the design and increase code sharing. The finished implementation is reasonable clean and efficient, although the locking strategy could be improved. Currently, the locks used are non-recursive, and so when a thread attempts to access a block it has already locked, a deadlock occurs. Most of these deadlocks were caused by tree code reading block identifiers from the wrong location, and so the deadlocks offer an easy way to spot and debug the errors. However some code, such as the code handling file renaming, is made more complicated since the same directory cannot be opened twice. An improved implementation would use separate read and write locks, simplifying the implementation and increasing concurrency for non-modifying operations.

8 Acknowledgements

I would like to thank my supervisor, Prof. Michael Goldsmith, for his help in guiding this project, as well as reviewing this report. I would also like to thank my two proofreaders, David and Francesca Joyce, who pointed out some grammatical and spelling errors.

*stegfs is about 5,000 and Rubberhose is 15,000+. Both are implemented in C.

9 References

[1] "albinoloverats". stegfs – a fuse based steganographic file system. Online at https://github.com/albinoloverats/stegfs/tree/8ddd255 and https://albinoloverats.net/projects/stegfs; version stegfs-2015.08.1-27-g8ddd255, 2018.

[2] J. Assange. Julian assange on rubber-hose proof (cryptographically deniable) file-systems (1997). Online at https://web.archive.org/web/20190224133115/https://lists.cpunks.org/pipermail/cypherpunks/2016-September/032235.html; archived on the Wayback Machine 24/02/2019.

[3] J. Assange. Julian assange: Physical coercion. Online at https://web.archive.org/web/20110816230712/https://embeddedsw.net/doc/physical_coercion.txt; archived on the Wayback Machine 16/08/2011.

[4] J. Assange, R. Weinmann, and S. Dreyfus. Rubberhose – marutukku.org. Online at https://web.archive.org/web/20120716034441/http://marutukku.org/; archived on the Wayback Machine 16/07/2012.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 107–141, 1970.

[6] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. *WPES'04: Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, pages 77–84, 2004.

[7] Crypto++® Library 8.2. Online at https://www.cryptopp.com/; accessed 02/05/2020.

[8] docopt.cpp: A C++11 Port. Online at https://github.com/docopt/docopt.cpp; accessed 02/05/2020.

[9] S. Dreyfus. The idiot savants' guide to rubberhose. Online at https://web.archive.org/web/20110519065709/http://iq.org/~proff/rubberhose.org/current/src/doc/maruguide.pdf; archived on the Wayback Machine 19/05/2011.

[10] FUSE – The Linux Kernel documentation. Online at https://www.kernel.org/doc/html/latest/filesystems/fuse.html; accessed 02/05/2020.

[11] C. Hargreaves and H. Chivers. Detecting hidden encrypted volumes. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6109 LNCS:233–244, 2010.

[12] K. Karampidis, E. Kavallieratou, and G. Papadourakis. A review of image steganalysis techniques for digital forensics. *Journal of Information Security and Applications*, 40:217–235, 2018.

[13] Y. Kim. Border Searches of Laptop Computers and Other Electronic Storage Devices. Online at https://fas.org/sgp/crs/homesec/RL34404.pdf; accessed 07/05/2020.

[14] libfuse. Online at https://github.com/libfuse/libfuse; accessed 02/05/2020.

[15] National Institute of Standards and Technology. Announcing the ADVANCED ENCRYPTION STANDARD (AES). Online at https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf; accessed 09/05/2020, 2001.

[16] Oracle. Selecting the Best RAID Level. Online at https://docs.oracle.com/cd/E19871-01/820-1847-20/appendixf.html#50515995_80148; accessed 14/05/2020, 2010.

[17] The X3DH Key Agreement Protocol. Online at https://signal.org/docs/specifications/x3dh/#deniability; accessed 07/05/2020.

[18] VeraCrypt – Free Open source disk encryption with strong security for the Paranoid. Online at https://www.veracrypt.fr/en/Home.html; accessed 27/04/2020.

[19] VeraCrypt Documentation – Hidden Volume. Online at https://www.veracrypt.fr/en/Hidden%20Volume.html; accessed 27/04/2020.

[20] VeraCrypt Documentation – Protection of Hidden Volumes Against Damage. Online at https://www.veracrypt.fr/en/Protection%20of%20Hidden%20Volumes.html; accessed 27/04/2020.

10 Appendices

10.1 Code listing

This section contains some of the more interesting parts of the file system implementation. The files listed are:

- `consts.hpp` – Contains a collection of constants specifying the size of individual blocks, encryption key size and so on.
- `disk.[hc]pp` – The disk manager, which handles block encryption, reading and writing.
- `buffer.[hc]pp` – The buffer manager, which handles flushing and concurrency.
- `dir.hpp` – The interface provided for modifying and querying directories.
- `file.hpp` – The interface for reading and writing to files.

The full source code can be found on GitHub under the name “dardefs”.

consts.hpp

```cpp
#ifndef CONSTS_HPP
#define CONSTS_HPP

const unsigned int PHYSICAL_BLOCK_SIZE = 4096;
const unsigned int CIPHER_BLOCK_SIZE = 16;
const unsigned int IV_SIZE = CIPHER_BLOCK_SIZE;
const unsigned int KEY_SIZE = 16;
const unsigned int LOGICAL_BLOCK_SIZE = PHYSICAL_BLOCK_SIZE - IV_SIZE;

const unsigned int BLOCK_POINTER_SIZE = 4;
const unsigned int MAPPING_POINTERS_PER_BLOCK = LOGICAL_BLOCK_SIZE / BLOCK_POINTER_SIZE;

const unsigned char FILE_TYPE = 'F';
const unsigned char DIR_TYPE = 'D';

const unsigned int NUM_HEADER_BLOCK_TREE_ENTRIES = 8;
const unsigned int NUM_TREE_BLOCK_TREE_ENTRIES = LOGICAL_BLOCK_SIZE / BLOCK_POINTER_SIZE;
const unsigned int BLOCK_TREE_OFFSET = 1;

const unsigned int DATA_OFFSET = BLOCK_TREE_OFFSET + BLOCK_POINTER_SIZE + BLOCK_POINTER_SIZE * NUM_HEADER_BLOCK_TREE_ENTRIES;
const unsigned int DATA_SIZE = LOGICAL_BLOCK_SIZE - DATA_OFFSET;
const unsigned int FILE_HEADER_SIZE = BLOCK_POINTER_SIZE;

const unsigned int FILE_NAME_SIZE = 255;
const unsigned int BTREE_RECORD_SIZE = FILE_NAME_SIZE + BLOCK_POINTER_SIZE;

const unsigned int DIR_LOOKUP_COST = 10;
const unsigned int DIR_WRITE_COST = DIR_LOOKUP_COST * 2;

const unsigned int FILE_LOOKUP_COST = 4;

#endif // CONSTS_HPP
```

disk.hpp

```cpp
#ifndef DISK_HPP
#define DISK_HPP

#include <string>
#include <tuple>
#include <fstream>
#include <mutex>

#include "types.hpp"

enum class BlockMappingType {
    COVER,
    HIDDEN,
    NEITHER
};

class Disk {
    std::fstream file;
    std::mutex file_lock;
    unsigned int number_of_blocks;
    unsigned int file_size;
    secure_string cover_key, hidden_key;

    void readRawBlock(unsigned int location, secure_string& out);
    void writeRawBlock(unsigned int location, const secure_string& in);
    void decryptBlock(const secure_string& in, secure_string& out, bool hidden) const;
    void encryptBlock(const secure_string& in, secure_string& out, bool hidden) const;

public:
    Disk(std::string fname, secure_string cover_key, secure_string hidden_key);

    void readBlock(unsigned int location, bool hidden, secure_string& buffer);
    void writeBlock(unsigned int location, bool hidden, const secure_string& buffer);

    unsigned int numberOfBlocks() const;
};

#endif // DISK_HPP
```

disk.cpp

```cpp
#include "disk.hpp"

#include "cryptopp/modes.h"
#include "cryptopp/aes.h"
#include "cryptopp/filters.h"
#include "cryptopp/osrng.h"

#include "consts.hpp"
#include "utilities.hpp"

#include <iostream>


Disk::Disk(std::string fname, secure_string cover_key, secure_string hidden_key) :
        cover_key(cover_key), hidden_key(hidden_key) {
    ensure(cover_key.size() == KEY_SIZE, "Disk::Disk") << "Cover key is the wrong size";
    ensure(hidden_key.size() == KEY_SIZE, "Disk::Disk") << "Hidden key is the wrong size";

    // unbuffered
    file.rdbuf()->pubsetbuf(0, 0);
    file.open(fname, std::ios::ate | std::ios::out | std::ios::in | std::ios::binary);
    ensure(!!file, "Disk::Disk") << "File could not be opened";
    file_size = file.tellg();
    ensure(file_size % PHYSICAL_BLOCK_SIZE == 0, "Disk::Disk") << "File size is not a multiple of the block size";
    number_of_blocks = file_size / PHYSICAL_BLOCK_SIZE;
}
```

```
void Disk::readRawBlock(unsigned int location, secure_string& out) {
    std::lock_guard<std::mutex> guard(file_lock);

    file.seekg(location);
    file.read(reinterpret_cast<char*>(&out[0]), out.size());

    ensure(static_cast<unsigned int>(file.gcount()) == out.size(), "Disk::readRawBlock") << "Did not read enough bytes";
}

void Disk::writeRawBlock(unsigned int location, const secure_string& in) {
    std::lock_guard<std::mutex> guard(file_lock);

    file.seekg(location);
    file.write(reinterpret_cast<const char*>(&in[0]), in.size());
}

void Disk::decryptBlock(const secure_string& in, secure_string& out, bool hidden) const {
    ensure(in.size() % CIPHER_BLOCK_SIZE == 0, "Disk::decryptBlock") << "Input is not a multiple of the cipher block size";
    ensure(out.size() == in.size() - IV_SIZE, "Disk::decryptBlock") << "Output is not the correct size";

    CryptoPP::CBC_Mode<CryptoPP::AES>::Decryption d;
    if (hidden) {
        d.SetKeyWithIV(&hidden_key[0], hidden_key.size(), &in[0]);
    }
    else {
        d.SetKeyWithIV(&cover_key[0], cover_key.size(), &in[0]);
    }

    CryptoPP::StringSource ss(&in[IV_SIZE], LOGICAL_BLOCK_SIZE, true,
        new CryptoPP::StreamTransformationFilter(
            d,
            new CryptoPP::ArraySink(&out[0], LOGICAL_BLOCK_SIZE),
            CryptoPP::StreamTransformationFilter::NO_PADDING
        )
    );
}

void Disk::encryptBlock(const secure_string& in, secure_string& out, bool hidden) const {
    ensure(in.size() % CIPHER_BLOCK_SIZE == 0, "Disk::encryptBlock") << "Input is not a multiple of the cipher block size";
    ensure(out.size() == in.size() + IV_SIZE, "Disk::encryptBlock") << "Output is not the correct size";

    CryptoPP::CBC_Mode<CryptoPP::AES>::Encryption e;

    CryptoPP::OS_GenerateRandomBlock(false, &out[0], IV_SIZE);

    if (hidden) {
        e.SetKeyWithIV(&hidden_key[0], hidden_key.size(), &out[0]);
    }
    else {
        e.SetKeyWithIV(&cover_key[0], cover_key.size(), &out[0]);
    }

    CryptoPP::StringSource ss(&in[0], LOGICAL_BLOCK_SIZE, true,
        new CryptoPP::StreamTransformationFilter(
            e,
            new CryptoPP::ArraySink(&out[IV_SIZE], LOGICAL_BLOCK_SIZE),
            CryptoPP::StreamTransformationFilter::NO_PADDING
        )
    );
}

void Disk::readBlock(unsigned int location, bool hidden, secure_string& buffer) {
    ensure(buffer.size() == LOGICAL_BLOCK_SIZE, "Disk::readBlock") << "Output is not the correct size";

    secure_string physical_block_buffer(PHYSICAL_BLOCK_SIZE, '\0');

    readRawBlock(location * PHYSICAL_BLOCK_SIZE, physical_block_buffer);
    decryptBlock(physical_block_buffer, buffer, hidden);
}

void Disk::writeBlock(unsigned int location, bool hidden, const secure_string& buffer) {
    ensure(buffer.size() == LOGICAL_BLOCK_SIZE, "Disk::writeBlock") << "Input is not the correct size";

    secure_string physical_block_buffer(PHYSICAL_BLOCK_SIZE, '\0');

    encryptBlock(buffer, physical_block_buffer, hidden);
    writeRawBlock(location * PHYSICAL_BLOCK_SIZE, physical_block_buffer);
}

unsigned int Disk::numberOfBlocks() const {
    return number_of_blocks;
}
```

buffer.hpp

```
#ifndef BUFFER_HPP
#define BUFFER_HPP

#include <map>
#include <vector>
#include <deque>
#include <mutex>
#include <set>
#include <thread>

#include "types.hpp"


class Disk;
class Buffer;

const unsigned int NO_BLOCK_ASSIGNED = -1;
const unsigned int NO_CACHE_LOC_ASSIGNED = -1;

struct BlockMappingInfo {
    unsigned int physical_block_id = NO_BLOCK_ASSIGNED;
    unsigned int cache_location = NO_CACHE_LOC_ASSIGNED;
};

struct BlockCacheEntry {
    secure_string data;
    std::pair<bool, unsigned int> logical_block_id = {false, NO_BLOCK_ASSIGNED};
    bool dirty = false, dirtied_by_current = false;
    std::mutex lock;

    BlockCacheEntry();
};

class BlockAccessor {
    Buffer& buffer_;
    BlockCacheEntry& cache_entry;
    bool moved = false;

public:
    BlockAccessor(Buffer& buffer, BlockCacheEntry& cache_entry);
    BlockAccessor(BlockAccessor&& other);
    ~BlockAccessor();

    const secure_string& read() const;
    secure_string& writable();
    std::pair<bool, unsigned int> block_id() const;
    Buffer& buffer() const;
};

struct BufferOperationData {
    unsigned int max_blocks;
    char hidden = 2;
    std::set<unsigned int> blocks = {};
    unsigned int block_requests = 0, block_writes = 0, max_cache_takeup = 0;
};

class BufferOperation {
    Buffer& buffer;

public:
    BufferOperation(Buffer& buf);
    BufferOperation(BufferOperation&) = delete;
    BufferOperation(BufferOperation&&) = delete;
    ~BufferOperation();

    friend class Buffer;
};

class Buffer {
    Disk& disk;
    std::mutex lock;
    std::map<std::pair<bool, unsigned int>, BlockMappingInfo> block_mapping;
    unsigned int max_cover_id = 0, max_hidden_id = 0, number_of_mapping_blocks = 0;
    unsigned int cover_blocks_allocated = 0, hidden_blocks_allocated = 0, reserved_cache_space = 0;
    unsigned int cover_blocks_changed = 0, hidden_blocks_changed;
    std::vector<unsigned int> unallocated_list, virtual_list;

    std::vector<std::pair<bool, unsigned int>> reverse_block_mapping;
    std::vector<BlockCacheEntry> cache;
    std::deque<unsigned int> least_recently_used;
    std::map<std::thread::id, BufferOperationData> operation_for_thread;
    bool enforce_operations, debug, no_hidden;
    std::mutex wait_for_ops_done, wait_for_flush_done;
    bool thread_is_waiting_to_flush = false;
    unsigned int waiting_for_flush_to_finish = 0;

    void scanEntriesTable();
    void writeEntriesTable();
    unsigned int freeCacheEntry();
    void return_block(BlockCacheEntry& cache_entry);
    void end_operation();
    void op_requested(unsigned int block_id, bool hid);
    void op_released(unsigned int block_id, bool hid, bool dirty);
    void unlocked_flush();
    BufferOperationData& current_operation();

public:
    Buffer(Disk& disk, unsigned int cache_size, bool wipe_mapping_table, bool enforce_operations, bool debug, bool no_hidden);

    unsigned int totalBlocks();
    unsigned int blocksAllocated();
    unsigned int blocksForAspect(bool hidden);
    unsigned int blocksAllocatedForAspect(bool hidden);

    BlockAccessor block(unsigned int block_id, bool hidden);
    BlockAccessor allocateBlock(bool hidden);
    void deallocateBlock(unsigned int block_id, bool hidden);
    void flush();
    BufferOperation operation(unsigned int max_blocks);
    inline bool isDebugging() const { return debug; }
    inline bool hasHidden() const { return !no_hidden; }
    bool allowed(bool hidden, unsigned int allocated, unsigned int changed, unsigned int deallocated);

    friend class BlockAccessor;
    friend class BufferOperation;
};

#endif // BUFFER_HPP
```

buffer.cpp

```
#include "buffer.hpp"
#include "disk.hpp"
#include "utilities.hpp"
#include "consts.hpp"

#include "cryptopp/osrng.h"

#include <iostream>

const unsigned int VIRTUAL_BLOCK = -2;

BlockCacheEntry::BlockCacheEntry() : data(LOGICAL_BLOCK_SIZE, '\xff') {
}

BlockAccessor::BlockAccessor(Buffer& buffer, BlockCacheEntry& cache_entry) : buffer_(buffer), cache_entry(cache_entry) {
    // Lock baton is passed from Buffer::block
}

BlockAccessor::BlockAccessor(BlockAccessor&& other) : buffer_(other.buffer_), cache_entry(other.cache_entry) {
    other.moved = true;
}

BlockAccessor::~BlockAccessor() {
    if (!moved) {
        buffer_.return_block(cache_entry);
    }
}

Buffer& BlockAccessor::buffer() const {
    return buffer_;
}


const secure_string& BlockAccessor::read() const {
    ensure(!moved, "BlockAccessor::read") << "Accessor has been moved";
    return cache_entry.data;
}

secure_string& BlockAccessor::writable() {
    ensure(!moved, "BlockAccessor::writable") << "Accessor has been moved";
    cache_entry.dirtied_by_current = true;
    return cache_entry.data;
}

std::pair<bool, unsigned int> BlockAccessor::block_id() const {
    ensure(!moved, "BlockAccessor::block_id") << "Accessor has been moved";
    return cache_entry.logical_block_id;
}

Buffer::Buffer(Disk& disk, unsigned int cache_size, bool wipe_mapping_table, bool enforce_operations, bool debug, bool no_hidden) :
        disk(disk), cache(cache_size), enforce_operations(enforce_operations), debug(debug), no_hidden(no_hidden) {
    wait_for_flush_done.lock();
    wait_for_ops_done.lock();

    for (auto i = 0u; i < cache.size(); ++i) {
        least_recently_used.push_back(i);
    }

    while (number_of_mapping_blocks * MAPPING_POINTERS_PER_BLOCK < disk.numberOfBlocks() - number_of_mapping_blocks * 2) {
        ++number_of_mapping_blocks;
    }

    if (wipe_mapping_table) {
        secure_string buf(LOGICAL_BLOCK_SIZE, '\xff');
        for (auto i = 0u; i < number_of_mapping_blocks; ++i) {
            disk.writeBlock(i, false, buf);
        }
        for (auto i = 0u; i < number_of_mapping_blocks; ++i) {
            disk.writeBlock(i + number_of_mapping_blocks, true, buf);
        }
    }
```

```cpp
    scanEntriesTable();

    for (auto item : block_mapping) {
        if (item.first.first) {
            max_hidden_id = std::max(max_hidden_id, item.first.second + 1);
            ++hidden_blocks_allocated;
        }
        else {
            max_cover_id = std::max(max_cover_id, item.first.second + 1);
            ++cover_blocks_allocated;
        }
    }
    ensure(hidden_blocks_allocated + virtual_list.size() == cover_blocks_allocated, "Buffer::Buffer")
        << "Number of cover blocks (" << cover_blocks_allocated << ") does not equal number of hidden blocks (" << hidden_blocks_allocated + virtual_list.size() << ")";

    ensure(disk.numberOfBlocks() == block_mapping.size() + unallocated_list.size() + virtual_list.size() + number_of_mapping_blocks * 2, "Buffer::unlocked_flush")
        << "Numbers of types don't add up";
}

void Buffer::scanEntriesTable() {
    std::lock_guard<std::mutex> lg(lock);
    secure_string buf(LOGICAL_BLOCK_SIZE, '\0');
    reverse_block_mapping.resize(totalBlocks());

    // Cover mapping
    for (auto i = 0u; i < number_of_mapping_blocks; ++i) {
        disk.readBlock(i, false, buf);
        for (auto pos = 0u; pos < MAPPING_POINTERS_PER_BLOCK; ++pos) {
            auto phy_blk_id = i * MAPPING_POINTERS_PER_BLOCK + pos;
            auto log_blk_id = intFromBytes(&buf[BLOCK_POINTER_SIZE * pos]);
            if (log_blk_id != NO_BLOCK_ASSIGNED) {
                ensure(phy_blk_id < totalBlocks(), "Buffer::scanEntriesTable") << "Block mapping set for non-existant block";
                if (log_blk_id == VIRTUAL_BLOCK) {
                    reverse_block_mapping[phy_blk_id] = {true, VIRTUAL_BLOCK};
                }
                else {
                    block_mapping[{false, log_blk_id}].physical_block_id = phy_blk_id;
                    reverse_block_mapping[phy_blk_id] = {false, log_blk_id};
                }
            }
            else if (phy_blk_id < totalBlocks()) {
                unallocated_list.push_back(phy_blk_id);
                reverse_block_mapping[phy_blk_id] = {false, NO_BLOCK_ASSIGNED};
            }
        }
    }

    if (!no_hidden) {
        // Hidden mapping
        for (auto i = 0u; i < number_of_mapping_blocks; ++i) {
            disk.readBlock(number_of_mapping_blocks + i, true, buf);
            for (auto pos = 0u; pos < MAPPING_POINTERS_PER_BLOCK; ++pos) {
                auto phy_blk_id = i * MAPPING_POINTERS_PER_BLOCK + pos;
                auto log_blk_id = intFromBytes(&buf[BLOCK_POINTER_SIZE * pos]);
                if (log_blk_id != NO_BLOCK_ASSIGNED) {
                    ensure(phy_blk_id < totalBlocks(), "Buffer::scanEntriesTable") << "Block mapping set for non-existant block";
                    ensure(reverse_block_mapping[phy_blk_id] == std::make_pair(true, VIRTUAL_BLOCK), "Buffer::scanEntriesTable") << "Hidden block not shown in cover block table";
                    block_mapping[{true, log_blk_id}].physical_block_id = phy_blk_id;
                    reverse_block_mapping[phy_blk_id] = {true, log_blk_id};
                }
            }
        }
    }

    for (auto i = 0u; i < totalBlocks(); ++i) {
        if (reverse_block_mapping[i].second == VIRTUAL_BLOCK) {
            virtual_list.push_back(i);
        }
    }
}

unsigned int Buffer::totalBlocks() {
    return disk.numberOfBlocks() - number_of_mapping_blocks * 2;
}

unsigned int Buffer::blocksAllocated() {
    return block_mapping.size();
}

unsigned int Buffer::blocksForAspect(bool /*hidden*/) {
    return totalBlocks() / 2;
}

unsigned int Buffer::blocksAllocatedForAspect(bool hidden) {
    return hidden ? hidden_blocks_allocated : cover_blocks_allocated;
}

BlockAccessor Buffer::block(unsigned int block_id, bool hidden) {
    if (enforce_operations) {
        std::lock_guard<std::mutex> lg(lock);
        op_requested(block_id, hidden);
    }

    unsigned int cache_location;
    auto start_time = std::chrono::high_resolution_clock::now();
    while (true) {
        {
            std::lock_guard<std::mutex> lg(lock);

            auto iter = block_mapping.find({hidden, block_id});
            ensure(iter != block_mapping.end(), "Buffer::block") << "Block " << block_id << "/" << hidden << " does not exist";
            auto& block_info = iter->second;
            if (block_info.cache_location != NO_CACHE_LOC_ASSIGNED) {
                auto& cache_entry = cache[block_info.cache_location];
                if (cache_entry.logical_block_id != std::make_pair(hidden, block_id)) {
                    block_info.cache_location = NO_CACHE_LOC_ASSIGNED;
                }
            }
            if (block_info.cache_location == NO_CACHE_LOC_ASSIGNED) {
                block_info.cache_location = freeCacheEntry();
                auto& cache_entry = cache[block_info.cache_location];
                cache_entry.logical_block_id = {hidden, block_id};
                cache_entry.dirty = false;
                if (block_info.physical_block_id != NO_BLOCK_ASSIGNED) {
                    disk.readBlock(block_info.physical_block_id + number_of_mapping_blocks * 2, hidden, cache_entry.data);
                }
                else {
                    cache_entry.dirtied_by_current = true;
                }
            }
            cache_location = block_info.cache_location;
        }

        auto& cache_entry = cache[cache_location];
        cache_entry.lock.lock();
        {
            std::lock_guard<std::mutex> lg(lock);
            if (cache_entry.logical_block_id == std::make_pair(hidden, block_id)) {
                auto iter = std::find(least_recently_used.begin(), least_recently_used.end(), cache_location);
                if (iter != least_recently_used.end()) {
                    least_recently_used.erase(iter);
                }
                return {*this, cache[cache_location]};
            }
            cache_entry.lock.unlock();
            auto cur_time = std::chrono::high_resolution_clock::now();
            using namespace std::chrono_literals;
            ensure(cur_time - start_time < 10s, "Buffer::block") << "Been waiting for block for > 10s";
        }
    }
}
```

```cpp
void Buffer::return_block(BlockCacheEntry& cache_entry) {
    std::unique_lock<std::mutex> lg(lock);

    auto [hidden, block_id] = cache_entry.logical_block_id;
    auto iter = block_mapping.find(cache_entry.logical_block_id);
    ensure(iter != block_mapping.end(), "Buffer::return_block") << "Block " << block_id << "/" << hidden << " does not exist";
    auto& block_info = iter->second;
    ensure(block_info.cache_location != NO_CACHE_LOC_ASSIGNED, "Buffer::return_block") << "No cache location for block being returned";
    ensure(&cache_entry == &cache[block_info.cache_location], "Buffer::return_block") << "Cache location of returned block is different";
    if (!cache_entry.dirty && !cache_entry.dirtied_by_current) {
        ensure(std::find(least_recently_used.begin(), least_recently_used.end(), block_info.cache_location) == least_recently_used.end(), "Buffer::return_block")
            << "Returned cache entry " << block_info.cache_location << " is already in the LRU cache!";
        least_recently_used.push_back(block_info.cache_location);
    }
    else if (block_info.physical_block_id != NO_BLOCK_ASSIGNED) {
        unallocated_list.push_back(block_info.physical_block_id);
        reverse_block_mapping[block_info.physical_block_id] = {false, NO_BLOCK_ASSIGNED};
        block_info.physical_block_id = NO_BLOCK_ASSIGNED;
    }
    if (enforce_operations) {
        op_released(block_id, hidden, cache_entry.dirtied_by_current);
    }
    if (cache_entry.dirtied_by_current && !cache_entry.dirty) {
        hidden ? ++hidden_blocks_changed : ++cover_blocks_changed;
        ensure(hidden_blocks_changed <= cover_blocks_changed, "Buffer::return_block") << "Too many hidden blocks changed";
    }
    cache_entry.dirty = cache_entry.dirty || cache_entry.dirtied_by_current;
    cache_entry.dirtied_by_current = false;
    cache_entry.lock.unlock();
}

unsigned int Buffer::freeCacheEntry() {
    if (least_recently_used.size()) {
        auto blk_id = least_recently_used.front();
        least_recently_used.pop_front();
        ensure(!enforce_operations || reserved_cache_space >= cache.size() - least_recently_used.size(), "Buffer::freeCacheEntry")
            << "Too much cache space used";
        return blk_id;
    }
    throw std::runtime_error("Cache full");
}

BlockAccessor Buffer::allocateBlock(bool hidden) {
    unsigned int block_id;
    {
        std::lock_guard<std::mutex> lg(lock);

        ensure(cover_blocks_allocated + hidden_blocks_allocated < totalBlocks(), "Buffer::allocateBlock") << "FS is full";
        ensure(hidden_blocks_allocated <= cover_blocks_allocated, "Buffer::allocateBlock") << "Too many hidden blocks allocated";
        block_id = hidden ? max_hidden_id++ : max_cover_id++;
        block_mapping[{hidden, block_id}] = {};
        if (isDebugging()) {
            std::cout << "Allocated " << block_id << "/" << hidden << std::endl;
        }
        hidden ? ++hidden_blocks_allocated : ++cover_blocks_allocated;
    }
    auto acc = block(block_id, hidden);
    acc.writable();
    return acc;
}

void Buffer::deallocateBlock(unsigned int block_id, bool hidden) {
    std::lock_guard<std::mutex> lg(lock);

    auto iter = block_mapping.find({hidden, block_id});
    ensure(iter != block_mapping.end(), "Buffer::deallocateBlock") << "Block " << block_id << "/" << hidden << " does not exist";
    auto& block_info = iter->second;

    if (block_info.cache_location != NO_CACHE_LOC_ASSIGNED) {
        auto& cache_entry = cache[block_info.cache_location];
        if (cache_entry.logical_block_id == std::make_pair(hidden, block_id)) {
            std::lock_guard<std::mutex> lg2(cache_entry.lock);
            ensure(cache_entry.logical_block_id == std::make_pair(hidden, block_id), "Buffer::deallocateBlock") << "Cache entry is not correct";
            cache_entry.logical_block_id = {false, NO_BLOCK_ASSIGNED};
            if (cache_entry.dirty) {
                least_recently_used.push_back(block_info.cache_location);
                hidden ? --hidden_blocks_changed : --cover_blocks_changed;
                ensure(hidden_blocks_changed <= cover_blocks_changed, "Buffer::deallocateBlock") << "Too many hidden blocks changed";
            }
            cache_entry.dirty = false;
        }
        block_info.cache_location = NO_CACHE_LOC_ASSIGNED;
        if (block_info.physical_block_id != NO_BLOCK_ASSIGNED) {
            unallocated_list.push_back(block_info.physical_block_id);
            reverse_block_mapping[block_info.physical_block_id] = {false, NO_BLOCK_ASSIGNED};
        }
    }
    block_mapping.erase({hidden, block_id});
    hidden ? --hidden_blocks_allocated : --cover_blocks_allocated;
    ensure(hidden_blocks_allocated <= cover_blocks_allocated, "Buffer::deallocateBlock") << "Too many hidden blocks deallocated";
    if (isDebugging()) {
        std::cout << "Deallocated " << block_id << "/" << hidden << std::endl;
    }
}

void Buffer::flush() {
    std::lock_guard<std::mutex> lg(lock);
    unlocked_flush();
}

void Buffer::unlocked_flush() {
    std::vector<std::pair<char, unsigned int>> to_flush;
    CryptoPP::AutoSeededRandomPool rng;
    unsigned int num_cover = 0, num_hidden = 0;

    for (auto i = 0u; i < cache.size(); ++i) {
        auto& cache_entry = cache[i];
        if (cache_entry.logical_block_id.second != NO_BLOCK_ASSIGNED && cache_entry.dirty) {
            cache_entry.logical_block_id.first ? ++num_hidden : ++num_cover;
            to_flush.push_back({'C', i});
        }
    }

    ensure(hidden_blocks_changed + cover_blocks_changed == to_flush.size(), "Buffer::unlocked_flush")
        << "Changed stats do not match";

    ensure(hidden_blocks_allocated + cover_blocks_allocated == block_mapping.size(), "Buffer::unlocked_flush")
        << "Changed stats do not match";

    ensure(disk.numberOfBlocks() == block_mapping.size() - to_flush.size() + unallocated_list.size() + virtual_list.size() + number_of_mapping_blocks * 2, "Buffer::unlocked_flush")
        << "Flush sizes don't add up: "
        << disk.numberOfBlocks() << " total blocks, "
        << block_mapping.size() << " allocated blocks, "
        << to_flush.size() << " changed blocks, "
        << unallocated_list.size() << " unallocated blocks, "
        << virtual_list.size() << " virtual blocks and "
        << number_of_mapping_blocks * 2 << " mapping blocks";

    while (hidden_blocks_allocated + virtual_list.size() < cover_blocks_allocated) {
        virtual_list.push_back(NO_BLOCK_ASSIGNED);
        to_flush.push_back({'V', virtual_list.size() - 1});
        ++num_hidden;
    }

    while (hidden_blocks_allocated + virtual_list.size() > cover_blocks_allocated) {
        auto idx = CryptoPP::Integer(rng, 0, virtual_list.size() - 1).ConvertToLong();
        auto phy_blk_id = virtual_list[idx];
        if (reverse_block_mapping[phy_blk_id].second != NO_BLOCK_ASSIGNED) {
            unallocated_list.push_back(phy_blk_id);
            reverse_block_mapping[phy_blk_id] = {false, NO_BLOCK_ASSIGNED};
        }
        virtual_list.erase(virtual_list.begin() + idx);
    }

    auto virtual_idx = 0u;
    while (num_hidden < num_cover) {
        if (virtual_list[virtual_idx] == NO_BLOCK_ASSIGNED) {
            break;
        }
        unallocated_list.push_back(virtual_list[virtual_idx]);
        reverse_block_mapping[virtual_list[virtual_idx]] = {false, NO_BLOCK_ASSIGNED};
        virtual_list[virtual_idx] = NO_BLOCK_ASSIGNED;
        to_flush.push_back({'V', virtual_idx});
        ++num_hidden;
        ++virtual_idx;
    }

    for (auto bm_iter = block_mapping.begin(); num_hidden < num_cover && bm_iter != block_mapping.end(); ++bm_iter) {
        if (bm_iter->first.first && bm_iter->second.physical_block_id != NO_BLOCK_ASSIGNED) {
            unallocated_list.push_back(bm_iter->second.physical_block_id);
            reverse_block_mapping[bm_iter->second.physical_block_id] = {false, NO_BLOCK_ASSIGNED};
            to_flush.push_back({'H', bm_iter->first.second});
            ++num_hidden;
        }
    }
    ensure(num_hidden == num_cover, "Buffer::unlocked_flush") << "Could not generate enough changed";

    secure_string buf(LOGICAL_BLOCK_SIZE, '\0');
    while (to_flush.size()) {
        auto idx = CryptoPP::Integer(rng, 0, to_flush.size() - 1).ConvertToLong();
        unsigned int rand = CryptoPP::Integer(rng, 0, unallocated_list.size() - 1).ConvertToLong();
        auto [mode, cache_idx] = to_flush[idx];
        auto phy_block_id = unallocated_list[rand];

        if (mode == 'C') {
            auto& cache_entry = cache[cache_idx];
            auto& block_info = block_mapping[cache_entry.logical_block_id];

            ensure(block_info.cache_location == cache_idx, "Buffer::unlocked_flush") << "Block info cache location is wrong";
            disk.writeBlock(phy_block_id + number_of_mapping_blocks * 2, cache_entry.logical_block_id.first, cache_entry.data);
            reverse_block_mapping[phy_block_id] = cache_entry.logical_block_id;
            cache_entry.dirty = false;
            block_info.physical_block_id = phy_block_id;

            ensure(std::find(least_recently_used.begin(), least_recently_used.end(), cache_idx) == least_recently_used.end(), "Buffer::unlocked_flush")
                << "Returned cache entry " << block_info.cache_location << " is already in the LRU cache!";
            least_recently_used.push_back(cache_idx);
        }
        else if (mode == 'V') {
            rng.GenerateBlock(&buf[0], LOGICAL_BLOCK_SIZE);
            disk.writeBlock(phy_block_id + number_of_mapping_blocks * 2, true, buf);
            reverse_block_mapping[phy_block_id] = {true, VIRTUAL_BLOCK};
            virtual_list[cache_idx] = phy_block_id;
        }
        else if (mode == 'H') {
            auto location = std::make_pair(true, cache_idx);
            auto& block_info = block_mapping[location];
            disk.readBlock(block_info.physical_block_id + number_of_mapping_blocks * 2, true, buf);
            disk.writeBlock(phy_block_id + number_of_mapping_blocks * 2, true, buf);
            reverse_block_mapping[phy_block_id] = {true, cache_idx};
            block_info.physical_block_id = phy_block_id;
        }

        to_flush.erase(to_flush.begin() + idx);
        unallocated_list.erase(unallocated_list.begin() + rand);
    }

    writeEntriesTable();

    cover_blocks_changed = hidden_blocks_changed = 0;
}

void Buffer::writeEntriesTable() {
    secure_string buf(LOGICAL_BLOCK_SIZE, '\0');

    // Cover mapping
    for (auto i = 0u; i < number_of_mapping_blocks; ++i) {
        buf.replace(0, LOGICAL_BLOCK_SIZE, LOGICAL_BLOCK_SIZE, '\xff');
        for (auto pos = 0u; pos < MAPPING_POINTERS_PER_BLOCK; ++pos) {
            auto phy_blk_id = i * MAPPING_POINTERS_PER_BLOCK + pos;
            if (phy_blk_id >= totalBlocks()) {
                break;
            }
            if (reverse_block_mapping[phy_blk_id].first) {
                intToBytes(&buf[BLOCK_POINTER_SIZE * pos], VIRTUAL_BLOCK);
            }
            else {
                intToBytes(&buf[BLOCK_POINTER_SIZE * pos], reverse_block_mapping[phy_blk_id].second);
            }
        }
        disk.writeBlock(i, false, buf);
    }

    // Hidden mapping
    for (auto i = 0u; i < number_of_mapping_blocks; ++i) {
        buf.replace(0, LOGICAL_BLOCK_SIZE, LOGICAL_BLOCK_SIZE, '\xff');
        for (auto pos = 0u; pos < MAPPING_POINTERS_PER_BLOCK; ++pos) {
            auto phy_blk_id = i * MAPPING_POINTERS_PER_BLOCK + pos;
            if (phy_blk_id >= totalBlocks()) {
                break;
            }
            if (reverse_block_mapping[phy_blk_id].first && reverse_block_mapping[phy_blk_id].second != VIRTUAL_BLOCK) {
                intToBytes(&buf[BLOCK_POINTER_SIZE * pos], reverse_block_mapping[phy_blk_id].second);
            }
        }
        disk.writeBlock(number_of_mapping_blocks + i, true, buf);
    }
}

BufferOperation Buffer::operation(unsigned int max_blocks) {
    auto id = std::this_thread::get_id();

    lock.lock();
    while (true) {
        if (reserved_cache_space + max_blocks <= cache.size() && !thread_is_waiting_to_flush) {
            reserved_cache_space += max_blocks;
            operation_for_thread[id] = {max_blocks};
            if (isDebugging()) {
                std::cout << "OPERATION begin by " << id << " requesting " << max_blocks << " blocks of cache space; " << operation_for_thread.size() << " operations ongoing" << std::endl;
            }
            lock.unlock();
            return {*this};
        }
        else if (thread_is_waiting_to_flush) {
            // wait for flush to end
            ++waiting_for_flush_to_finish;
            lock.unlock();
            wait_for_flush_done.lock();
            // Baton passing style: mutex passed to us.
            --waiting_for_flush_to_finish;
            if (waiting_for_flush_to_finish) {
                wait_for_flush_done.unlock();
            }
        }
        else {
            thread_is_waiting_to_flush = true;
            if (operation_for_thread.size()) {
                lock.unlock();
                wait_for_ops_done.lock();
                // Baton passing style: mutex passed to us.
            }
            unlocked_flush();
            thread_is_waiting_to_flush = false;
            reserved_cache_space = 0;
            if (waiting_for_flush_to_finish) {
                wait_for_flush_done.unlock();
            }
        }
    }
}

void Buffer::end_operation() {
    std::lock_guard<std::mutex> lg(lock);
    auto id = std::this_thread::get_id();
    auto& data = operation_for_thread[id];
    reserved_cache_space -= data.max_blocks - data.blocks.size();
    if (isDebugging()) {
        std::cout << "OPERATION ended by " << id << "; max usage " << data.max_cache_takeup << " (max predicted " << data.max_blocks << "), "
            << data.block_requests << " requests, "
            << data.block_writes << " writes." << std::endl;

        std::cout << "Current state: "
            << "Hidden Alloc = " << hidden_blocks_allocated << ", Cover Alloc = " << cover_blocks_allocated
            << ", Hidden Changed = " << hidden_blocks_changed << ", Cover Changed = " << cover_blocks_changed << std::endl;
    }
    operation_for_thread.erase(id);
    if (thread_is_waiting_to_flush && !operation_for_thread.size()) {
        wait_for_ops_done.unlock();
    }
}

BufferOperationData& Buffer::current_operation() {
    auto id = std::this_thread::get_id();
    auto iter = operation_for_thread.find(id);
    ensure(iter != operation_for_thread.end(), "Buffer::current_operation") << "No operation for current thread " << id;
    return iter->second;
}

BufferOperation::BufferOperation(Buffer& buf) :
        buffer(buf) {
}

BufferOperation::~BufferOperation() {
    buffer.end_operation();
}

void Buffer::op_requested(unsigned int block_id, bool hid) {
    auto& data = current_operation();
    if (data.hidden != 2) {
        ensure(data.hidden == hid, "BufferOperation::requested") << "Requested block is in a different aspect from the previous one";
    }
    data.hidden = hid;
    data.blocks.insert(block_id);
    ++data.block_requests;
    data.max_cache_takeup = std::max(data.max_cache_takeup, static_cast<unsigned int>(data.blocks.size()));
    ensure(data.blocks.size() <= data.max_blocks, "BufferOperation::requested") << "Too many blocks requested";
}

void Buffer::op_released(unsigned int block_id, bool hid, bool dirty) {
    auto& data = current_operation();
    ensure(data.hidden == hid, "BufferOperation::released") << "Released block wasn't requested (hidden deviation)";
    ensure(data.blocks.count(block_id), "BufferOperation::released") << "Released block wasn't requested (block_id deviation)";
    if (!dirty) {
        data.blocks.erase(block_id);
    }
    else {
        ++data.block_writes;
    }
}

bool Buffer::allowed(bool hidden, unsigned int allocated, unsigned int changed, unsigned int deallocated) {
    long after_cover_blocks_allocated = cover_blocks_allocated,
         after_hidden_blocks_allocated = hidden_blocks_allocated,
         after_cover_blocks_changed = cover_blocks_changed,
         after_hidden_blocks_changed = hidden_blocks_changed;

    if (hidden) {
        after_hidden_blocks_allocated += allocated - static_cast<long>(deallocated);
        after_hidden_blocks_changed += allocated + changed - static_cast<long>(deallocated);
    }
    else {
        after_cover_blocks_allocated += allocated - static_cast<long>(deallocated);
        after_cover_blocks_changed += allocated + changed - static_cast<long>(deallocated);
    }
    after_cover_blocks_allocated = std::max(after_cover_blocks_allocated, 0l);
    after_hidden_blocks_allocated = std::max(after_hidden_blocks_allocated, 0l);
    after_cover_blocks_changed = std::max(after_cover_blocks_changed, 0l);
    after_hidden_blocks_changed = std::max(after_hidden_blocks_changed, 0l);

    if (isDebugging()) {
        std::cout << "Evaluating operation: "
            << "Hidden Alloc = " << after_hidden_blocks_allocated << ", Cover Alloc = " << after_cover_blocks_allocated
            << ", Hidden Changed = " << after_hidden_blocks_changed << ", Cover Changed = " << after_cover_blocks_changed << std::endl;
    }

    return after_hidden_blocks_allocated <= after_cover_blocks_allocated && after_hidden_blocks_changed <= after_cover_blocks_changed;
}
```

dir.hpp

```
#ifndef DIR_HPP
#define DIR_HPP

#include "buffer.hpp"

class Dir;

class DirIterator {
    const Dir& dir;
    std::vector<BlockAccessor> accessors;
    std::vector<unsigned int> positions;
    bool hidden;

public:
    DirIterator(const Dir& dir, std::vector<BlockAccessor>&& accessors, std::vector<unsigned int>&& positions, bool hidden);
    std::pair<secure_string, unsigned int> operator*() const;
    DirIterator& operator++();
    void operator++(int) { ++*this; }
    bool operator==(const DirIterator& other) const;
    bool operator!=(const DirIterator& other) const { return !(*this == other); }
};

class Dir {
    Buffer& buffer;
    BlockAccessor header_acc;

    unsigned int height() const;
    unsigned int blocks() const;

    bool node_add(const secure_string& fname, unsigned int value, unsigned int blck_id, unsigned int height);
    std::pair<unsigned int, bool> node_remove(const secure_string& fname, unsigned int blck_id, unsigned int height);
    std::tuple<unsigned int, secure_string, unsigned int> split_node(unsigned int blck_id, unsigned int height);
    void refill_node(BlockAccessor& top_acc, unsigned int uf_pos, unsigned int uf_byte_pos, unsigned int height, unsigned int top_offset, unsigned int top_size);
    bool pop_left_largest_and_replace_rec(BlockAccessor& orig_acc, unsigned int orig_byte_pos,
                                          BlockAccessor& acc, unsigned int byte_pos, unsigned int height);
    bool pop_left_largest_and_replace(BlockAccessor& acc, unsigned int byte_pos, unsigned int height);
    void split_root();
    void unsplit_root();
    void merge_nodes();

public:
    Dir(Buffer& buf, BlockAccessor&& header_acc);
    Dir(Buffer& buf, unsigned int block_id, bool hidden);
    Dir(Dir&& other);

    std::pair<bool, unsigned int> block_id() const;

    unsigned int size() const;

    void add(const secure_string& fname, unsigned int value);
    unsigned int remove(const secure_string& fname);
    void debug();

    DirIterator find(const secure_string& start) const;
    DirIterator begin() const;
    DirIterator end() const;

    static Dir newDir(Buffer& buffer, bool hidden);

    friend class DirIterator;
};

#endif // DIR_HPP
```

file.hpp

```
#ifndef FILE_HPP
#define FILE_HPP

#include "blockfile.hpp"

class Buffer;

class File {
    BlockFile bf;

    File(BlockFile&& bf);

public:
    File(Buffer& buf, BlockAccessor&& acc);
    File(Buffer& buf, unsigned int block_id, bool hidden);
    File(File&& other);

    std::pair<bool, unsigned int> block_id() const;

    unsigned int size(); // XXX const this
    unsigned int numberOfBlocks() const;
    unsigned int blocksForSize(unsigned int size) const;

    unsigned int read(unsigned int pos, unsigned int n, unsigned char* buf);
    unsigned int write(unsigned int pos, unsigned int n, const unsigned char* buf);
    void truncate(unsigned int pos);

    static File newFile(Buffer& buffer, bool hidden);
};

#endif // FILE_HPP
```