JOHANNES KEPLER
UNIVERSITY LINZ

Author
Martin Hengstberger, BSc

Submitted at
Institute of Networks and Security

Supervisor
Univ.-Prof. DI Dr. René Mayrhofer

July 2016

STEGANOGRAPHY IN FILE SYSTEMS FOR MOBILE ENVIRONMENTS WITH PLAUSIBLE DENIABILITY

Master's Thesis
to confer the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

Danksagung

Als erstes, ein wirklich großes Danke an meine Familie, meine Onkel und ganz besonders an meine Eltern für die großartige und andauernde Unterstützung, die es mir ermöglicht hat diese Ausbildung abzuschließen. Diese ist alles andere als selbstverständlich und ein Vorbild für mich um Sie an die nächste Generation weiter zu geben.
Mein Dank und meine Bewunderung gilt meinem Betreuer Prof. Mayrhofer, seiner außergewöhnlichen fachlichen und menschlichen Kompetenz, mit der er mich in vielen persönlichen Besprechungen bereichert hat.
Dank auch an meine drei engsten Kollegen, mit denen ich das ganze Studium gemeinsam meistern dürfte. Es war eine lehrreiche und lustige Zeit, auch oft auf engsten Raum - ein wirklich tolles Team.
Danke an die JKU die mir viel gegeben hat, nicht nur wertvolles Wissen und (Auslands-)Erfahrung, außerdem lernte ich dort eine ganz besondere Frau kennen, meine Freundin. Danke für deinen Beistand.

ABSTRACT

Conventional encryption entirely relies on the secrecy of a password (or secret key), which is not always good enough. Today, passwords can be guessed, stolen or the users are forced to reveal them, which then renders encryption ineffective.
Encryption with additional file hiding is secure and deniable. This thesis proposes a new file system that allows the use to reveal passwords and still retain confidentiality of high importance data, only surrendering predefined low importance data.
This file system improves file system security against forced password revealing, technical checks (e.g. string/magic number search), and is particularly beneficial for the user in court cases due to Plausible Deniability. This new file system hides and encrypts data in file slacks.

ZUSAMMENFASSUNG

Konventionelle Verschlüsselung verlässt sich völlig auf ein Passwort oder einen geheimen Schlüssel, was nicht immer gut genug ist. Passwörter können erraten, gestohlen oder die Herrausgabe erzwungen werden etc. Dadurch wird jede Verschlüsselung zwecklos.
Verschlüsselung zusätzlich kombiniert mit Verstecken von Dateien ist sicher(er) und ableugbar. Diese Masterarbeit schlägt ein neues Dateisystem vor, welches der NutzerIn erlaubt Passwörter heraus zu geben. Trotzdem bleibt die Vertraulichkeit von wichtigen Daten gewahrt . Es werden unwichtige vordefiniert, die dann herrausgegeben werden können.
Dieses Dateisystem verbessert die Dateisystemsicherheit gegen erzwungene Passwortherausgabe und technische Überprüfungen z.B.: Suche nach Zeichenketten oder "magischen Zahlen". Das ist besonders vorteilhaft für die NutzerIn bei gerichtlichen Verhandlungen. Dieses neue Dateisystem versteckt und verschlüsselt Daten in "Datei-Slacks".

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

Part I

ABOUT THIS AND RELATED WORK

1 INTRODUCTION

This thesis is about a program to hide files on a mobile device e.g. an Android smart phone. Files are stored in blocks. Some files do not need a whole block. That leaves unused space (file slack) to hide data. The hidden files will be stored in file slacks. This choice is explained in Chapter 2. Using password protection helps to limit access to personal or sensitive data, but if the password is compromised, this protection is lost. The password quality is important and at the same time often low. Humans are generally the weakest link in this regard. It is possible to memorize a complicated password with 15 characters or more, but it is not convenient. Password leaks like the one at a popular carrier networking site `Linkedin.com` prove that the most popular passwords are simple to guess e.g. "123456", "password", "111111" or "qwerty" [22].
A user could be forced by judge in a legal proceeding to reveal a password or a user could be blackmailed. If a journalist wants to pass a country border of a war zone he could forced by soldier to reveal passwords with a threat of violence. Relying only on the secrecy of a password in insufficient in these cases, but there are ways for improvement and this is what this thesis exemplifies. In software security it is very (probably most) economic to secure the weakest link first. So additional protection for these cases is desirable. This can be achieved with complementary steganography and Plausible Deniability.
There are countless programs that hide information in pictures or audio files. Hiding files in a file system works at a larger scale and can offer more space and less restrictions. Picture steganography needs picture file formats and audio steganography needs audio file formats. In file systems there are no restrictions on file types as opposed to the mass of picture and audio steganography methods. Any file type can be used in file system steganography. However, it can be more complicated too. Hiding large files might needs spliting into smaller part that have to be reconstructed again. Furthermore there is a danger that hidden data is overwritten by accident. This problem needs to be mitigated.
Mobile devices are widely available and are typically always carried along with the owner. It is unsuspicious to carry e.g. a smart phone in and out of many places. So it is simple and common to physically transport data. During traveling the devices might be inspected e.g. at an airport with sensitive data on the device. The owner might want to protect the data on his mobile devices, particularly in repressive or instable regimes this can be important.
Plausible Deniability is sometimes a desirable quality when important sensitive data is stored and active or passive involvement can later be believably denied. With this Plausible Deniability characteristic it is technically very hard, in the best case impossible to prove the

possession of sensitive data. It can be a significant advantage if one can state to have no knowledge of something and the opposing party can not disprove this claim. Additionally if sensitive data is hidden, then an inspector has to find it first. There is a chance that the data will be overlooked to begin with. The need of searching for that data raises the afford an inspector has to make. An inspector will not have infinite time and resources to continue searching for hidden data. So he might stop looking before he has found the data. Steganography and Plausible Deniability complement each other well.

1.1 STRUCTURE OF THIS WORK

The first chapter gives a context to the problem of password leakage and the objective of this thesis. Furthermore several approaches and existing tools are presented. Then two prominent file system drivers (Ext4 and FUSE) are reviewed that ultimately had to be rejected. The goals of this thesis can not be reached with them.
The second chapter evaluates various methods of hiding data in file systems and chooses one data location that satisfies a set of criteria best. This data location is the file slack.
The third chapter discovers issues and potential of the chosen data location, especially the available space in file slacks.
The fourth chapter contains important implementation details, a design rationale of the created proof of concept implementation, a description of the used encryption and the internal processes of the new file system. Finally, the security of the file system is analyzed and the android user interface is pictured. This chapter holds the essence of this master thesis.
The last chapter five includes future work, a summary and the conclusion of the created material.

1.2 PROBLEM DEFINITION AND OBJECTIVE

A user stores files and wants these files to be private. The should stay private even if the device on which the files are stored is checked by an inspector.
An inspector would be a person that has physical control of the device and performs automated and manual tests trying to find what ever information the device contains. Furthermore such an inspector can force the user to surrender passwords with e.g. court order, blackmail, violence etc.
This is a seemingly hopeless situation to keep the private files private.
The objective of this thesis is to create a file system that will allow the user reveal passwords and still guarantee the confidentiality and integrity of high importance data, while some mildly compromising data is made available with those passwords. So the existence of a special file system made for hiding data is plausibly justified. Further-

more the file system should not allow to circumvent the password protection. In addition the private file should be hidden. So finally a file system hides files and guarantees plausible deniability for very important data. Non-technical measures are out of scope. The file system should both hide the existence of the private files as well as protect its security. Security is represented by Confidentiality, Integrity and Availability. This proof-of-concept file system shall minimize the traces of the private files. The scope is only on the file system itself, not on leakage of other components e.g. application leakage.

Providing Availability is hard. An inspector with physical control can always delete all data or even destroy the device.

1.3 USE CASE

Imagine a journalist wants to carry delicate pictures or documents across country borders or war zones. Then it would be important that the existence of the data is hidden additionally to making it impossible to access for an inspector.
Another case could be a traveling business person, who could take data e.g. construction plans through mandatory security checks. If the device's data is copied using the on board file system drivers, the file slack will not be copied. This is a major advantage.
Some repressive regimes may impose regular automated security scans on citizens or regime members. Steganography can help to disguise data and pass automated tests. Even if the steganography file system is found, the user can give away one or two passwords that will explain the existence of the file system. Those compartments hold only mildly compromising data e.g. some picture showing naked persons. The important sensitive data's password(s) are not surrendered and remain private e.g. trade secrets, bank account credentials. Through the Plausible Deniability characteristic the number of used compartments (virtual drives) can not be determined. So nobody could prove if one, two, three, all or none compartments were used. The user is not able to change the predefined number of compartments. This is important for the construction of the Plausible Deniability characteristic. A simple use case including a test output are shown in Section A.1.

1.4 APPROACH

This section introduces relevant definitions and the subsections then describe known approaches to steganography file systems.

Definition 1: Steganography is the art of concealing an information (file, message, image or video). The term derives from the Greek words *steganos* meaning "covered, concealed or protected" and *graphein* meaning "writing". [51]

Definition 2: "Cryptography is the practice and study of secure communication in the presence of third parties called adversaries"[49]

This definition makes clear that cryptography makes an apparent message unreadable for an adversary, while the two communicating parties can still understand the message. Steganography hides the mere existence of a message. This thesis will combine both characteristics.

Definition 3: "Plausible Deniability is the ability for persons to deny knowledge of or responsibility for any damnable actions committed by others because of a lack of evidence that can confirm their participation, even if they were personally involved in or at least willfully ignorant of the actions."[50]

In this case Plausible Deniability refers to a security property of technical systems that allows believably denying to own an information e.g. when confronted with a judge or an inspector. The user can not be caught with technical measures. In theory the solution is the *one time pad*. It offers perfect encryption with plausible deniability [5]. However, its high requirements discourage its use. Unfortunately the one time pad is impractical to use for large data amounts, because a random key has to have at least the length of the plaintext and can never be reused. So this is not a good approach for data storage.

The following paragraphs discuss more practical approaches. The first known designs of an encrypted file store with built in Plausible Deniability were described by Anderson, Needleham and Shamir [2] under the name *steganographic file system*.
The goal of such systems is to provide protection against the risk of *being forced to reveal encryption keys*. The user can believably deny that further unrevealed data exists on the storage medium. Steganography file systems feature a high degree of security against forced disclosure of their content. The legitimate user who knows the password for a file is able to access it. The attacker without this knowledge can neither access any file nor even gain information on whether a file exists, even with full physical access.

There are two basic concepts introduced by Anderson, Needleham and Shamir [2]. One assumes that the attacker does not have any knowledge about the plain text and therefore calculation intense encryption is substituted with linear algebra operations. The system uses a set of cover files which contain initially random data. The data is stored by modifying several cover files. The hidden file can be retrieved as linear a combination (XOR) of the cover files. They must be large enough to guarantees that a brute force attack remains computationally infeasible. Large cover files have a large search space so an attack would have only a tiny chance of recovering the data by just trying all possible combinations to read the data. This approach is rather vulnerable since no proper encryption is in place and the non-random plain text can be (partly) known to the attacker.
In the second concept presented by Anderson, Needleham and Shamir [2], it is suggested to initially fill the entire file system with random

data and then write the encrypted file into pseudo random blocks depending on a function of the path, filename and the password. The carrier blocks are disguised within all the random data. With increasing filling degree of the medium collisions will happen and non-random data will be overwritten. According to the birthday paradoxon this will most likely occur for the first time when $\sqrt{n}$ blocks were filled with hidden data, where n is the number of non-random data blocks. Naturally the more used blocks the more likely are collisions. Hence only very little of the disk space could actually be used to avoid unpleasant data loss. As mitigation a used data block can be written multiple times on the medium and overwritten blocks would have to be at least identified.

1.5 RELATED WORK

This chapter discusses several steganography tools and file systems and their characteristics. Each section describes what algorithm or mechanism the steganography is based on and on which file system types, e.g. FAT32, EXT4 or file formats, e.g. WAV, JPEG, it works on. Also some known issues or limitations of each method are mentioned. Typically performance, data integrity and data confidentiality are the design goals of the tools and file systems discussed at this point. The systems and tools here represent only a very small portion of what's available.
Of the listed related work almost all tools implement cryptography, only MobiHydra and Truecrypt provide Plausible Deniability. Though, MobiHydra and Truecrypt do not offer steganography. This thesis proposes a new file system that unites all three characteristics: steganography, cryptography and Plausible Deniability. Additionally it uses a completely different experimental data location, that location will be file slacks. The data location (selection) is extensively discussed in Section 2.2.2

1.5.1 *StegFS: Steganographic File System*

StegFS [31] is a deniable steganographic file system that tries to mitigate data loss at a file system level. This approach uses unallocated memory and an additional block allocation bitmap. Each entry is an encrypted 128-Bit entry. Without the proper security levels key, nothing more than the presence of StegFS can be determined. The user could give away some of the lower security level to justify the existence of the file system driver, which is not hidden. StegFS uses its own partition to avoid accidental overwriting by normal user operations. This partition looks like it is unallocated and has been wiped with a deletion tool, holding random data. StegFS implements a removable kernel module that is based on the Ext2 file system. The kernel module is located between the hardware device driver that manages the blocks of the block device and the Virtual File System

(VFS) interface in the Linux kernel that serves system calls e.g. `open()`, `read()`, `write()` which applications use to interact with the OS.
The user could be forced to overwrite the whole partition which would effectively delete all data which is always possible. StegFS contains the standard driver for Ext2 and a slightly altered version to realize its hiding functionality. StegFS has a fixed number of 15 security levels and manages all security contexts. The payload data is encrypted with Advanced Encryption Standard (AES) in Cipher Block Chaining (CBC) mode with a 16 Byte initialization vector.
Users should not move files from a higher security level to a lower one since higher Inode numbers indicate the existence of higher security levels. Then an intermediate range of Inode numbers would not be visible because the according files are stored in a high security level. Also symbolic links, paths, log files or the shell history could give an adversary important hints, which the file system can not mitigate.
StegFS allows to plausibly deny the total number of files on the drive and the confidentiality of hidden files is guaranteed. Deletion of both hidden or non-hidden files automatically initiates a secure deletion with overwrite. Lower security levels can be revealed to any adversary forcing to give away a pass phrase, which reveals mildly compromising data, but justifies the installation of StegFS. Writing while not all security levels are open can lead to overwriting of data from higher security levels, but with a low probability assuming only a moderate amount of writes occurs. Non-hidden files continue to be accessible if the StegFS driver is removed. If the driver is installed again also the hidden files are accessible again. Write performance to hidden layer is slightly worse than to non-hidden layers.
StegFS and DHFS are very similar, but differ in one important point, the data location. StegFS uses unallocated memory to store hidden files. This implies that the user must not allocate all memory on the medium, he can only allocate up to a certain limit otherwise hidden data is lost. DHFS uses file slack to store hidden files. So the advantage of DHFS is that the medium can be allocated up to 100% without loss of data.

1.5.2 *DEFY: A Deniable File System For Flash Memory*

DEFY [36] is designed to work exclusively on solid-state drives, targeted on ones that can be found in mobile devices. These drives have unique properties and this deniable file system design addresses their needs. This includes multiple layers of deniability, encryption and the means to delete data securely. DEFY is implemented for Yet Another Flash File System (YAFFS) with a full disk encryption. The deniability of the file system is based on a set of keys that are needed to access the encrypted disk. Each key reveals just one compartment of files. The user may only give away a few keys that reveal normal data, not keys that reveal sensitive data. An adversary can not determine the total number of keys.

DEFY provides a kernel module that handles wear leveling and attempts to mitigate data loss. Flash memory blocks only sustain a limited number of write procedures. Therefore flash memory blocks that failed can be reassigned to new unused blocks in order to extend the memory's life time. The forward writing nature of log-structured file systems and the way of normal user interactions aid the deniability of DEFY. WhisperYaffs is a similar tool that DEFY compares itself with and proves comparable performance. DEFY hides its sensitive data in unallocated blocks of the memory. There are several security levels, so lower levels are not aware which blocks are allocated by higher security levels. Hence the user is only guaranteed prevention of data loss if all security levels are unlocked and all allocated blocks are known to the kernel module. With no or low security level blocks could be allocated that are used by a higher security level, then data loss can occur with a certain probability. DEFY is designed for Unix-based OSes and requires to load a kernel module running with root permissions as seen in Figure 1. This kernel module lies between the virtual file system interface, which is the file system abstraction the kernel uses and the underlying hardware device driver.

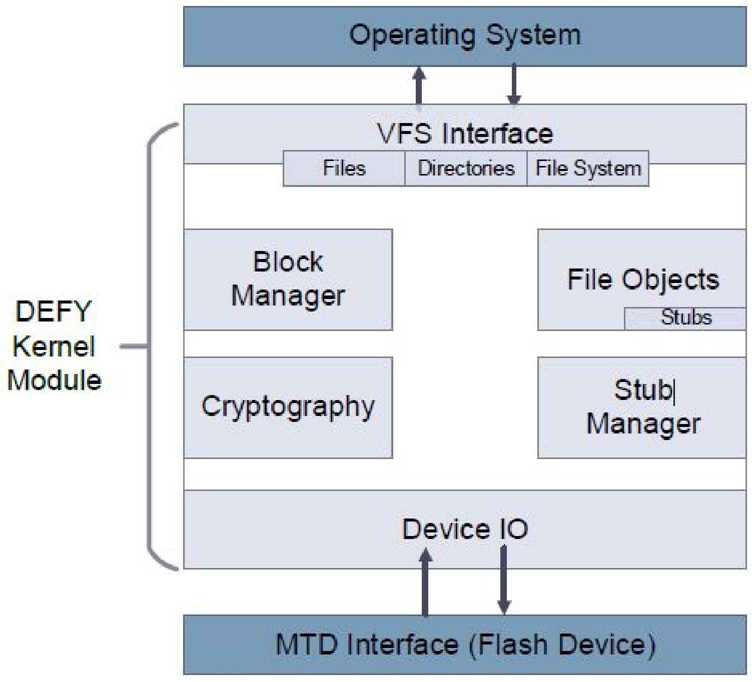


Figure 1: Abstract system overview over the DEFY kernel module and its components. [36]

DEFY has several disadvantages that DHFS has not. DEFY requires flash memory. DHFS works on any block oriented hardware. DHFS allows 100% medium allocation, DEFY does not. DEFY needs to add an kernel module that is write and signature protected. This requires root permissions on android. DHFS is self contained and does not require root permissions.

1.5.3 *MobiHydra: Pragmatic and Multi-level Plausibly Denieable Encryption Storage for Mobile Devices*

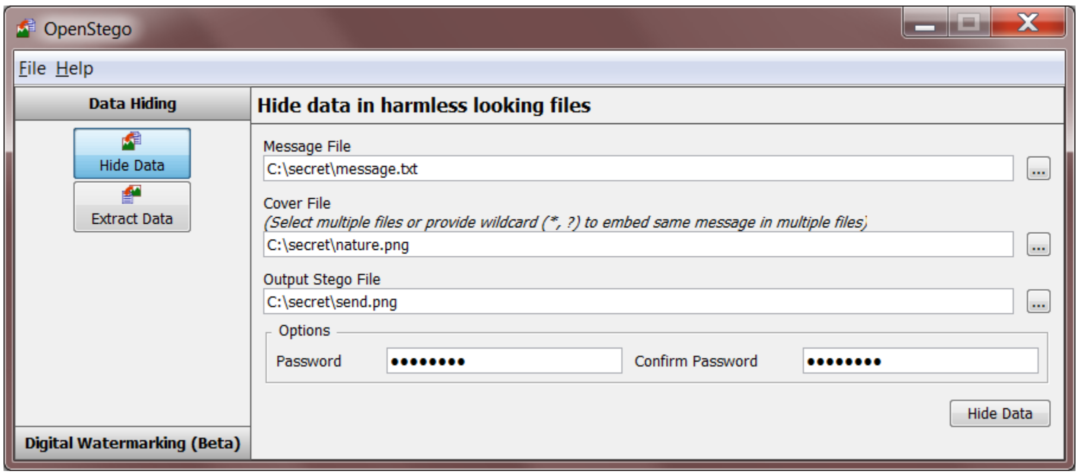
MobiHydra [52] is a solution for securely hiding data in several hidden volumes also providing plausible deniability. It is meant for mobile devices and supports multiple security levels, in this case user-defined at installation time. It offers a plausibly deniable encryption (PDE) mode and a standard mode. The user chooses the mode by entering either the public password or the hidden password at boot time. MobiHydra calculates an offset from the hidden password and tries to mount the volume onto the file system mount point that is otherwise used by the physical storage. The offset calculation is complicated and unique for the current volume of theoretically an arbitrary amount of volumes, including hash functions and a salt component. The hidden volumes are located in the external storage of the device in empty space. The hidden volumes are first filled with random bits to make them indistinguishable from empty (encrypted) space. There is also a special hidden volume "*Shelter*" that is used temporally to transfer data from standard- to PDE-mode. The Shelter volume is wiped securely after each transfer to avoid traces. In order to decrease the chance of an accidental overwrite the hidden volumes and the Shelter volume are located close to the end of the storage. A linear beginning to end allocation strategy is assumed. MobiHydra uses a pair of 1024Bit RSA keys for its volumes and AES-XTS keys for data. The authors[52] include a camera app that stores images directly in the Shelter volume.
This tool requires a separate physical FAT32 storage partition. The user has to configure the equally big volume size so the offset does not need to be stored. It can not be resized and data in the Shelter volume is visible to all deniability levels.
MobiHydra mitigates the risk of a boot-time attack, which is a common vulnerability of other similar PDE tools that rely on hidden volumes. MobiHydra compares itself to Android Full Disk Encryption (FDE) and Mobiflag. Mobiflag was the first attempt to customize Android FDE for plausible deniability. MobiHydra has performance advantages comparing to Mobiflag and have similar features. FDE does not provide deniability.
DHFS does not allow a boot-time attack and does not have the need to allocate 50% of an FAT32 formatted medium. DHFS differs from MobiHydra greatly in the user interaction. Also DHFS does not require reboots. It can switch from hidden to accessible and back at the user's discretion. Therefore DHFS imposes less requirements, while providing similar features.

1.5.4 *OpenStego*

OpenStego [45] is an open source Java steganography application that supports two features. The first is data hiding in images and the second is file water marking in images. It supports compression and encryption of the payload data. The digital watermarking is based on "Dugad's algorithm" and the data hiding is based on a "randomized LSB" algorithm. The payload data is hidden in the error tolerant image data. Each pixel of the image is stored with 3 color channel Red, Green, Blue (RGB). The least significant bit (LSB) of a color channel could be changed according to the payload file, which is how the payload is encoded in the carrier image file. The picture only changes slightly, since the LSB has very little influence on the actual appearance of the final picture. If one channel does not offer enough space for the data all channels could be used. This effective adds some noise to the picture that the viewer will not notice. Depending on the file format some format specific data in the header (if existing) e.g. magic numbers at the beginning or end of the files have to be adequately left unchanged.

In computer science magic numbers are numeric constants. These are often used to identify a file format or generally as distinctive unique values such as an identifier.

This program can read and write a hidden message in an image file. It is platform independent (Java) and was tested on Windows and Linux as claimed by the author. This concept can only work with error tolerant data, but it is built with a modular architecture so other algorithms could be added easily. OpenStego does not work on a file system level, but on a file format level. It offers a command line interface and a graphical user interface as shown in Figure 2.
This application represents one example of a very large pool of similar programs. Hiding data in pictures may be the most popular way of implementing Steganography. OpenStego was released under GNU General Public License v2.0.



Message File
C:\secret\message.txt

Cover File
(Select multiple files or provide wildcard (*, ?) to embed same message in multiple files)
C:\secret\nature.png

Output Stego File
C:\secret\send.png

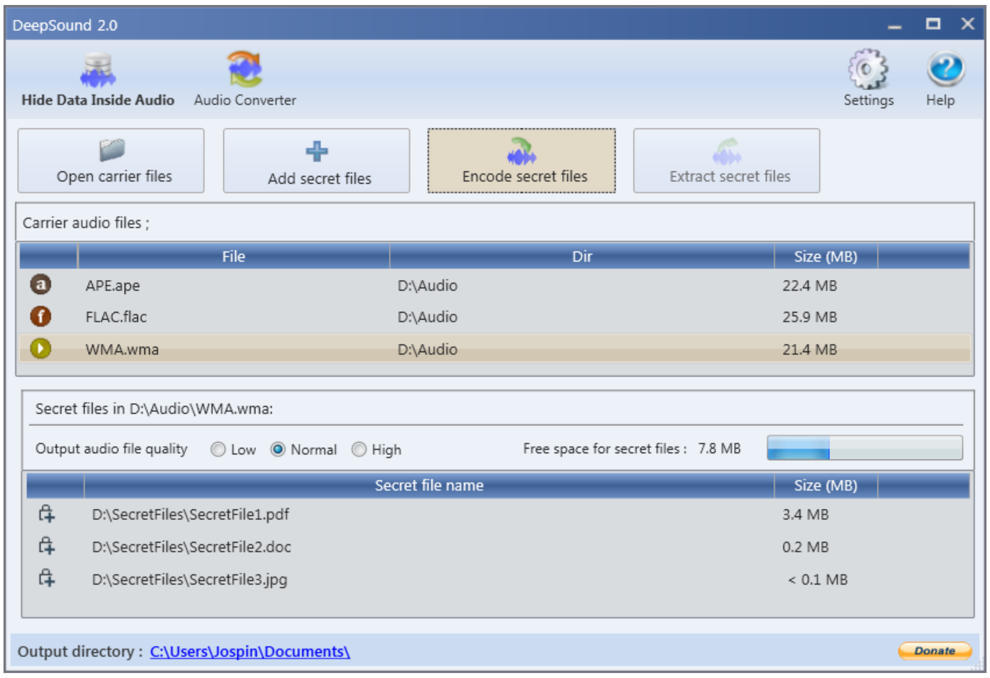
Password ••••••••
Confirm Password ••••••••

Figure 2: OpenSteg graphical user interface for hiding a secret message in a harmless looking image file.[45]

1.5.5 *DeepSound*

DeepSound [44] is a closed source free steganography tool that allows the user to hide secret data in audio files. Of course the secret data can also be extracted again. It supports the file formats FLAC, MP3, WMA, WAV as well as APE and can convert one audio format into another. DeepSound is a Windows only application and requires the

Microsoft .NET Framework 4.0. It allows to hide multiple payload files within the audio carrier. It is recommended to disable volume normalization if the audio file is burned to a CD to avoid data loss. The resulting audio quality can be configured with three levels to influence the maximum payload size. Low quality for a WAV audio file allows to use up to 50% of the carrier file for hidden data, medium quality 25% and high quality 12.5%. DeepSound supports AES-256 encryption to improve data protection and comes with a graphical user interface only as depicted in Figure 3.
There is no source code available for this tool so only the manufacturer knows how the steganography is implemented here. Kaliappan [20] suggests to use a LSB-modification strategy. The encoding of the secret data in the LSB of each audio sample at each discrete point in time in the audio file is a simple solution. It is likely that OpenStego operates this way. Audio files are good steganography carriers because the human auditory system is less sensitive than the visual system. Therefore small errors are less noticeable in audio than in visual context.



Carrier audio files ;

	File	Dir	Size (MB)
	APE.ape	D:\Audio	22.4 MB
	FLAC.flac	D:\Audio	25.9 MB
	WMA.wma	D:\Audio	21.4 MB

Secret files in D:\Audio\WMA.wma:

Output audio file quality Low [✓ Normal] High

Free space for secret files : 7.8 MB

	Secret file name	Size (MB)
	D:\SecretFiles\SecretFile1.pdf	3.4 MB
	D:\SecretFiles\SecretFile2.doc	0.2 MB
	D:\SecretFiles\SecretFile3.jpg	< 0.1 MB

Output directory : C:\Users\Jospin\Documents\

Figure 3: DeepSound graphical user interface for hiding some secret information within an audio file.[44]

1.5.6 *TrueCrypt*

TrueCrypt [14] is a freely available open source on-the-fly encryption solution. It is a discontinued project since May 2014, despite its great success in the security community that made security for the masses its motto. The message "TrueCrypt is not secure" appeared on the manufacturer's website, ending the official product's lifetime. It is assumed that there has not been an incident that really rendered it insecure, but that no further security fixes will be provided from that

time on. An independent audit reviewed it [1] and found four vulnerabilities, although no significant flaws were present.
TrueCrypt can create virtual encrypted volumes in a file, partition or encrypt the entire storage device. TrueCrypt is written in C/C++ and Assembly and is available for Windows, OS X, Linux and Android (LUKS manager) and was released under the "TrueCrypt" License. It has a command line interface (CLI) and a GUI as shown in Figure 4
Plausible deniability is supported by allowing to create hidden volumes within hidden volumes. Still application leakage is an inherent problem with any steganography system. TrueCrypt supports parallelization and has decent performance on standard workstations, but does not support Trusted Platform Modules. Several encryption algorithms and algorithm combinations are supported, including AES, Serpent and Twofish. Also several cryptographic hash functions are available: RIPEMD-160, SHA-512 and Whirlpool. TrueCrypt uses the XTS mode for encryption.

XEX-based tweaked-codebook mode with hertext stealing TS) is a block her mode of ration used for disk encryption. llows random ess.

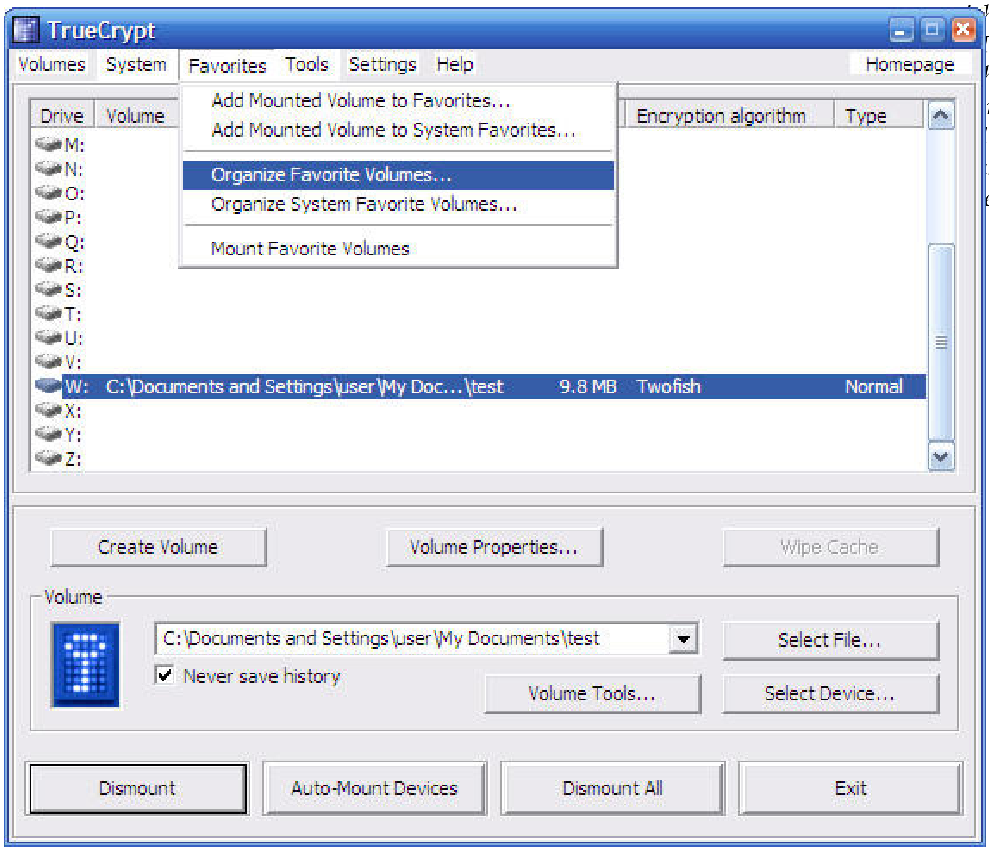


Figure 4: TrueCrypt's Windows GUI client showing several encrypted volumes.[32]

1.5.7 *Bmap*

Bmap [39] is an open source forensic tool that is able to hide, recover and clear data from the file slack. Encryption of the payload data is supported. It was designed as a CLI tool that works on the Ext2 file system, but it's compatible with later Ext file system versions. It

harnesses a powerful *file system independent* mechanism for mapping logical to hardware sectors on a storage device. This tools determines the device sectors that are used by a particular carrier file to then calculate the free slack space by using the device's block size and the carrier's file size. After that it can write an encrypted and secret message in the file slack. This method may work for a larger variety of file systems.
File slacks may be overwritten by other processes or an unaware user, then the secret data is lost. This tool is publicly available, it released without license and is therefore copyrighted by default.

1.5.8 *Comparison DHFS and Related Work*

This section shows the motivation for creating DHFS. DHFS combines advantages of existing systems, while eliminating some restrictions. The mentioned steganographic file systems all use unallocated space as data location which leads to limited utilization. DHFS uses file slacks instead of unallocated space and has therefore no space allocation limit. Table 1

Name	Encryption	Security levels	Steganographic	Deniable	No root required	Medium utilization	Hide-out location
DHFS	**Y**	**Y**	**Y**	**Y**	**Y**	**no limit (100%)**	file slack
StegFS	Y	Y	Y	Y	N	limited	unallocated space
DEFY	Y	Y	Y	Y	N	limited	unallocated space
MobiHydra	Y	Y	Y	Y	N	limited to 50%	unallocated space
Truecrypt	Y	Y	N	Y	N	no limit	visible volumes
OpenStego	Y	N	Y	N	N	no limit	Image files
DeepSound	Y	N	Y	N	Y	no limit	Audio files
Bmap	N	N	Y	N	Y	no limit	file slack

Table 1: Comparison of DHFS and related work. Y = YES, N = NO

1.6 FOURTH EXTENDED FILE SSYTEM (EXT4)

This section investigates commonly used encryption technologies, in particular the Ext4 file system's out-of-the-box cryptography features and its alternatives.
Ext4 could in the end not be used to satisfy the goals of this thesis, but there is potential to do so in future versions.

1.6.1 *Ext4 and its built-in Cryptography Feature*

Linux and also Android are able to use the newly released Ext4 encryption technology. This feature is particularly interesting for the possibility of encrypting the Linux home directories individually. Hence each user can not access another user's data. This was not supported before. The Ext4 encryption is already available for testing, but is still in development. There is a general trend towards encryption in Android.
Many Linux distributions already use Ext4 by default. Well-established alternatives are Ecryptfs [25] and Cryptsetup-LUKS with Dm-Crypt [11]. Ext4 tries to combine the efficiency of Dm-Crypt with the flexibility of Ecryptfs. However, for now Cryptsetup-LUKS with Dm-Crypt remains the first choice for notebook systems. In June 2015 the Linux kernel version 4.1 was released with the first appearance of the Ext4 encryption feature. "Theodore Ts'o" is strongly involved in the development, who introduced the Ext file system family and is responsible for the Ext4 development.[47] The Ext4 encryption was especially developed for a multi-user environment. This includes devices like Android tablets, shared Notebooks as well as many cloud services that are offered in great varieties nowadays, for example Amazon EC, Google Drive, Dropbox etc.
Guillaume [21] summarizes the Ext4 encryption feature, which is based on directories. An encryption policy can be applied to a initially empty directory. File names and file content are separately encrypted. File name encryption needs padding (4,8,16 or 32 Byte). In kernel version 4.1.3 the encryption algorithm and mode are hard coded. For the file name AES-256 with Cipher Block Chaining (CBC) and Cipher Text Stealing with Initialization Vector (IV) of 0 is used. The ciphertext is then encoded in a Base64-like encoding. No integrity checks are available in this version. Later version may contain more cipher modes including Galois/Counter Mode (GCM) which implements authenticated encryption (Integrity check included). The policy of an directory stores an 8 Byte descriptor referencing a master key from the user keyring in the kernel. The key must be of type `logon`, which is only accessible for the kernel. When a directory is accessed the master key is fetched and a new directory specific key is derived from it as well as a NONCE using AES-128 with Electronic Codebook (aka ECB) mode so the content can be decrypted. In order to be able to use the ext4 encryption feature the kernel needs to be complied with

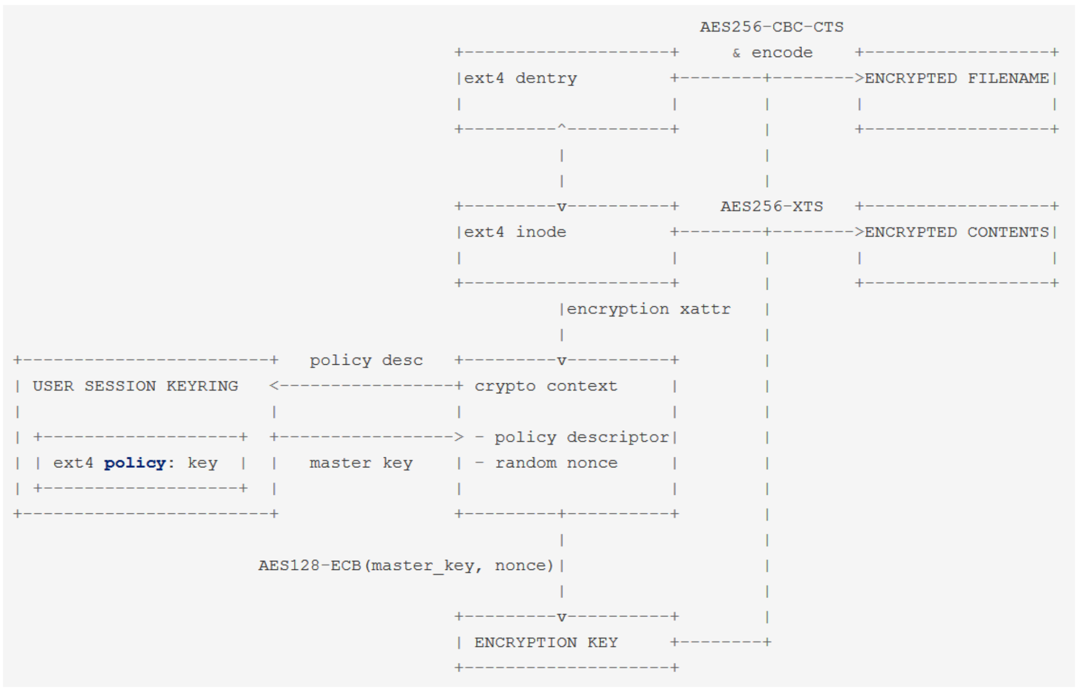


Figure 5: Overview of the Ext4 encryption process [21]

the flag `CONFIG_EXT4_ENCRYPTION`. The command line tool `e4crypt` allows the user to activate and configure the ext4 encryption. This feature is in kernel version 4.1.3 not activated by default and is not deemed stable by the author Theodore Ts'o. Development is going own. However, there is no indication that Steganography or Plausible Deniability will be integrated into Ext4. This is the most discriminating difference to DHFS which implements both characteristics.
Linux offers several encryption options with installer support, among those are Truecrypt (see Section 1.5.6) and Cryptsetup-LUKS with Dm-Crypt. Yet they are developed for a single user use case and encrypt everything that is written to the disk (Full Disk Encryption). In order to make it possible to separate users, each user would need to have their own partition, which imposes size limits and an additional organizational afford. Furthermore it is necessary for each user to use his own encryption key.
Ext4 avoids these problems by implementing encryption on a file basis. Each file is encrypted independently in contrary to the full disk encryption. This offers significantly more flexibility. Ext4 can contain encrypted and unencrypted files. Different files can be encrypted with different encryption keys, thus allowing encryption systems with multiple users with different keys. They can coexist on the same physical device, even the same partition.
The well known Ubuntu Linux distribution already implements a home directory encryption in a multi-users environment with Ecryptfs. Ecryptfs works in a transparent layer on top of any compatible file system, which could be e.g. Ext4, Ext3, NFS, etc. Ecryptfs takes the payload file, encrypts the content and creates an encrypted file name in the underlying file system like Ext4. The layered architecture is resource intense for processor and main memory. Ext4 and Dm-Crypt

encryption techniques perform better.
Encrptfs is badly suited for smart phones, tablets and similar mobile devices due to the relatively low performance and the limited power supply by a battery. Using resource demanding methods reduces the run duration for mobile devices that are powered by battery cells.[29]

1.6.2 *Comparison of Ext4, Dm-Crypt and Ecryptfs*

All three Ext4, Dm-Crypt and Ecryptfs are meant to provide decent security in the case that a mobile device is analyzed by an inspector e.g. at brief security checks at an airport or a thief that stole the device or a person in the same environment that seizes an opportunity to inspect an unwatched device. None of the mentioned cryptography systems provide an integrity check, whether or not the encrypted data is the same as the decrypted data. Therefore the tools do not recognize if the data was reconstructed incorrectly after an en- and decryption cycle. This might very well happen if an attacker modifies the encrypted data. This could also happen if the data carrier has read problems and returns some faulty data. Of course this is a fatal scenario when using encryption. If a single bit flips, the whole file would not be deciphered correctly and appears as random data. The standard solution for this is to introduce an checksum. Such a solution is missing in all three systems.
Ecryptfs and the Ext4 encryption are always recognizable as such. In a court it cannot be denied that the according cryptography system was used at all (no Plausible Deniability).

The new Ext4 file system can encrypt data within the file system as seen in Figure 6. The whole disk encryption approach of Cryptosetup/LUKS and Dm-Crypt resides in between the file system layer and the block layer. Ecryptfs runs on top of other file systems and stores encrypted data using another file system. This occupies processing and memory resources.

Ext4 encryption works on a per-file basis. Only the file content and the file name are encrypted. Metadata other than that is *not* encrypted and visible for any user with the according permissions. The creation, modification and access timestamps may reveal file usage. File permissions and file sizes are likewise visible. Maybe the metadata per se does not reveal any secrets, but might still allow some minor conclusion about the files. Any user with the according file system permissions could change or delete encrypted files. The secrets would not be revealed, but are effectively lost if no backup exists. It depends on the application if that disadvantage can be accepted or not. However, this characteristics could also be leveraged by the Android operating system. Temporary data or the browser cache could then be deleted, if memory is needed. That could be even done if the data is encrypted.

Of course the time stamps can be faked easily with standard tools.

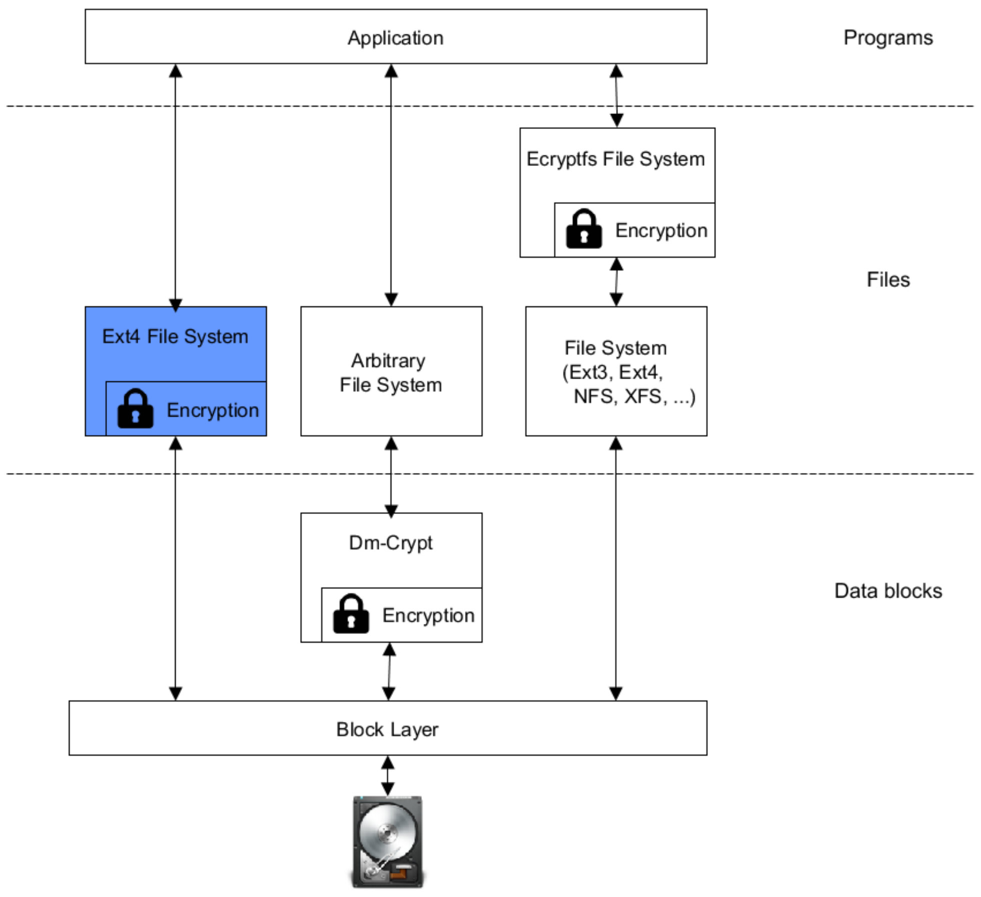


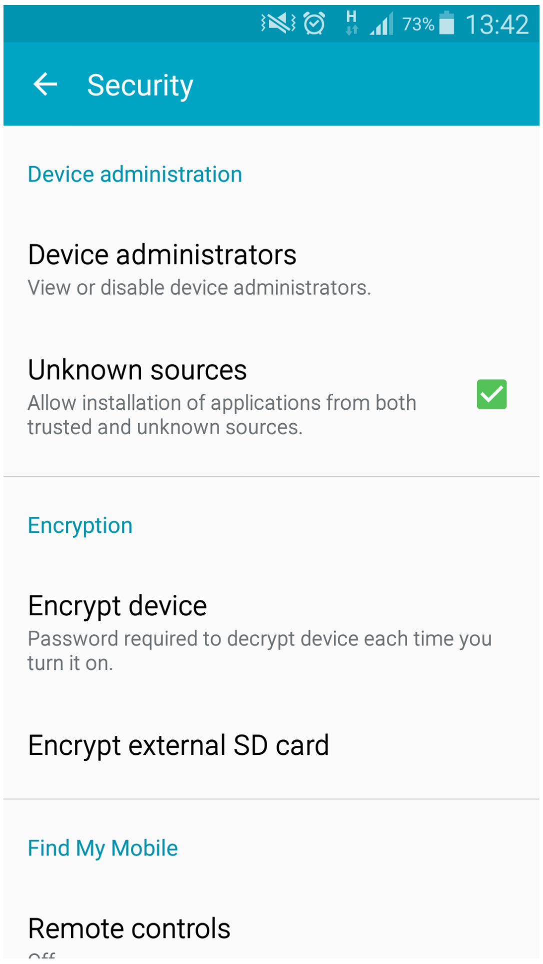
Figure 6: This scematic overview illustrates three well known crpyto systems for unixoid OSes. Ext4, Ecryptfs and Dm-Crypt are shown with their according point of contact in a layered architecture.[29]

1.6.3 *Encryption in Android M (6.0)*

Leemhuis [29] discusses the available encryption in Android. Future versions of Android can be expected to support Ext4 encryption and maybe even to use it *by default*.
The current version of Android M (Marshmallow or 6.0) offers encryption for an memory encryption as seen in Figure 7. In Android 5.0 (L) the encryption was meant to be enabled by default, but shortly before the release the compatibility guidelines of Android were adjusted. So the encryption implemented with Dm-Crypt and the Advanced Encryption Standard (AES) encryption method became only strongly recommended instead of default. At the same time a warning was issued that in future versions this may be none optional. From an external point of view it is most likely that Dm-Crypt did not perform well enough. Low performance requirements are absolutely essential on mobile hardware to prevent high energy intense use of processing and memory resources. Battery usage is an important criterion whether or not to obligate an encryption technology. The assumption that this decision was related to the release of a new smart phone (Nexus 6) with worse performance than expected seems logical. The low performance in this particular case is not due to

Dm-Crypt, but due to a missing driver for AES acceleration in the Andorid 5.0 OS image that was used for the Nexus 6. The current version Android M includes the driver for AES acceleration and therefore eliminate the issue effectively.
The preview [17] of the next version Android N further refines the app-API to consider device encryption. An app can register to a limited functionality after an unexpected reboot of an encrypted device. This is useful if an app handles alarms, messages or calls. The app can then continue to notify the user. The "Direct boot" allows the file based encryption of Android to enable fine grained policies for system and app data. The system uses a device-encrypted store in order to select system data and explicitly registered app data. By default the credential-encrypted store is used for everything else such as system-, user-, apps and app-data. At the boot time the system starts in restricted mode with only the device-encrypted data, without general access to other data or apps. If an app component wants to run in the restricted mode it can be registered in the app's manifest. After the reboot the component will receive a broadcast with an `LOCKED_BOOT_COMPLETED` intent. The system guarantees to make device-encrypted app data available before the unlock. Other data and apps are unavailable until the user enters the credentials for the credential-encrypted data. There is no indication of a change of the encryption system from DmCrypt to Ext4 in the Android N preview.

SD cards are by default formated with FAT32 by the SD card vendors because it is the common denominator among all file systems. FAT32 is quite simple and almost all mobile devices can process FAT32, at least for backwards compatibility. Sometimes missing processing capabilities are the reason for using a simple file system. A low cost micro processor on a digital camera or even a "feature" phone can work effortlessly with FAT32. More advanced devices support other file systems optionally as well.
It is an important question what happens if a FAT32 formated SD card already has data on it. As soon as the SD card encryption with Android 5.0 is enabled for that card, a password with at least 6 characters containing at least one number is required in the given example. In practice a user, will likely choose a password with exactly 6 characters. Furthermore the screen lock pattern is requested for enabling the encryption, if one is used. A password is the only allowed lock mechanism and consequently substitutes any prior used method e.g. PIN or unlock pattern.
The user has the possibility to encrypt the whole SD card or to encrypt only the files created on that device, given the default ROM adjusted by Samsung is used. The ROMs by Google do not yet support such encryption capabilities. The encryption even allows to included or excluded media files of file types *.avi, *.jpg, *.mp3 etc. independently from each other. The encryption key is not shared with any other device or displayed. The encrypted files are bound to the device. If the device is set back to factory defaults the encrypted files can not be decrypted again. The decrypted files would still be usable


73% 13:42

Security

Device administration

Device administrators
View or disable device administrators.

Unknown sources ✓
Allow installation of applications from both trusted and unknown sources.

Encryption

Encrypt device
Password required to decrypt device each time you turn it on.

Encrypt external SD card

Find My Mobile

Remote controls

Figure 7: This screenshot of an Android 5.0 (Lollipop) device shows the available encryption options. The SD card and the device itself can be encrypted. Device: Samsung Galaxy S4

in case of a factory reset respectively lost password. Dm-crypt can be used with any arbitrary file system since it is working on a block level underneath the actual file system. Hence an existing file system *can* be used without the need for reformatting.

If the option "files created on this device" is chosen, only newly created files are encrypted and none of the existing files that might be used on other devices as well. Directories remain uninvolved. Encrypted files viewed on another device show no sign of being encrypted to the user until they are opened. With the file based encryption all metadata (e.g. file name, time stamps and file size) are visible and properly readable. Only file content and *file slack* are encrypted. The *file slack is encrypted* due to the block oriented encryption. Any program trying to use an encrypted file will fail with a message similar to "file may be damaged" or "unknown format". *No* randomized initialization vector (IV) is used for the encryption. The same file encrypted twice results in the same cipher.

1.7 FILE SYSTEM IN USER SPACE (FUSE)

FUSE is a special module within the Linux kernel that allows standard users to create and use their own file systems, without the need to change the kernel or require root privileges. This section introduces FUSE and its capabilities. In the end it will be shown that it is not possible to use FUSE to implement a file slack file system, which is the goal of this thesis.
FUSE was released under GNU GPL and is publicly available online [46] as GitHub repository. FUSE is a well known stable tool and also available in common Linux package repositories. It works with virtual file systems, but not all virtual file systems work with FUSE. It is installed in the kernel, but the file system can be handled in user space. Normally the file systems are handled in the kernel and are not directly available to the user. Exposing virtual file systems to the user space is the core feature.
Users can mount existing file systems e.g. image files (.iso) or archives (.tar) and more importantly create their own file systems. FUSE was originally implemented in C, but it is available for many languages e.g. Python, Ruby, C# and Java.
For using FUSE the user has to mount a FUSE file system into an (accessible) empty directory. Then the FUSE file system is located at that directory. FUSE acts as a mediator. The user can access the FUSE file system. The FUSE file system interacts with the Linux kernel abstraction of the Virtual File System (VFS) module that can only be accessed by a privileged user process. VFS then handles the non-virtual file system and eventually proceeds to the hardware read/write. All users may access the FUSE file system and FUSE may access the VFS module. VFS is privileged. FUSE is not, but FUSE is trusted by the VFS. Therefore any FUSE file system can be used without requiring additional privileged system permissions.
FUSE also allows to implement custom file systems. Really any application that allow data to be interpreted as files can be implemented. Implementing such a file system is relatively simple. There is quite a variety of examples:

- WikipediaFS: allows to read and edit wikipedia articles as if they were files [4].
- SSDHFS: allows file access to a remote file system via the Secure Shell (aka SSH) protocol [34].
- GmailFS: allows to store files as E-mails provided by the well known free mail provider Google [24].
- FTPFS: allows to access remote File Transport Protocol (FTP) servers files by using the FTP [30].

FUSE offers an interface of file operations that can be implemented to create a new file system. FUSE contains a "hello world" example to get started, which produces the output shown in Listing 1. This minimalistic example implements the operations `read`, `open`, `readdir` and `getattr`. These basic file operations are standardized by the POSIX (Portable Operating System Interface) system calls. FUSE knows 25 basic file operations that have a corresponding POSIX system call, supporting reading and writing files, directories, file attributes and file permissions. Additionally, symbolic links are supported. It is possible to implement only a subset of those operations.

Listing 1: Executing the FUSE Hello World example [27]

```
> mkdir /tmp/fuse
> ./hello /tmp/fuse   #program will vanish into the background
> ls -la /tmp/fuse
total 4
drwxr-xr-x 2 root root      0 Jan  1  1970 ./
drwxrwx--- 1 root vboxsf 4096 Jun 16 23:12 ../
-r--r--r-- 1 root root     13 Jan  1  1970 hello
>cat /tmp/fuse/hello
Hello World!
>fusermount -u /tmp/fuse
```

Figure 8 shows the command flow of a directory listing of the FUSE [27] "hello world" example. The user would have mounted the "hello world" FUSE file system already as documented in Listing 1. Then the user executes a directory listing. The underlying GNU Standard C Library (glibc) issues a `readdir` system call to the OS, in this case the kernel VFS module. VFS then recognizes that the FUSE kernel module is responsible for that path and passes the call on to the FUSE "hello world" driver through the `glibc` and the `libfuse` libraries. It then looks up the custom implementation of the `readdir` operation, which it turn returns the directory list as shown in Listing 1 the same way back.

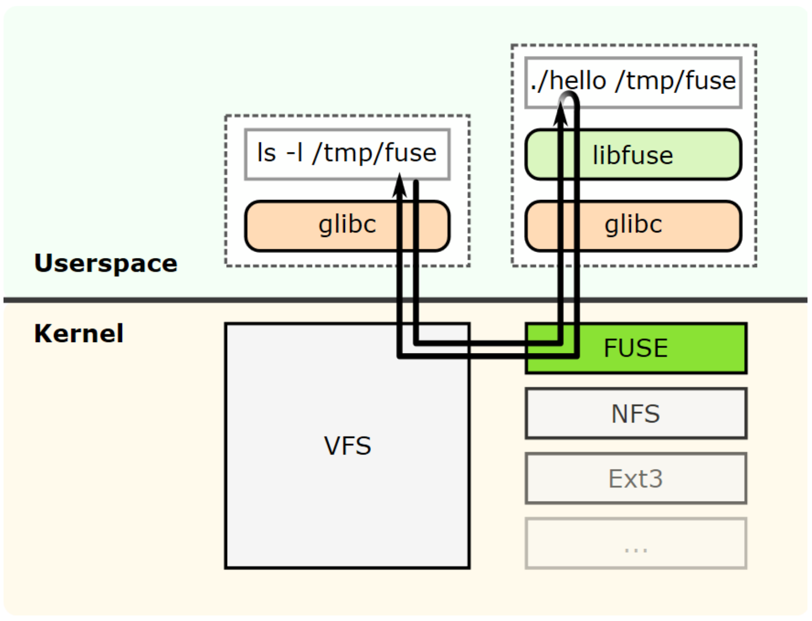


Figure 8: Flow chart of an access to the FUSE "hello world" example [48]

Fortunately Android has FUSE already installed. So the opportunity presents itself to implement a file-slack file system with FUSE. It is used to show the user only a limited view on the file system(s) used in Android. Access to the SD card and to the user's personal folders (downloads, documents, music, etc.) is done through FUSE. FUSE is an additional layer so the user is not directly working on the underlying file system. This allows Android to manage file system permissions flexibly. Android apps may not even leave this file system sandbox that will be a different one for each app.
FUSE is using POSIX system calls and they do *not* allow access to the file slacks. They were not specified to do that. File slacks are a undesired, but logical side effect. It is possible in the GNU FAT32, Microsoft FAT32 and (Ubuntu 14.04 LTS) Extended 4 file system driver implementations to write to file slacks. This can be done as described in Section 1.7.1. However, Android has another additional security layer called Security-Enhanced Linux (SELinux) that allows to define extremely fine granular permissions for system and user processes. Android prevents the file slacks to be modified on a system level.

By default the GNU and Microsoft FAT32 driver overwrite the file slack with zeros.

Unfortunately reading the file slack using POSIX calls can not be done in any of the mentioned implementations as described in Section 1.7.2. Attempts with Java and C over the Java Native Interface (JNI) provided the same results. The application ultimately has to issue POSIX systems calls. POSIX system calls are the communication between application and OS, hence it does not matter which programming language is used.
A file-slack file system can not be implemented using a FUSE due to a lack of file slack access as prevented by SE-Linux policies.

1.7.1 *Modify Slack space*

In Listing 2 a pseudo code abstraction for writing into a file slack is noted. This method is only possible if no components like SELinux interfere. Unfortunately modern Android systems are secured with SELinux policies and the default forbids access this way. The SELinux policies are created and maintained by the device manufacturer for Android e.g. Samsung. It is possible but very unlikely that a manufacturer would intentionally allow access file slacks. This is a major practical obstacle to implement a file-slack file system in Android.

Listing 2: Pseudo code for writing file slacks with comments

```
writeSlack ( File carrier,  String payload )
        metadata = carrier.getmeta() // time stamps, file size
        carrier.append(payload.append(endOfPayloadConstant))
        carrier.setTimeStamps(metadata.timestamps)
        carrier.setFileSize(metadata.fsize) // critical!
return carrier
```

1.7.2 *Read access*

In the Java API there is a way to access a file with a given offset and lengh by using the `RandomAccessFile` class within the `java.io` package as attempted in Listing 3. Unfortunately the according method `readFully` is protected against memory access beyond the file size of the file. The method returns with an `EOFException`: "EOFException: if the end of the source stream is reached before enough bytes have been read." [19] For standard use of the file API the existence of this exception is desirable. In this case the goal is to access the file slack which this exception prohibits effectively.
The Java API does not offer the possibility to access file slacks directly. Therefore another way must be found.

Listing 3: Attempt to read a file slack using Java in an Android environment

```
raf = new RandomAccessFile(PATH + FILENAME,"r");
byte[] inByte = new byte[4096];
raf.readFully(inByte, 0, 4095);
stringbuffer.append(inByte.toString());
```

1.7.3 *Write access*

The write process could just as well be divided in three steps:

- appending the payload data to the carrier file
- truncating the file to its previous size
- resetting the file's time stamps to its previous state

Step 1 and 2 are easily possible via the standard API in Java, but step 3 requires the `BasicFileAttributeView` class of the `java.nio` package (since Java 7). This class is not available in Android operating systems since Android customized the file access API and refined the permission system. Nevertheless Android supports the POSIX [43] system call conventions so it might be possible to directly interact with the operating system avoiding the Android permission system. Android native applications allow to embedd C code (using JNI) which can be used to directly execute system calls and circumvent the Java API.[28] However SELinux policies can not be circumvented, as a result this way of write access is *not permitted*. An adjustment of SELinux policies is necessary, which requires either root permissions or a change of the default policies by the manufacturer.

Part II

FILE SYSTEMS IN THEORY

2 STEGANOGRAPHY IN FILE SYSTEMS

The first chapter of this part contains information about basic file system architecture and discusses a set of possible steganography strategies for file systems. Most importantly eleven candidate data locations are discussed and finally one (file slack) is selected according to specific criteria. Then the risk of presence of steganography software is explained and a recommendation is given.
The second chapter is about about the idea of a file slack file system and its implications. It is reasoned why there is file slack, how it can be increased and how the size of file slacks can be estimated and calculated. Furthermore the life cycle of a file is mentioned. Finally the to-be-expected challenges of a file slack file system are noted.

2.1 FILE SYSTEM ARCHITECTURE

The file system manages the abstraction of files and directories of data and interacts with the adjacent layers as depicted in Figure 9. The top layer contains a shell, which allows the user convenient access to files. The shell is also used by user space applications. The lower layer already interfaces the hardware and describes I/O control, e.g. device drivers or interrupt handlers.
In monolithic operating system kernels the file system driver is a kernel space component and part of the OS. The file system driver cannot easily be manipulated by the user or user space applications. Layered architectures like the one at hand, have interfaces between the layers. As a result a single layer can be exchanged with another component that implements the interface to the lower and upper layer [33].

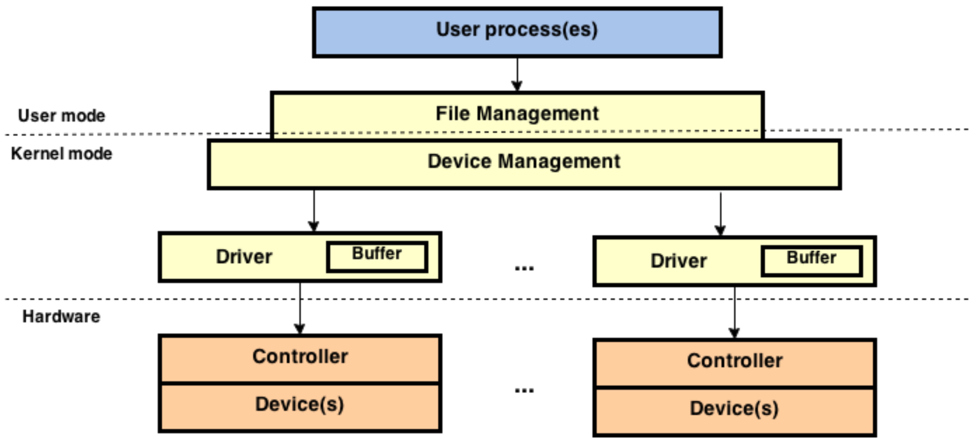


Figure 9: Layered file system architecture

2.2 STEGANOGRAPHY STRATEGIES

The following sections describe various data locations that can serve to hide data. First a general overview is given by the example of hard disks. Then a set of candidates is discussed in detail and one location is chosen that will later get implemented as a proof of concept.

2.2.1 *Hide-out Data Locations by the Example of Hard Disks*

Berghel et al. [23] describe storage hiding mechanisms that are summarized in this section and are further refined in Section 2.2.2.
There is a large variety of storage media available including hard drives, USB flash drives, SD cards, CD-ROMs, DVDs, Blu-rays etc. The architectural challenge of storing data on a medium is often very similar to hard drives. So the example of hard drives shall serve as demonstration of where potential data is unintentionally left behind or can be hidden on purpose. In mobile environments most commonly flash memory is used.
A hard drive has a geometric structure that ultimately contains a set of nested data structures: hard drives, partitions, file systems, files, records, and fields. All these levels can offer space to hide data. Various hiding mechanisms are depicted in Figure 10 and begin with the hard drive level and then work its way down to all the nested data structures. Many of these mechanisms apply to other media as well.

The Host Protected Area and Device Configuration Overlay as seen in Figure 10 item 1 have a reserved area at the end of the disk that allows to modify features of the hard drive e.g. a rescue system to set the device back to factory defaults. This allows vendors to store data which is not seen by OS tools and therefore not affected by format or delete utilities. It is still possible to reconfigure the controller to allow access to all blocks, but it is not possible by default. This represents a "physical" hiding feature to ultimately improve the usability.
Standard OSs require that the hard drive is partitioned into virtual file system(s) before it can be used, even if there is just a single virtual file system present. A partition is a set of consecutive blocks[1] on a medium that the OS can see as a separate logical volume. In the Windows OS family separate logical volumes are called *drives* and in the UNIX OS family referred to as *mount points*. Hard drives using DOS partitions have a reserved space in the beginning to hold the Master Boot Record (MBR), see Figure 10 item 2. This is important for the boot process and this will contain a partition table, if a partitioned medium is present. The MBR is gradually replaced by a newer boot process defined by Unified Extensible Firmware Interface (UEFI) which provides a compelling pre-OS environment including network capabilities. UEFI is combined with the Globally Unique Identifier (GUID) Partition Table (GPT). The GPT uses GUID partition entries to store information about the partitions. There can be extended par-

1 Blocks in Microsoft terminology are the same as clusters. The terms can be used interchangeably.

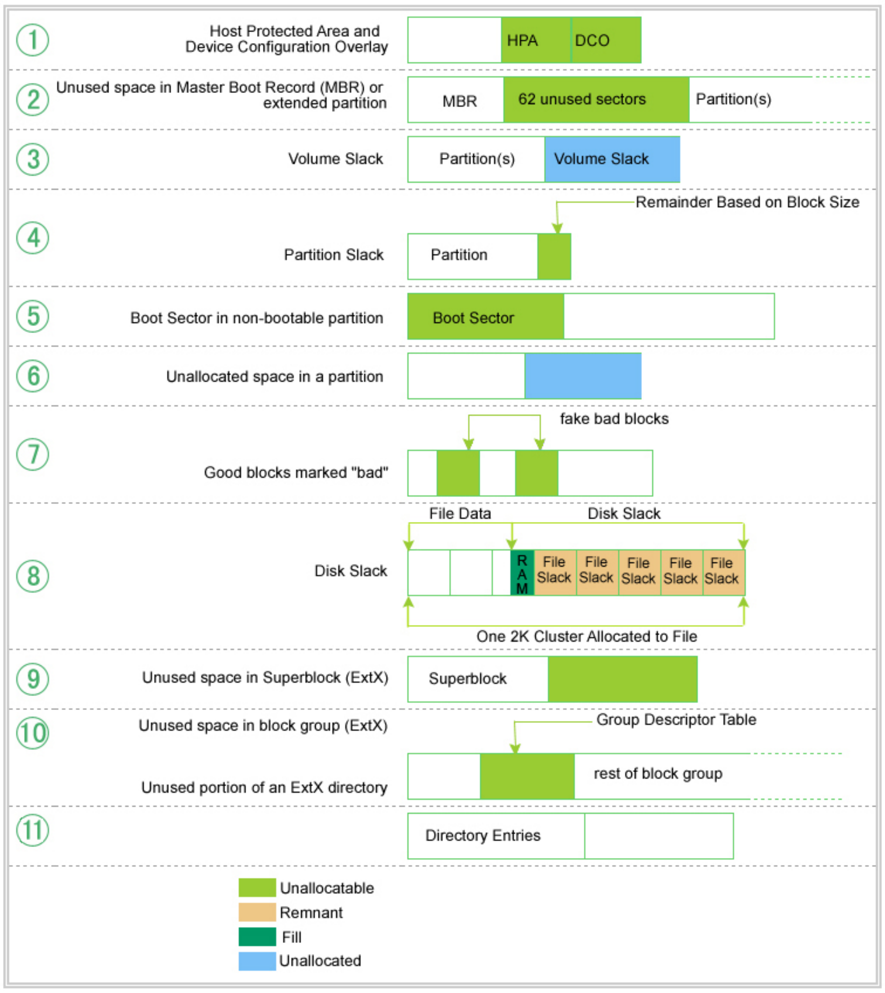

1 Host Protected Area and Device Configuration Overlay
HPA
DCO
2 Unused space in Master Boot Record (MBR) or extended partition
MBR
62 unused sectors
Partition(s)
3 Volume Slack
Partition(s)
Volume Slack
4 Partition Slack
Remainder Based on Block Size
Partition
5 Boot Sector in non-bootable partition
Boot Sector
6 Unallocated space in a partition
7 Good blocks marked "bad"
fake bad blocks
8 Disk Slack
File Data
Disk Slack
RAM
File Slack
One 2K Cluster Allocated to File
9 Unused space in Superblock (ExtX)
Superblock
10 Unused space in block group (ExtX)
Group Descriptor Table
rest of block group
Unused portion of an ExtX directory
11
Directory Entries
Unallocatable
Remnant
Fill
Unallocated

Figure 10: Loop holes in digital storage devices [23]

titions, which can contain logical partitions. Partitions must start on a cylinder boundary (this is hard drive specific). The legacy MBR only uses one sector, the rest of the sectors in this cylinder are unused. For mobile environments almost only flash memory is used and with the state-of-the-art GPT/UEFI.

There is a pattern here, with each abstraction level another hiding place turns up.

If the partitions do not use all of the space the hard disk offers, the remaining space can not be used by the OS (by default). This space is called volume slack as shown in Figure 10 item 3. Of course a hard disk can have several partitions filled with data. Volume slack also occurs naturally if the number of sectors in a partition is not a multiple of the cluster size.
If one partition is deleted only the metadata about the partition is deleted e.g. start address, length etc., but the data in the partition is still alive and hidden, see Figure 10 item 4.
Every partition contains a boot sector. If it's not a bootable partition, the MBR of that partition offers space for hidden data as depicted in Figure 10 item 5.
Also any space within a partition that is not allocated to a file could still hold data, compare Figure 10 item 6. Typically OSs clear data late or lazy. That is right before a new allocation of that area, meanwhile it is untouched by the file system.
File systems are capable of recognizing faulty clusters e.g. `Bad Block` (`0xFFF7`) in FAT or `$BadClust` in NTFS. Well working "good" clusters could be marked as bad to make the file system leave this cluster untouched, see Figure 10 item 7.
Disk slack as it is referred to in Figure 10 item 8, is a byproduct of an acceleration mean to speed up the disk's access by always writing a whole cluster of sectors. In very old OSs data was written to the disk with padding if needed. Berghel et al.[23] defines the data from the file end to the next end of the cluster as called RAM slack. It used to be data from the RAM memory to do the padding with. Berghel et al.[23] defines file slacks differently than this thesis. They call unused sectors of used cluster, file slack . In contrast, this thesis understands file slack from the end of the file to the end of the cluster. In the Extended file system family (ExtX) the superblock allocates 1 KByte and 788 Byte can be used depending on the cluster size, see Figure 10 item 9.
Furthermore the ExtX group descriptor is only 32 Byte long. This is followed by a block bitmap that must start on a cluster boundary. Hence only 32 Byte of 1024 Byte are used, leaving 992 Byte empty as illustrated in Figure 10 item 10, for hiding data. ExtX does store directories just like files in a whole cluster. Similar to file slack there is directory slack from the last directory entry to the end of the cluster which can be seen in Figure 10 item 11.
Another very powerful way of hiding data that must be mentioned are Alternate Data Streams in the New Technology File System (NTFS) that allows files to have an attribute that can store large amounts of data. There are many more strategies that are not further discussed such as swap files, changing unused registry files, rename a file e.g. to .dll, hiding data in text documents e.g. with a white font, using

comments in a .html file, steganography in databases, compression and encryption mechanisms.

When partitioning is done, the next layer is about the file system that is needed for the OS in order to access data. Modern OSs support several different file systems, but they require at least one.

2.2.2 *Candidate Hide-out Strategies and Evaluation*

In this section eleven steganography strategies for hiding secret payload data are discussed and evaluated for their potential space, quality and important characteristics. For the following proposals assume a 4 GByte Secure Digital (SD) card with a 4 KByte block size and formated with the file system *FAT32 (File Allocation Table 32)*. The numbering of the alternatives in this section is independent from Figure 10.

All the alternatives shown in Table 2 have a special property. This property is to be invisible to unaware user. All eleven alternatives can hide data without a change in the total memory usage of a medium shown to the user. It is the goal to find a data location that is not part of the normal free space calculation. For example under Linux the command `df` (disk free) would show exactly the same use of disk before and after storing the payload data. Of course similar tools like `df` and even the OS should not be aware of the payload data. Each alternative describes a data location and an estimate of its properties.

Alternative 1 (unused space in MBR): The MBR contains several boot parameters that are necessary to detect the file system and contains some code that is responsible for continuing the boot process. The MBR starts at the first sector on the medium and it typically does not use up the space that is available, which is exactly one block. In this example it is 4 KByte big. The MBR would also have to fit into a block of 512 Byte for compatibility of systems with a cluster size of 512 Byte. The MBR typically does not even need these 512 Byte. Hence at least 3.5 KByte are available for hiding data. In practice it may be DHFShundred Bits more. The MBR is also a popular place for viruses and other malware. In the MBR malware can modify code early before other software or an OS is loaded. Malware wants to survive a system reboot and wants to be started with every system boot. Therefore it is also a popular place for Anti-Virus Scanners to check the data stored there. As a result the MBR is maybe not the best choice to hiding additional data. Hidden data could also be mistaken for malware.

Nr.	Data location	discovered by automated search tools	space gain estimation	necessity to change (meta)data	quality of hide-out	persistence	Comment
1	unused space in master boot record	yes	< 4 KByte	yes	medium	no	
2	boot record space in non-bootable partition	yes	512 Byte per Partition	yes	medium	no	
3	unused/outdated file system attributes: "heads"	unlikely	several Byte	yes	good	no	
4	file slack	unlikely	several cluster/2 per file	yes or no, depending on the methodology	good	yes	
5	partition slack	unlikely	several 100 MByte	yes, partition table	good	yes	
6	unallocated space	likely	several GByte	yes, administration info	medium	yes	
7	deleted files (`0xE5` FAT32 specific)	yes	several GByte	no	bad	yes	
8	use good blocks marked as bad	unlikely	several clusters	yes, administration info	good	yes	
9	Encode bits in error tolerant data (e.g. time stamps)	no	several Byte per file	yes	good	yes	
10	Overwrite the second FAT (FAT32 specific)	likely	about 128MB	yes, severe	bad	yes	might result into file system errors
11	Encode data with ordering of used blocks	no	1Bit per Block	no	good	no	easily destroyed

Table 2: Comparison of specific locations for steganography in file systems.

Alternative 2 (boot sector in non-bootable partition): Assume the physical memory is partitioned in not only one but several partitions. Each partition can have its own file system and would be independent from the file system on the other partitions. And assume further that all the partitions hold again FAT32, then each first sector of all partitions would be reserved for the corresponding MBR. But only the first MBR of the first partition would actually be necessary to start up the OS contained on the first partition. If the other partitions do not hold an OS then they are not bootable so there is no need for a MBR at all. Nevertheless the MBR exists also for non-bootable partitions. Hence all MBRs except the one of the first partition could be salvaged as hide-out for steganographic data. If the partition is viewed as a block device with a hex editor the steganographic data would be visible. Each MBR is one block long. In this case the block size is 4 KByte per partition. Typically only one partition will be needed on a relatively small SD card of 4GB. Though it is still possible and plausible to have several partitions. However, a large number of partitions is uncommon e.g. ten would definitely attract attention and therefore be suspicious.

Alternative 3 (unused file system attributes): The MBR contains metadata and boot code. The metadata consists of the Boot Parameter Block where several characteristics about the file system and the used hardware are provided. The technology of the MBR reaches back to the first hard disks. Some of the boot parameters there are deprecated by now and not used anymore. They still exist for backward compatibility. For example one parameter stores the number of heads of a hard disk. Hardware specific data for memory is nowadays accessed via the S.M.A.R.T. (Self-Monitoring, Analysis and Reporting Technology). Hence it is safe to change the value of the "heads" variable in the MBR (excluding legacy systems). It is very unlikely that changes in this variable would be noticed. The "heads" variable itself is only 1 Byte long, so the reachable capacity is extremely limited. There are only very few other candidate parameters to increase the capacity. Due to the low capacity this location qualifies only to store some meta information e.g. a marker where to find more data or a flag that this partition contains other structures for steganography. It is very unlikely that this variable would be changed during normal operations so it would deliver good persistence.

Alternative 4 (file slack): A file is stored in one or more clusters. File content data starts at the beginning of the first cluster. If one cluster is not enough to hold the file, a second block is allocated and so on. Depending on the actual file length the last cluster in the cluster chain is probably not used completely. Hence at the end of the cluster can be some space left over.
A file containing only 1 Byte uses one whole 4 KByte cluster that can not be used by any other file. *4095 Byte (4096 Byte - 1 Byte) of file slack*, this represents the worst utilization but the highest possible slack space. On the contrary a file that has exactly 4096 Byte utilizes

the offered memory to 100%. Then there is no slack space at all available. So statistically, the last block of a file chain will be used for 50% of its capacity on average. That leaves half a block empty for each file on the file system. This space can be vacated by a secret payload. If the memory is examined byte by byte of course the slack space will be visible, but the viewer would still have to be aware of the file system metadata, in particular the file length to tell where the file ends and the slack space starts. If a file has been shortened, securely erased or overwritten there can be some leftover data in the slack space. So data in slack space can have various plausible reasons. The file systems' user interface does not allow access to file slack as it is a file system internal effect. Even a virus scanner would not be able to access it and check its content. However if a file gets extended, the file slack will be overwritten. Slack data can be overwritten during normal operations so the data integrity is definitely a problem here.
The potential space that can be harnessed depends strongly on the number of files and the cluster size and the size of the medium, but can be conservatively estimated with 10 to 100 MByte, this was experimentally determined. With an according setup several GByte are possible. File systems keep track of the files creation, modification and access timestamps. Depending on the method of accessing a file slack they might have to be changed to efficiently hide the slack access.

Alternative 5 (partition slack): Similar to file slack, partition slack is the unused space after the end of a partition. In particular the last partition on a physical medium potentially offers a slack space depending on the size of the medium and size of the partition. The OS may not allow the access of partition slack in user space. The partition slack is essentially unallocated space once the partitioning of the medium took place. Partitioning is rarely changed over the lifetime of the installed OS and sometimes not even when a new OS is installed. The space gain for payload data can be constructed by the partitions sizes.

Alternative 6 (unallocated space): Unallocated space in a file system can hold secret data and appears as free space at the same time. The data could be allocated by the memory manager and be provided to any process. Therefore it is highly volatile. In order to store data here, several restrictions must be made. The file system, respectively the memory manager has a known allocation strategy and the user does not allocate all the available space. For example FAT32 must (and typically does) use the allocation strategy *"next free cluster"*. This means the file system is filled up from the lower memory address up to the higher addresses in a linear way. If a file in a low address space is deleted its cluster become free again and it will be used before an address in the high address range. In conjunction to that the user is aware that always x Byte of the disk must remain unallocated in order to keep space for hidden data available. Then a program could start to fill up the disk from the high memory addresses to the lower

ones with x Byte of payload.
Furthermore this method would need a newly initialized physical medium. An inhabited medium could already have high address space allocated. This approach is known in literature as *hidden volume*.

Alternative 7 (deleted files): There are several levels of file deletion. In the case of FAT32, the driver only marks files as deleted by prepending the magic number `0xE5` to the file name instead of actually zeroing all the metadata and file content. Hence files that have not been overwritten can be at least partly recovered. If the file was constructed of more than one block, the first copy the block chain is will be lost. The file chain is stored redundantly and can therefore still be available for a deleted file. A file that has only one block does not need the block chain information and it can be recovered by simply undelete it, which means to remove the deleted marker `0xE5`. Of course if the file cluster was overwritten with another file, the overwritten data is lost. There are many undelete tools for FAT32 that can recover such files. This undelete method is well known and as a result this is not a very good place for hiding data. However major parts of the medium could be salvaged in this way. The user has to be aware of space limitations to avoid overwrites. Fragmentation might become a problem because it also might allocate big consecutive chucks of data. Likewise defragmentation is a problem because a lot of clusters are rearranged on the medium and it is likely that slack data is overwritten.

Alternative 8 (good blocks marked bad): Storage mediums have a limited lifetime and with growing age some sectors start to fail. Modern flash memory can offer reserve sectors to mitigate this problem. Older media do not have such a feature and need the file system to react to this failure. In FAT32 the cluster is marked as `bad` in the file allocation table and will not be used any longer. Nowadays media take care of this problem themselves and bad clusters should not occur anymore. If they do nonetheless it is a sign that the mediums' lifetime comes to an end.
Some valid clusters could be marked as bad and still be used to hold secret data. This way they would be guaranteed not to be used anymore by the file system. The hidden data would not be overwritten. It is visible if the medium is cluster wise inspected by e.g. a forensic tool. There is a limited but existing risk that a file system check tool such as `chdisk` (checkdisk) would recognize valid clusters that were "mistakenly" set to bad and set them back to free/valid (`0x00`). A disadvantage is the presence of bad clusters in purely image files, because then bad cluster are obsolete. There will not be a fault in a virtual disk. Bad clusters then are very suspicious.

Alternative 9 (encode bits in error tolerant data): Encoding payload data in error tolerant data is one of the central topics in steganography. In case of encoding data in the file metadata, only a small amount of data can be hidden. In FAT32 the access time is often not used at all, sometimes not even displayed. A few Byte per file could be harnessed to store hidden payload data. It is rather obvious that there is suspicious data if an investigator would check. It is very likely that an investigator would notice this. If the data would be preprocessed so that only valid dates would appear the chances are better. Even better if the access date would be in the past, not in the future, this would be even more plausible and would probably go unnoticed. If the file system is mounted in a device that does set the access time, then the payload data will be overwritten. A user with standard read and write permissions can set the time stamps to any value, without a need to change the OS or the file system. Therefore this is a universal method that works for any file system that supports such timestamps, which is a basic feature found in most file systems.

Alternative 10 (Overwrite the second FAT): Overwriting the redundant copy of the file allocation table (FAT) is dangerous and a destructive way of hiding data. The redundant copy exists to repair the first copy in case of a system crash. This purpose would not be served anymore if the second FAT is overwritten. Also the second FAT is regularly written. An invaild second FAT would definitely be noticed by file system checking tools, probably seen as a large area of corrupt data. The size of the FAT depends on the size of the medium. For each cluster there is one 32 Bit address stored. The cluster size is typically either 512 Byte or 4 KByte (or at least a power of 2). To get the size of the needed FAT the medium size has to be divided by the cluster size. This location is quite obviously visible in case that some error with the file system occurs and the payload in the FAT will very likely not survive a repair attempt depending on what the repair tool tries to do exactly. The repair attempt is also very unlikely to be successful at all since vital data has been overwritten. Even if no errors occur at some point the changes from the first FAT will be backed up in the second FAT.

Alternative 11 (Encoding data with ordering of used blocks): Encoding data by selecting a special order of the used clusters is an approach known in databases. Watermarking in databases uses a fixed order for its data sets. The order does not influence the functionality, but lets a binary copy be discriminated from a copy, that merely happens to have the same content. Exporting a database dump and then importing a dump will use the default ordering and not the predefined watermark ordering.
This methodology can be reused with data allocation strategies of a file system. The default ordering of cluster is "next free cluster". If this was changed to another pattern, secret data could be encoded. For example a file needs ten clusters and the memory manager has an area where 20 clusters are free consecutively. The first cluster would have to start at the first address anyway, but by choosing to either use the next cluster or to skip a cluster one Bit of information can be encoded. So the potential space gain is identical to the number of clusters in the file system. This method will produce a lot of fragmentation which should not be defragmented otherwise the payload data is lost. Also

the read and write process for the payload data would have to be able to change the allocation mechanism which is part of the file system driver.

2.3 CANDIDATE SELECTION OF A HIDE-OUT LOCATION

The candidate shall be selected by several criteria. The first criterion is that the space used to store the payload data should not be visible to the user or the OS. Hence taking care of the central requirement of the payload being hidden. This can be tested by determining the free space on the medium via the OS. Before and after the payload data was stored the free space must be the same. A simple method like setting the hidden attribute of a file would not fulfill this criterion, but in this case all eleven candidates discussed in Section 2.2.2 reach this criterion.
The hide-out quality shall be rated "good" as discussed in Table 2 to secure the primary goal of this method to hide the payload as well as possible. That leaves candidates 3, 4, 5, 8, 9 and 11.
Furthermore the data location shall at least offer enough space for several text documents, DHFSpictures or multiple table calculation files. This will require at least 20 MB for average size and average resolution of a digital camera picture. Hence candidate 8 (good blocks marked bad) can be disqualified. One cluster offers only 4 KByte and marking many blocks as bad would indicate that the medium is damaged and therefore would attract too much attention. 20 MByte would need 5120 clusters. S.M.A.R.T. will definitely diagnose a medium bad health status if 100 clusters are bad. Even worse is candidate 9 (encoding bits in error tolerant data) that can only encode several byte in file metadata and does not provide enough space. The additional requirement that the payload data in the binary image shall not be stand out if it is read with a hex editor, eliminates candidate 3, leaving candidates 4 and 5.
Candidate 5 (partition slack) provides naturally only DHFSMByte due to rounding remainder or sometimes even none at all. To guarantee at least 20 MByte, a partition has to be deliberately shirked or a small partition created and then deleted. Automatic partitioning tools will in most cases not waste this space. On these grounds it can be suspected that the user left this space intentionally unpartitioned.
Candidate 4 (file slacks) offer enough space on a moderately filled file system. Essentially the space is defined by the number of files on the file system which can be roughly estimated by 2 to 10% of the allocated medium from experience. For example on a 4 GByte medium filled to 50% this would mean about 20 to 100 MByte. This estimation and the exact determination of the available payload space is illustrated in more detail in Section 3.2.4. The data location exists without the need for additional user interaction to allocate the payload as it would be needed with creating and deleting a partition (candidate 5).
Both for partition slack and file slack the data could be viewed with a device block viewer which is not a standard tool that comes with an

OS by default. File slacks however have a greater risk of being accidentally overwritten by normal file operations, while partition slack is assumed to be quite static during normal use. Nevertheless potential system upgrades, reinstalling an OS or setting back to factory defaults will also destroy data in partition slack. A trade off between better data persistence in partition slack and better hiding quality has to be made. The goal of this thesis is to leave the least amount of traces possible as mentioned in Section 1.2, as a result candidate 4 *(file slack)*, is chosen.

2.4 PRESENCE OF A STEGANOGRAPHY TOOL

Whenever a user wants to use steganography software it has to be installed on the system at hand. The software is intended to hide the payload data on the target file system. Even if it does so perfectly, the mere existence of the software indicates to an inspector that it was installed and used to hide the payload data. With knowledge of the software source code, the methodology e.g. hiding data in file slacks will be known. An inspector would know where to look for hidden data. Therefore the steganography effect is decreased.
There are several strategies against this risk:

1. A rather theoretical solution is to memorize the code of the tool. Then type it into the system, compile it, use it and delete it again. This would leave the minimal amount of traces, but is also very unpractical.

2. It might be possible to develop a self-modifying program, that would destroy itself. It is however extremly hard to create and modern OSs would likely not support such a program or even forbid it.

3. If the software would be part of wide spread packages, frameworks or even included by default in a common OS, the presence of the steganography tool would not imply that it was deliberately installed and used for its sole purpose, to hide data.

4. Develop a tool that also tries to hide the existence of itself. This is however out of scope of this thesis.

5. Leave the software installed and prepare an excuse e.g. this tool was evaluated on my device, but in the end not filled with data. This might work for legal issues.

6. Accept the risk and leave the software installed.

7. Uninstall the software before a check or respectively uninstall right after use and reinstall it from an external source right before the next use. This needs user action and might be forgotten by an undisciplined user.

The last two options 6 and 7 seem to be the most realistic choices. Choice 6 is very simple and need no attention of the user, so it is the default option for users that do not care much. Choice 7 is still simple enough so that any user is capable of doing so. It is more effort for the user, but the protection is significantly improved. Therefore option 7 is the recommended one.

3 A FILE SLACK FILE SYSTEM

This chapter is about file slacks, why they exist, how they can be increased, what are good carrier files and how the file slack size can be estimated and calculated. Furthermore the life cycle of a file is mentioned. Finally the to-be-expected challenges of a file slack file system are noted.

3.1 TERMINOLOGY

This section explains some technical terms, so the reader is clear on what is meant.

Carrier file: A file that carries additional data in its file slack. Here the files stored on FAT32 are meant.

Deniable Hidden File System (DHFS) is a proof-of-concept file system that this thesis proposes.

Payload file: The files stored in DHFS are payload files. DHFS uses carrier files to store payload files in the carrier files' file slack.

Carrier file system: Here FAT32 is the file system that contains the carrier files.

Payload file system: Here DHFS, is the file system of the payload files that inhabit the carrier file slacks.

3.2 FILE SLACK

File slacks exist in any block based file system, which is the dominant class of file systems. A file slack file system can be potentially implemented on any block based file system. FAT32 is chosen as carrier file system, because it is very simple and wide spread e.g. the default file system for SD cards. FAT32 is quite old and represents the "common denominator" of file systems. Many systems still support FAT32. Modern file systems have advanced features, but might not be supported by such a large diversity of systems.

3.2.1 *Cluster Justification*

Files are stored in blocks or clusters. The terms are exchangeable for this thesis. File Systems use clusters as their smallest addressable unit.

Clusters can consist of one or more sectors. Sectors are the smallest addressable unit in hardware such as hard disks, Solid State Disks (SSD), Universal Serial Bus (USB) sticks or flash memory in general. It became a widely accepted standard to use a cluster size of 4 KByte, before that 512 Byte was commonly used. Details for cluster sizes can be found in Table 3.
There is a need for clusters in order to keep large data amounts manageable. If each Byte or even each Bit on a hard drive were addressable on its own, the address space for accessing the data would be extremely large. For example, if 1 GB of data would be addressable per Byte, roughly one Billion (exactly 1 073 741 824) different addresses must exist to access them all. As Equation 1 illustrates, this needs an address space of 2^{30}. Which means that a 30 Bit long address variable is needed. With 32 Bit addresses and an address space of 2^{32}, 4 GByte of data can be addressed.

$$1\,\text{GByte} = 1073741824\,\text{Byte} = 1024 \cdot 1024 \cdot 1024\,\text{Byte} = 2^{30}\,\text{Byte} \quad (1)$$

That means the address variable would have to have at least 30 Bit to hold all addresses. Hence a 32 Bit variable was used. A 32 Bit architecture is just enough to process such an address variable in one calculation step. However, even 32-Bit architecture systems have the desire to address more than just 4 GByte of storage, so Bytes are grouped into clusters to increase the addressable data amount without further increasing the address space. 64-Bit architectures could of course process a 64 Bit address variable with many addresses more in single step. Nevertheless file systems need to be independent of the architecture, so the cluster concept is continued. Additionally a lot of flexibility for a wide range of storage capacities is gained.

On hard drives with moving heads, mechanical repositioning of the head is a very slow operation in the range of milliseconds while reading from the disk. When the heads are already positioned correctly it takes very little time to read in comparison, in the range of microseconds. Moving the heads is expensive. At the point in time when a file is deleted, it is not necessary to actually overwrite the data with zeros or random data, which would be a strong delete operation. Instead it is enough to mark a file as deleted, and set the used memory space back to unallocated. Hence the data is only marked deleted and no further actions are needed, therefore it is significantly faster than overwriting. This default *lazy delete* strategy absolutely makes sense considering performance. It is possible but not default to use security tools actually overwrite the data.

This can be proven experimentally by comparing the time of copying a 5 GByte file to deleting a 5 GByte file. If the time is not more or less the same, an overwrite did not occur.

Today hard drives with mechanical parts get more and more obsolete. New media like flash memory is a significantly faster alternative and flash memory prices arrive at a quite affordable level. Particularly for mobile devices flash is predominating. Due to the lack of mechanical parts the flash memory is quite indifferent to vibrations, also mechanically more stable.
The cluster concept is used independently of the underlying hardware.

3.2.2 *Increasing Payload Space*

If the available payload space is too small, it can be increased. As mentioned in Section 2.3, the offered size by this DHFS depends essentially on the cluster size and the number of files. There are several options how that space can be increased:

1. Creating many small files e.g. text files that contain only one Byte drastically increases the slack space. That means the cluster size minus one Byte is the created slack space, which is the maximum slack per file. This can be repeated with many files towards the maximum slack of the partition, which is a partition completely filled with one Byte files.
2. Formating the medium intentionally with a large cluster size also increases space significantly. There are cluster sizes of 512 Byte, 1, 4, 8, 16 and 32 KByte possible. Choosing 32 KByte (32768 Byte) is the maximum and yields a maximum slack of 32767 Byte per file.
3. Providing a larger medium so it can hold more files also increases the potential space. FAT32 has a maximum volume size of 2 TByte, with a cluster size of 4 KByte, which is still more than enough for a mobile environment.

Option 1, creating many one Byte files can not be recommended, because their usefulness and existence is hard to be explained. This is suspicious. It is still possible to design the file lengths of files so that the remainder of the file size divided by the cluster size is large. However this is also not recommended at this point because a plausible explanation for such files is hard find.
Option 2, choosing a larger cluster size than default is a good approach. It is unlikely to be discovered and could still be explained by a misconfiguration of a formating tool. Still it is not standard to change this, so it leaves a small trace.
Option 3 offers the most advantages. A large medium has a lot of space to store files, hence the number of files can be overall larger than on a smaller medium. As seen in Table 3, the default cluster size rises with the increasing size of the medium stepwise. As a result of a large medium two key factors are positively influenced, on the one hand the increasing maximum number of files and on the other hand the increasing cluster size. Additionally it is a simple and convenient measure. MicroSD cards for tablets or phones with 32 GByte are inexpensive, even microSD cards with 128 GByte are available for about 40 € at the time of writing this thesis. In future the available memory on a microSD cards can be expected to rise.

The conclusion of this section is that the carrier medium should contain as **many small files**, then a lot of slack space is available. The bigger the medium the better. Using a big medium is the recommended (option 3) which automatically includes a large cluster size (option 2).

medium size	cluster size
500 MByte - 8 GByte	4 KByte
8 - 16 GByte	8 KByte
16 - 32 GByte	16 KByte
over 32 GByte	32 KByte

Table 3: Standard cluster sizes for storage medium sizes with FAT32.

3.2.3 *Carrier Data Suggestions*

As Section 3.2.2 describes the medium should contain many small files to provide slack space. The worst case is an empty medium with no files at all. No carrier files means no file slacks.
There are *no requirements* on the carrier file type, many other steganography tools do require a file type or format e.g. *.jpg, *.mp3 etc.
A plausible way to fulfill the criterion of having many small files, is to create a mirror of a large website. It typically contains many *.html files which can be estimated with about 20 KByte each. An incomplete mirror of the Johannes Kepler University website delivered about 45000 files with a total size of 2.22 GByte. This data stored on a medium with a cluster size of 4 KByte produces about 80 MByte of slack space.
Some applications also have a lot of files e.g. the integrated development environment "Eclipse - Mars" (3668 files) is 318 MByte large and offers about 10 MByte of slack space with again 4 KByte cluster size.
A large collection of single icon files could also produce a great slack space, particularly if the icons are small and the medium is large.
Particularly well suited is the source code repository of the Linux kernel. The version 4.6.3 has 53645 files which needs 689 MByte on a medium with cluster size 4 KByte of which only 594 MB are content, consequently 95 MByte (14%) are file slack. Git was developed for the development of the Linux kernel, so it is (one of) the most important projects using Git. A large number of code repositories are available under *github.com*. From testing experience they typically satisfy the criterion of having many small files.
The user on a mobile system e.g. a smart phone will use data that is on the microSD card already. These could be pictures taken with the device, music to listen to while traveling, some downloaded data and probably some data of applications that use the external storage.

3.2.4 *Estimation of Payload Space Size*

Media with a size over 32 GByte are definitely recommended to reach useful capacities. Additionally the assumption that many small files are stored on the medium must hold. Then the file slack size can be estimated with an statistical average of 50% of the cluster size per file as formulated in Equation 2. These are 16 KByte per file in case of a 32 KByte cluster size or 2 KByte in case of a 4 KByte cluster size.

The website mirror mentioned in Section 3.2.3 is estimated with 88.18 MByte. Actually it delivers 80 MByte slack space so the estimation in this case is slightly above the real value here.

$$\mathrm{TotalSlackSize} = \mathrm{NumberOfFiles} \cdot \frac{\mathrm{ClusterSize}}{2} \tag{2}$$

$$\mathrm{TotalSlackSize}_{\mathrm{Website}} = 45142\,\text{files} \cdot \frac{4\,\text{KByte}}{2} = 88.18\,\text{MByte} \tag{3}$$

This leads to the very rough general estimation that 1 to 15% of the medium size can be harvested for slack space.

3.2.5 *Calculation of Payload Space Size*

The estimation of the previous section provides only a very rough estimate. All the needed factors can be known a priori, which leads to an exact result. This section provides the exact calculation of how much slack space is available for a given medium. In order to get the number of used clusters of a file the file length divided by the cluster size has to be rounded up as Equation 4 describes. Then the maximum size of the file on the medium is determined by Equation 5. After that the slack size of the current file is the difference of the actual file length and the maximum size as seen in Equation 6. Combining these three equation by substituting them and adding a sum over all available files n on the partition Equation 7 emerges. It is recommended, but not necessary, to use only one partition on the medium. DHFS allows only the use of one partition.

$$\mathrm{NumberOfClusters} = \left\lceil \frac{\mathrm{FileLength}}{\mathrm{ClusterSize}} \right\rceil \tag{4}$$

The cluster size can only have certain values which must be powers of 2.

$$\mathrm{MaximumSize} = \mathrm{NumberOfClusters} \cdot \mathrm{ClusterSize} \tag{5}$$

$$\mathrm{SlackSize} = \mathrm{MaximumSize} - \mathrm{FileLength} \tag{6}$$

$$\mathrm{ClusterSize} \in \{512, 1024, 2048, 4096, 8192, 16384, 32768\}$$

$$\mathrm{TotalSlackSpace} = \sum_{i=0}^{n} \left\lceil \frac{\mathrm{FileLength}_i}{\mathrm{ClusterSize}} \right\rceil \cdot \mathrm{ClusterSize} - \mathrm{FileLength}_i \tag{7}$$

3.3 FILE LIFECYLE

There are two principal parts separated in OSs, the user space and the kernel space. The user space is the set of programs and memory the user can directly influence. The kernel space can not directly be influenced. It is the area of the OS and it may only be accessed via the interfaces the OS provides.
The lifetime process of a file is as follows: The first step is the file creation. A process requests space to create a file. This request is passed from the user space via a system call to the kernel space. Then the kernel forwards this request to the file system driver. The memory manager of the file system driver allocates a previously unallocated space, marks it allocated and returns the according start address. By default the allocation guarantees the according memory to be all zeros. Hence the memory is overwritten at allocation time, not before (lazy delete). This can be time consuming if the process asked for a big chunk of memory. This also depends on the write speed of the medium.
There are memory wipe tools that implement secure delete that overwrite the deleted files immediately, sometimes even multiple times to be sure. These tools could also overwrite unallocated data by allocating it and then releasing it again. It is much harder to overwrite file slacks since the OS does not allow accessing it. But it is still possible to do so, by copying data into newly allocated memory and securely deleting the original. However this is a very time consuming process and is very specific to file slacks, which are typically ignored during normal operations. A more systematic approach would be to start up a system from a live CD or bootable USB stick to be able to have complete access to the whole medium instead of starting an OS that runs on the medium that should be cleaned. On laptops this is easily possible, on a smart phone or a tablet this is more difficult because the boot process is more restricted and there are no external drives like a CD-ROM drive.

3.4 PROBLEMS OF A FILE SLACK FILE SYSTEM

Assume the payload file is a PDF (Portable Document Format) file with a size of 1MB. With a statistical average of 2 KByte file slack per file 1024 KByte / 2 KByte = 512 file slacks would proximately be needed to store the file. Now the question arises which carrier files to select and how to store the order of the 512 parts. Managing this information and additionally the metadata are essential duties for each file system. The answers to these questions will be discussed in Section 4.7 and Section 4.8
Due to the distribution of data over many file slacks, advantages of a sequential read of the medium can not be used. This will much more be like a random access. This in addition to the large number of file slacks might be a performance issue. Read and write performance is integral for any file system.

File slacks can be overwritten by normal use of the file system. In fact the cases where existing files are extended, deleted or moved are problematic, while creating new files or reading files are not a problem. A file slack file system would have to somehow mitigate this risk of losing data.

Part III

A FILE SLACK FILE SYSTEM IN PRACTISE

4 ON IMPLEMENTING A STEGANOGRAPHIC FILE SYSTEM FOR ANDROID

This chapter describes design and implementation decisions of a proof-of-concept file system, DHFS. The implementation is discussed in detail, which is followed by a security analysis.

4.1 SYSTEM OVERVIEW

This section gives an overview of the implementation. DHFS is a steganography file system that uses 8 independent compartments to construct the plausible deniability characteristic. Two compartments are intended for filling with midly compromising data to surrender. Another two for storing top secret data that stays confidential and another 4 as reserve. It is up to the user how many compartments are used for what purpose. Files stored in DHFS are stored in file slacks of the carrier file system. In order to access file slacks a custom driver is used, because standard drivers do not suffice.

The number-of-compartments constant (default 8) can be set in the source code in the range of positive integer value limits.

There are two major components of DHFS. One is the index that stores which file slack is part of which payload file. The index is compressed to avoid wasting space. The second is the file slack header. The file slacks stores data and a header that holds meta data about file slack length, integrity and encryption.

DHFS has a focus on security, so each compartment is encrypted with a password. The details of the encryption are discussed in detail, because even small mistakes can make a system insecure.

Examples of the index and the memory clarify how DHFS works. DHFS should stay hidden so the index is also stored in file slacks, which makes the system more complicated but hides its existence better. In the initialization first the index has to be found. Then individual files can be displayed. There is always the danger that the carrier file system overwrites file slack, which can lead to data loss. A mitigation for that can be redundancy, which DHFS does not (yet) take care of. However payload files are integrity protected with a cryptographic checksum, so if data is changed or damaged, the system will notice.

4.2 ACCESS METHODOLOGY

There are basically two methods how to get a block device to work with DHFS on Android. One is actually using a block device supplied by the system, located at `/dev/block/`. The second is to use an image file of a block device e.g. stored on the SD card, which can then be opened as a file. In both cases the file system has to be read and understood by DHFS, which integrates a FAT32 driver in order to do so.

Using the system block device is the method DHFS is intended for. The FAT32 implementation of Android is not able to access file slacks. If the system driver is used the data in the file slack cannot be accessed, so data there is well hidden. However there are many barriers that one has to overcome to directly use a block device circumventing the volume daemon `vold` which responsible for handling the SD card. The volume daemon is responsible for mounting the SD card on start up of the system and also for mounting and unmounting it if the SD card access is passed on to a PC over the smart phone's USB port. One of the barriers to access the SD card are the file system permissions. The block device is managed by Android and an app or the user is not permitted to directly access it. Thus the user needs root permissions, which can be done by "rooting" it. Various tools depending on the phone and installed Android version can help with that. Some organizations may forbid that their phones are rooted. This can be detected. However in Asia (3-4%)and Russia (~1%) it is common to use a non-malicously-rooted smart phone, because it is needed to add features that the Android API does not provide [15]. Some testing was conducted to explore this method on a Samsung Galaxy S5 with Android 4.4.2. The block device of the SD card is located at `/dev/block/vold/179:65`.

The volume daemon of the Android emulator only mounts an SD card image once at start up. Passing on the access via USB is not intended on the emulator. Testing needs to be done with a physical device.

The block device [7] is named with *"major device number":"minor device number"*. The major device number is selects which device is selected for input and output operations of the device driver. The minor device number is a parameter for the device driver, its use depends completely on the device driver. The manual of the device driver should state the meaning of the minor device number.
Only setting the block device file to public readable (permission mask 777) is not enough to access the file. All the directories in the path also have to be accessible. So with root permission those directories were also set to public readable (permission mask 777). Still the block device can not be accessed. In order to avoid access from the volume daemon and other processes all mount points involving the SD card were removed as seen in Listing 4.

Listing 4: Unmounting all mount points concerning the SD card on Android 4.4.2 with an Samsung Galaxy S5

```
C:\Users\user\AppData\Local\Android\sdk\platform-tools\adb.exe -
    shell
shell@klte:/ $ mount   # lists all active mount points
...
/data/knox/sdcard /mnt/shell/knox-emulated sdcardfs rw,nosuid,
    nodev,relatime,uid=1000,gid=1000,derive=none 0 0
/data/privatemode /mnt/shell/privatemode sdcardfs rw,nosuid,nodev
    ,relatime,uid=1000,gid=1000,derive=none 0 0
/data/media /mnt/shell/emulated sdcardfs rw,nosuid,nodev,relatime
    ,uid=1023,gid=1023,derive=legacy,reserved=20MB 0 0
/dev/block/vold/179:65 /mnt/media_rw/extSdCard vfat rw,dirsync,
    nosuid,nodev,noexec,noatime,nodiratime,uid=1023,gid=1023,
fmask=0007,dmask=0007,allow_utime=0020,codepage=cp437,iocharset=
    iso8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
```

```
/dev/block/vold/179:65 /mnt/secure/asec vfat rw,dirsync,nosuid,
    nodev,noexec,noatime,nodiratime,uid=1023,gid=1023,fmask=0
007,dmask=0007,allow_utime=0020,codepage=cp437,iocharset=iso
    8859-1,shortname=mixed,utf8,errors=remount-ro 0 0
/mnt/media_rw/extSdCard /storage/extSdCard sdcardfs rw,nosuid,
    nodev,relatime,uid=1023,gid=1023,derive=unified 0 0
shell@klte:/ $su
shell@klte:/ $umount /data/knox/sdcard
shell@klte:/ $umount /data/privatemode
shell@klte:/ $umount /data/media
shell@klte:/ $umount /dev/block/vold/179:65
shell@klte:/ $umount /mnt/media_rw/extSdCard
```

These steps are not yet enough to make the block device available to an test app. When the app tries to read the block device an error: " `Error 20 : A component of path is not a directory.`" appears in the Android logging tool (logcat). The log messages do not indicate a denied permission of SELinux, however it is a likely explanation for this behavior. This method still needs more exploration, since it offers desirable qualities e.g. no need for an easy to access and visible image file (better repudiability). Using a block device would not be leave a visible image file and also make it harder to make to view and change file slacks. The payload data is well hidden and traces can be removed to a satisfactory extend.

As result the simpler method of using an image file is used. For this proof-of-concept the visible image file is accepted. This method allows any user or application to inspect the image file including the file slacks, which is undesired. In addition this image is a significant trace of data being hidden.

4.3 FAT32 DRIVER CHANGES

Android offers a developer tool called "JOBB" [16] that allows to create Android application package (aka APK) expansions in the Opaque Binary Blob (OBB) format. These OBB files can provide additional files like graphics, sounds, videos to an Android application in a separate APK file.
This JOBB tool internally uses a Java library that represents a full implementation of a FAT32 driver without external dependencies. This is a publicly available open source driver [10] licensed under LGPL. Based on this driver, access to a block device or image file can be achieved (as `java.io.RandomAccessFile`). Still this driver is not able to access file slacks. Accessing file slacks is a new use case for this file system driver. However DHFS requires access to file slacks to fulfill its purpose. Hence the driver must be adapted in order to be able to allow access to file slacks which other drivers rightfully do not permit.
The driver contains two interesting packages. The first package (`de.waldheinz.fs.fat`) contains the FAT32 classes that actually implement the core of the driver, which represents the index chain management,

file allocation table, allocation strategy, file metadata format, short file names, long file names, directories and some data types.
The second package (`de.waldheinz.fs`) essentially contains a more abstract view on the FAT32 file system. It has an interface that any file system could implement that is provided to the library user. As a result the layered architecture is reflected and by using this generic interface the file system could be exchanged for another one without changing the interface.

The following paragraphs contain technical details of how the FAT32 driver was changed. Figure 11 shows the interlacing of FAT32 driver classes for accessing files. The driver works on a block device which is a low-level view to a storage medium e.g. an SD Card. Several classes on top of that create the high-level abstraction of files. The original FAT32 driver uses the methods `read` and `write` that do not allow file slack access. The methods `readExtended(...)` and `writeExtended(...)` were added, that *do* allow access to the file slack.

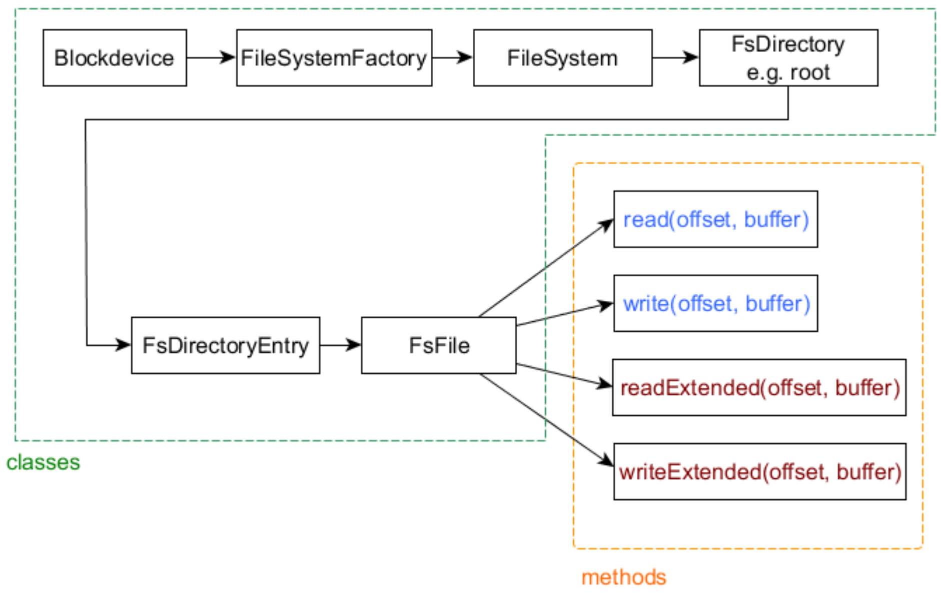


Figure 11: Block diagram of the FAT32 classes for reading and writing.

The main change for the access to file slacks lies within the driver implementation package (`de.waldheinz.fs.fat`) and there in the functions that contain

- `read(long offset, ByteBuffer dest)`
- `write(long offset, ByteBuffer srcBuf)`

to a file located in the class `FatFile`. Exchangeable to these two functions, two new functions

- `readExtended(long offset, ByteBuffer dest)`
- `writeExtended(long offset, ByteBuffer srcBuf)`

are created, see Listing 5.

Listing 5: Implementation of the methods readExtended and writeExtended

```
public void readExtended(long offset, ByteBuffer dest) throws
    IOException {
checkValid();
final int len = dest.remaining();
if (len == 0) return;

double numOfClusters = Math.ceil( (double)getLength()/(double)
    chain.getClusterSize() );
long maxSize = (long)numOfClusters * (long)chain.getClusterSize()
    ;
//This calculation includes the file slack as a new maximum read
//length it allows the last cluster to be read completely

    // no real EOF check
        if(offset+len > maxSize){
                throw new EOFException();
        }
//no update for the timestamp

chain.readData(offset, dest);
}

public void writeExtended(long offset, ByteBuffer srcBuf) throws
    ReadOnlyException, IOException {
checkWritable();
// no update of timestamps
final long lastByte = offset + srcBuf.remaining();
double numOfClusters = Math.ceil( (double)getLength()/(double)
    chain.getClusterSize() );
long maxSize = (long)numOfClusters * (long)chain.getClusterSize()
    ;
//This calculation includes the file slack as a new maximum read
//length it allows the last cluster to be read/written completely

        if (lastByte > maxSize) {
                throw new EOFException();
                //setLength(lastByte); // omitted
        }

chain.writeData(offset, srcBuf);
}
```

The second parameter (ByteBuffer) represents a buffer to which the data can be read in or be written to. It contains the file content. The offset parameter describes at which offset of the `dest` or `srcBuf` buffers shall begin to be written to or read from. The length of that content is determined by the `remaining()` function of the `ByteBuffer` class. That means the length of the content is determined by the remaining space of the buffer.
Hence the length is configured when the buffer is created with a specific size. That size must be calculated accordingly to the amount of content data for reading or writing.

Reading beyond the End of File (EOF) is not possible, an `EOFException` would be triggered by the `read` and `write` functions.
The function `read(long offset, ByteBuffer dest)` was copied and some aspects were changed. The determination if the `destination` buffer is big enough can stay the same. Updating the access timestamp depending on if the device is writable can be left out. Since writing the access time of a file is a hint that the file was indeed accessed. That is undesirable for the purpose of hiding secret data. In the next step the offset and the remaining length of the buffer are compared with the file's size stored in the `FatDirectoryEntry` that contains also the timestamps and some file attributes. This is indeed the critical part. This check is changed to allow not only to read up until the file size is reached, but until the cluster border is reached. Then the file slack can also be read. The new check needs to know the cluster size. The FAT32 file system reads that information from the MBR's boot parameter block when the file system is initialized. Fortunately the cluster size is also stored in the class *ClusterChain* and this class allows access so there is no need to change any more code. With that cluster size the total number of clusters used by the current file can be calculated by rounding up to the next cluster as shown in Equation 4 in Section 3.2.5. Afterwards the new maximum size is calculated, which the whole file occupies on the medium. This space is partly file content and partly file slack see Equation 5. This new maximum size is then substituted for the file size that was mentioned beforehand.

It is important to know that internally the driver can only read or write full clusters. E.g. an append for one Byte needs to read a full cluster, append one Byte and then write a full cluster back.

Then the function `chain.readData(offset, dest)` is called that looks up the file's cluster chain entries and copies the file content into the buffer. This function reads the file *cluster-wise*, since a cluster is the smallest addressable unit in a file system.
The function for writing, `writeExtended(long offset, ByteBuffer srcBuf)`, is built and changed in a similar way to the read function. However, before the actual writing the `write` function checks if the block device is writable. This is also useful in the `writeExtended` function. The update of the modification and access timestamp are not wanted in the new version and are skipped. Then again the *NumberOfClusters* and the new *MaximumSize* are calculated as described in Equation 4 and Equation 5. Then there is a check that compares the offset and the remaining buffer length to the file size. If the file size is exceeded, an append operation is recognized and the new file size is set. This needs to be changed since the file size must not change when writing to the file slack. So this check now issues a `EOFException` if the *MaximumSize* is exceeded. This has an twofold effect, for one that a write beyond EOF is now possible and second that the file size is not changed. Hence this function should not be used to extend a file, but only to write to the file slack. In the last step the file chain function `writeData` writes the buffer with the file content and the slack as a whole to the memory. A new class that handles these parameters and offers a more friendly interface to read or write only the file slack would be the best way to enable DHFS access to file slacks.
Now that the two functions `readExtended` and `writeExtended` are cre-

ated, they need to be accessed somehow. Either the file system specific internal interface is used or the generic file system independent interface is used. The architect of this driver clearly intended the generic interface because it is one abstraction layer above the specific interface. Therefore the two access functions are added accordingly to the public generic interface. The specific `FatFile` class implements the generic `FsFile` interface. The naming convention is that the specific implementation classes start with the prefix `Fat` and the generic interfaces that will be used for the steganography tool start with `Fs` (file system).

4.4 DESIGN RATIONALE

This section covers design decisions and motivation for the created file system prototype DHFS. Major elements are explained and reasons why they are needed are given.

4.4.1 *Index Table*

Many payload files can not be stored in one continuous chunk because they do not fit in one single file slack. File slacks range from several Bytes up to about 32 KByte, depending on the cluster size. That means these files have to be split up in fragments and the fragments are stored in several carrier files as the carriers are available. When the payload file is read the fragments have to be merged again. For merging the order of the fragments has to be stored, otherwise they could be merged in the wrong order. This storing is done in a clear and systematic way in the index file. The design of the index table is explained in Section 4.7.1.

4.4.2 *Index Entries*

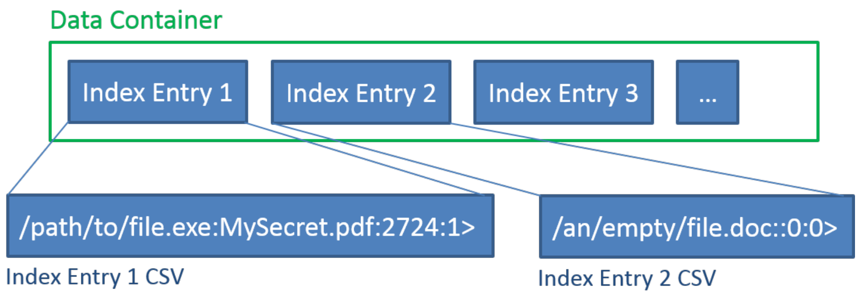


Figure 12: A compartment index stored in a data container with example data.

One entry of the index table consists of exactly four poperties, see Figure 12. It has to identify a carrier file uniquely. Fortunately, this does not need an extra ID variable, because the path of the carrier file already uniquely identifies it. So this path is stored first in the index

entry.
The second part is the name of the payload file. It is possible to extend this to a payload *file path*, but the proof-of-concept implementation DHFS is restricted to payload files. It does not allow folders and subfolders.
As a third part the payload file length is stored. The motivation for this can be found in Section 4.4.6.
The last part is a sequence number to store which part of the payload file is stored in which carrier file. The payload file parts have to be joined together in the exact same order as it was before the split-up into several fragments. This sequence number allows to save this order. The correct order matters here, in contrast to the compartment index where the order is irrelevant for the functionality.
An index entry needs to be stored in a binary format in order to store it. It is the simplest way to process a list of comma separated values (CSV) for an index entry. So each of the four parameters are printed in a string, separated with a separation character and finalized with a special character.
The comma character "," is a bad choice since it could already appear in one of the parameters, most likely in the file paths. There are several characters "?", "<", ">", "|", "\", "/", "*", """ and ":" which are reserved in FAT long file names. So a file name can not contain them. This makes them better separator characters than the comma. So the separation character is the colon character ":" and the finalization character is the "greater than" character ">". This is a limitation on FAT. Other file systems may have different character limitations. In UNIX systems paths can be escaped, so the only forbidden characters are the null character and the backslash. However sometimes command line shells introduce D DHFSmore reserved characters. Most UNIX systems are POSIX compliant. POSIX fully compliant file names [26] contain only characters "A-Z", "a-z", "0-9", ".", "_" and "-". So almost all special characters that are problematic, are excluded. DHFS will process fully POSIX compliant file names correctly. It will also handle FAT and NTFS file names correctly that allow several special characters more. In fact only the characters that are problematic are the chosen characters "colon" and "greater than". In NTFS the limitations are as well the backslash and the null character and further forbids """, "*", ":", "<", ">", "?", "\", "/" and "|". In general NTFS allows e.g. Unicode or UTF-16 character sets.
A CSV is easy to generate and simple to parse back in. This also implies that neither of the parameter can contain a comma, which is anyway not possible for numeric values. Carrier file paths and payload file names must not contain the coma character ",". It is uncommon to use the comma in a file name or path, but possible with FAT32 long files names. For FAT 8.3 file names (also known as short file name) the comma is not permitted, so there is no problem with short file names in both carrier and payload files.
For writing the CSV an encoding has to be chosen. This encoding is UTF-8 in DHFS, which is a wide spread standard encoding. UTF-8 is backward compatible with ASCII encoding.

4.4.3 *Checksum*

If payload data is read, the DHFS file system driver needs to be able to validate it. The read process finds data in the file slack. It might be all zeros (empty), random data (written by the initialization), data that is left over from the carrier file system or actual encrypted payload data. In order to identify what data is at hand, DHFS calculates and prepends a checksum of the payload data in this file slack. Hence the data is decrypted with the password the user entered. For resulting data the prepended checksum is read and another checksum of the payload data is calculated. If the checksums match, the payload data is valid and can be further processed. If the checksums do not match, the data is invalid. This could be the case because the password was wrong and the cipher text was decrypted into some invalid random data. The checksum could be wrong also because there was random data or other data in the file slack in the first place. If a wrong password was entered the user or inspector can not know if the password was wrong or if there is no data there at all.
The chosen checksums is discussed in Section 4.4.4 and Section 4.5.8.

4.4.4 *CRC32 Checksum*

The Cyclic Redundancy Check with 32 Bit (CRC32) [37] is an error detection code. There are several versions which differ in the resulting hash value (checksum). In case of DHFS the checksum has 32 Bit and its purpose is to detect errors after storage (or transmission).
CRC32 is based on a binary polynomial division without considering the carry. The processing is a repeated blockwise division of the long binary data. The resulting checksum is the remainder of that division. CRC32 an be efficiently implemented in hardware because XOR can be used instead of subtraction. CRC32 is mainly for detecting errors, but it is also able to correct D DHFSBit errors. DHFS uses it for an integrity check only. CRC32 can *not* always guarantee to recognize if the data is not valid. Actually it is easy to manipulate the data in a way so that CRC does not notice a change due to the systematic algorithm. Cryptographic hash functions do give this guarantee e.g. MD5 or SHA1. This guarantee is desirable for DHFS, but MD5 hash values need 128 Bit and SHA1 hash values are even 160 Bit long. As discussed in Section 4.4.3 file slack is sparsely available and a tradeoff has to be made between quality of the checksum and investing space in the slack header, for DHFS in favor of CRC32.
A cryptographic integrity check is done at payload file level additionally which is discussed in Section 4.5.8.

4.4.5 *Index Marker*

As discussed in Section 4.4.1 the DHFS index is stored in the file slack just the same as the payload data. During the read process, see Section 4.4.3, the DHFS driver needs to decide if the found data is data

of the index, actual payload data or neither. As a result there has to be some kind of marker in the header, if the current data is index or payload data.
This information can be stored in a single bit in the header. Due to the header alignment the POC implementation uses a magic number to store this information. For development it is easier to invest a whole Byte instead of using one Bit. Then Byte alignment allows nice debugging of the implementation. This magic number is finally stored encrypted so there are no security issues expected. The header layout is explained in Section 4.6.

4.4.6 *End of Payload File (EEOF)*

The use of EOF produces file slack. Ironically this system on top of carrier files, uses of a marker for the end of payload, EEOF, produces slack as well, file-slack-slack. Theoretically another system on top of this could produce file-slack-slack-slack... recursively until the disk space is 100% utilized.

Similar to carrier files that can be smaller than the available cluster size, payload file can be smaller than the available slack of a carrier. This is handled by FAT32 with EOF as mentioned in Section 4.3. DHFS has to solve the same issue. EOF suggests by its name that there is an end marker at the end of every file. But in fact there is no magic number at the end of the file, because then this magic number could not be stored within any file. It would be mistakenly recognized as the end of the file. The magic number would have to be escaped with an escape symbol. This is a inconvenient approach.
It is much better to store the length of the file as a meta information in the file header. Then any data can be stored. Since the maximum file slack size is 32 KByte the payload data length of one file slack can not be greater than 32 KByte. The header data and its data types are described in Section 4.6.

4.4.7 *Compartments*

A compartment is a container of payload files. The whole payload file system (DHFS) contains multiple compartments to generate the plausible deniability property. The compartment number is fixed to eight. So the user can plausibly explain the existence of all compartments. He can not change the number. Each compartment is encrypted with its own password, see Section 4.5.1 and Section 4.5. So a compartment holding random data can not be distinguished from a compartment with secret data, without the correct password.
The plausible deniability allows the user to give away one or two compartments that hold mildly compromising data. An inspector could force the user to give up those two compartments. This explains why the tool was installed and what it was used for. The inspector can not determine whether or not the rest of the compartments are unused or still contain more data. So the user can claim to have given away all passwords and it can not be proven otherwise. Since some data was given up it is plausible and believable to have found all data, while the user can still have important secrets in the rest of the compartments.

4.4.8 *Index Compartment Fragmentation*

Each compartment has its own index. If this index does not fit into a single file slack, multiple file slacks have to be used. A compartment index can be fragmented, this is a likely case since all files allocated to that compartment store one entry in the index. The general goal of having many small files for generating many file slacks, also generated many entries in the index. One compartment can easily have D DHFSthousand entries or more.
The internal order of the entries in the index does not matter. So there is no need for an additional sequence number. The compartment number, see Section 4.4.9, is enough to reconstruct the index of a compartment. In order to find all compartment index data, all file slacks have to be checked in an *exhaustive search*.

4.4.9 *Compartment Numbers*

Each compartment has to store an Identification (ID) number so when index data is found, it can be connected with the right compartment. If there were no compartment ID it would be impossible to decide which index data corresponds to which compartment.

4.4.10 *Compression*

The index table stores which carrier files holds which payload file. Not only the payload data is stored in file slack, also the index tables are stored in file slacks. Slack space is sparsely available. In order to minimize the file system overhead the index table should be compressed. The index table is stored as Comma Separated Values (CSV) and can therefore be compressed effectively similar to normal text. Also there are many entries that have the same structure.
Consider the following example as proof of the necessity of compression for DHFS: Assume there are 20 000 carrier files. That means 20 000 carrier paths have to be stored, no matter if any are used or not. Those are approximately equally distributed over eight compartments, resulting in 2 500 paths per compartment. Now assume an average path will have 45 characters (also about 45 Byte). That sums up to 112 500 Byte (about 110 KByte) per compartment. With compression this could be reduced by approximately one third.
Since the use of this file system encourages the use of many small files, the compression is obviously useful in place. The more carrier files the more efficient the compression will get. A typical set-up can easily have 100 000 files or more.
Any text compression algorithm could be used here. "GZip" is a standard compression tool that is described in Section 4.4.10.2.

4.4.10.1 *Index Compression*

The index table entries stored in carrier files are compressed with the Deflate algorithm. The payload data is not compressed, because it

could already be compressed and then no further advantages can be gained. For Index data there is a gurantee that it is not compressed yet. The index format is known and is promising to allow good compression results. There are two ways of implementing the compression:
Variant A: compress the whole index table and then distribute the fragments to carrier files.
Variant B: distribute index entries first and then compress the fragments.

In **variant A** the compression can achieve better effectiveness since many similar entries can be compressed better than fewer. But then the binary fragments would have to be reconstructed in the correct order so the original index table could be decompressed again. This would result in a need for storing the order of the fragments with an additional index carrier header parameter e.g. fragment number. This fragment number could then provide the order information. Another disadvantage would be that if one index carrier file is overwritten or somehow damaged the complete compartment index is invalid.

In **variant B** with distributing the index entries first and then compressing only the fragments, the compression will be less effective than in variant A. Particularly for very small file slack sizes the compression could be completely ineffective, so a lower bound for index carrier sizes should be introduced, which is determined in Section 4.4.10.3. The minimum of approximately three index entries have to fit in one index carrier, so the compression will actually produce shorter output. Assuming the recommendation of media larger than 32 GB is followed, the block size will be 32 KByte, so a statistical average of 16 KByte file slack size can be expected. In this optimal situation a lower bound of three index entries ($3 \cdot 66\,\text{Byte}$) is insignificant. So files that do not offer the lower bound of slack space would be disqualified as index carriers, but could still be used as payload carriers.
The most important advantage of variant B is that there is no need for the index carriers to be ordered, so no fragment number is needed. If one index carrier should be overwritten or destroyed, only this one block is invalid and not the whole index.

At this point robustness is more important than space efficiency, so DHFS will implement *variant B*, with distributing index entries first and then compressing the fragments instead of the other way around.

4.4.10.2 *GZip Format*

The GZip [53] lossless compression program uses the "DEFLATE" compression algorithm and is part of the GNU Project. It is publicly available under GNU GPLv3 and is also part of a standard Java library in the package `java.util.zip`. The class *Gzip* uses this library to provide convenient methods to compress and decompress data.

The GZip binary format starts with a magic number (`0x1F8B`), a version number and a timestamp as header, more optional parameters are allowed. Then the body contains the compressed binary data. In the end is a footer containing a CRC32 checksum to validate the integrity of the binary data and also the length of the original data. The total of the GZip header and footer take at least 12 Byte.
Due to this header and footer a very short input could result in a larger output. Space is sparse for DHFS and the resulting space has to be calculated to decide how many carrier files are needed. In order to minimize the overhead, the header and footer are not used and only the body remains. The implementation can now directly use the DEFLATE algorithm provided by `java.util.zip.Deflater` and `java.util.zip.Inflater` implemented in the class `DeflateCompression`.
Compression with GZip including the header and footer of 1 Byte results into an output of 21 Byte, so 20 Byte of overhead are added. Omitting the GZip header and footer reduces the output length to nine Byte, so only 8 Byte of overhead remain due to a minimum output length of the deflate algorithm. So there is still a minimum overhead, but smaller than if GZip is used.

4.4.10.3 *Deflate Break Even Point*

So the question arises from which input length will the compression actually result in a shorter output than input. Generally that depends strongly on the input entropy. In the case of "lorem ipsum" the break even point was determined at 66 characters using the DEFLATE algorithm without the GZip headers. At 66 characters input to the compression also 66 characters output are produced. For less input characters the output is larger than the input. For more than 66 input characters shorter outputs than inputs are the outcome. Naturally the users want to be above that break even point to store data with less than the original size.
For index entries this break even point will be approximately two to three entries. One entry may have about 35 Byte. To be on the safe side the break even point for DHFS can be estimated to about three index entries. If more than three index entries are stored in one fragment the compression gives an advantage. This is a moderate requirement. The index compression can be expected to be effective.

4.5 ENCRYPTION

This section covers encryption-motivation, -challenges and -solutions for DHFS. The used encryption is explained and a motivation for the choices is given.

4.5.1 *Need for Encryption*

All the data stored in DHFS is stored in file slack. File slack is not directly accessible by the OS, but if the raw storage data of a bock

is viewed the data is visible. If nobody checks the file slack the data is not found, but if this data is inspected then the secrets would be discovered. A simple string search will find the data. To avoid only relying on the steganography component the used file slacks must be encrypted. Hence an inspector would only see encrypted data (cipher text) which is not distinguishable from random data. So if the steganography fails, there is still a strong encryption in place that protects the secrets from an inspector. Choosing a strong encryption algorithm and all needed parameters is a difficult task and can quickly result in a system that looks secure to the layman, but can be trivial to break for a knowledgeable attacker. Among these parameters are the algorithm itself, the encryption mode, the initialization vector and many other small details. This is discussed in detail in Section 4.5.
Each compartment will have its own encryption key generated with a password. This means only secret key algorithms will be used in contrast to public key systems. Secret key systems have only one secret key that only the person that created the compartment knows. So each compartment is independent from the others. This is also important for the plausible deniability characteristic of DHFS.

4.5.2 *Individual File Encryption*

For file systems it is desirable to encrypt each file differently, so two identical files stored twice do not result in the same cipher text. This is because some data is recurring often e.g. the file header of a PDF file. If recurring passages in the cipher text appear, one could guess that this must be a PDF file header, thus a known plain text attack is created. In order to minimize the attack surface each file will be encrypted differently.
In 2015 and 2016 ransomeware became quite a trend among classical malware, due to its relatively low programming effort in comparison to e.g. banking tojans. If ransomeware somehow infects a systems, it silently starts to encrypt all accessible files and then asks the user to pay a certain amount of money to get the data back. Otherwise the data stays encrypted with a key the victim does not know.
So ransomware faces the same technical problem to implement secure encryption. For example Schmidt [42] describes that Teslacrypt 2.0 is malware that works with this concept. It uses one symmetric key to encrypt each file. Assume there are n=10 000 files then there are n keys to access all of the files. Now these n keys are again encrypted with a master key that is sent to (or predefined by) the extortioner and then deleted from the local disk. Telascrypt made a mistake with encryption of the n keys. So with some effort the n individual keys could be recovered without the master key. Teslacrypt 2.0 was broken. For the extortioner it would have been better not to store the n individual keys. If they had not stored them in the first place, they would not have been retrieved.
An alternative to using n different keys is to look closely at the encryption mode and use one initialization vector (IV) and add a sequence

number to that IV for each file. This is a standard technique for cryptography systems. Then each file is encrypted differently, so no recurring ciphers appear among files. The IV can be publicly known, without weakening the security of the encryption. No individual keys must be stored, so no keys can be recovered from storage. Only one master key is used.

Speific limitations depends on the encryption mode e.g. The IV used with CBC-mode needs to be protected against modification.

DHFS is using a different IV for each carrier file. Also Teslacrypt creators (black hats) noticed this flaw and evolved Teslacrypt to version 4.0, which was discontinued in May 2016. The producers of Teslacrypt published their master key with which all files can be decrypted.
Ransomware is still very popular and several implementations without major design flaws circulate.
There are also other examples of file system encryption, Ext4 encryption mechanics were already discussed in Section 1.6.1.

4.5.3 *Entropy for Random Number Generators*

Security related processes often require or use random numbers. The quality of these random numbers is often attacked, so they can be predicted which sometimes has an impact on security.
This first implementation decision is to use a Pseudo Random Number Generator (PRNG) or a True Random Number Generator (TRNG). PRNGs can inherently be predicted. Sometimes they are seeded with a start value to make this prediction more complicated. There are poor start values that should *not* be used e.g. the current time and date or CPU fan speed that may come to mind. Assumptions about those sources can be made. The used time stamp can be estimated to be in certain range which does not offer a enough entropy. Similarly the fan speed has a small range and it can even be manipulated by letting the system calculate difficult problems. Then the system will heat up and the fan is likely to run at its maximum or at least a very high speed. Mobile systems are usually cooled passively so this is not an option to begin with.
However, mobile devices e.g. smart phones, tablets etc. have a variety of sensors and receivers that can be used to gain entropy properly. This is an advantage over many other devices that do not have that many sources available. There are e.g. the gyroscope that measures the three dimensional acceleration, the WiFi, GSM, UMTS, LTE and Bluetooth receivers as well as light sensors, the camera and microphone that could potentially be a source of entropy.
The Android operating system offers entropy which is used for the Java TRNG `java.security.SecureRandom`. Additionally the user could be asked to draw a random pattern with his finger, if one does not want to rely on system entropy. The user can always be a source of entropy independently from the device.
In few cases there is a hardware module to generate entropy by measuring thermal noise of a semiconductor (diode), which is a very good source because it is hard to influence.

4.5.4 *Encrypt-then-MAC vs. MAC-then-Encrypt*

Namprempre and Bellare [3] suggest that simple encryption modes e.g. Cipher Block Chaining (CBC) do not include a Message Authentication Code (MAC) which allows to check integrity and authenticity additionally to the confidentiality and non-repudiation provided by the encryption itself. More advanced encryption modes like Galois/-Counter Mode (GCM) already automatically include a MAC.
A MAC can be created e.g. as a hash based MAC (HMAC) with a hash function. A HMAC uses both the secret key and the message to create a unique MAC using a cryptographic hash function e.g. MD5 (deprecated), SHA-1 (deprecated) or SHA-256 (strong). The correct name would then be HMAC-SHA256 for example.
There are basically three ways of assembling encrytion and MAC:

- **Encrypt-and-MAC**: Computes a MAC on the clear text, encrypts the clear text and appends the unencrypted MAC to the cipher text.

- **MAC-then-Encrypt**: Computes a MAC on the clear text, appends it to the data and finally encrypts the combination.

- **Encrpyt-then-MAC**: Encrypts the clear text, then computes the MAC of the cipher text and appends the MAC to the cipher text.

Encrypt-and-MAC provides no integrity on the cipher text because the MAC has the clear text as input. This is dangerous because chosen-cipher-text attacks are then possible, which is an unnecessary attack surface.
Still the integrity of the plain text can be verified. If the cipher scheme is malleable the cipher text could be changed. Malleable means that the cipher text can be changed so after decryption a related plain text is the result. Then after decryption the plain text is invalid, even though just slightly changed. Of course decryption errors can be exploited. The MAC might reveal information about the plain text. Also a repeated message will have the same MAC unless the MAC already includes random input.
MAC-then-Encrypt does not provide integrity on the cipher text. Without decryption there is no way of proving if the message is authentic or forged.
After decryption the plain text can be checked for integrity.
If the cipher scheme is malleable it might be possible to change the clear text and also change the MAC accordingly so that the message appears valid.
In this case the MAC is encrypted so it can not reveal any information about the clear text, which is good.
Encrypt-then-MAC provides a mean to check the integrity of the *cipher text*. As long as the MAC shared secret was not compromised it is possible to prove if the cipher text is indeed authentic or if it was forged.

If the cipher scheme became malleable, the MAC will show this immediately and no further processing takes place. Integrity of the plain text can still be verified.
The MAC is not giving away any of the plain text information since the output cipher appears to be random and the MAC does so too. The structure of the plain text is not reflected in the MAC, because it was created from the cipher in addition to the one-way function property of the MAC calculation.

To summarize this section, *Encrypt-then-MAC is the best choice*. Any modification on the cipher text can be detected and if so the message will be ignored. Invalid data is not even decrypted preventing any further attacks on the decryption implementation. Since the MAC has the cipher text (appears as random data) as input, the MAC itself can not be used to infer any information about the plain text. Encrypt-then-MAC provides the highest security of the three alternatives.
The choice between the three alternatives matters most in Client-Server architectures, where the message is transported on a public wire. For file systems the transport of data takes place over time instead of space, so forging cipher texts would require write access to the medium. Doing that unnoticed is even harder.
The Xor-Encrypt-Xor (XEX) -based tweaked-codebook mode with ciphertext stealing (XTS) by Rogerway [40] Encryption mode does not provide any integrity check of the ciphertext. XTS-mode is intended for and used in at least nine disk encryption products, indeed it is *only* NIST approved for this purpose, other usage is discouraged. XTS is fast and the the de-facto standard in Full Disk Encryption. This mode uses relatively small blocks (16 Byte) in contrast to other modes like CBC. XTS is a wide-block tweakable mode from a narrow-block tweakable mode. This is a complicated construct, in other words, hard to prove its correctness. XTS does not provide integrity protection mechanism e.g. MAC is missing. Sector data (file content) is by default not integrity protected. The most important weakness of the XTS-mode is that the integrity of the cipher text is not validated. Hence the cipher text can be manipulated and will be decrypted unnoticed unless the application on top of XTS implements an integrity check. XTS is vulnerable to chosen-ciphertext attacks, replay and randomization attacks as Ptacek [38] describe. An authentication each sector is too expensive. Rogerway [40] criticizes an unclear specification of the security goals and argues that three goals are limitedly reached and not formally proved: limited privacy, limited non-malleability and "some sort of key-dependent message-security at least something strong enough to allow the encryption of ones own key" [40].

4.5.5 *Advanced Encryption Standard*

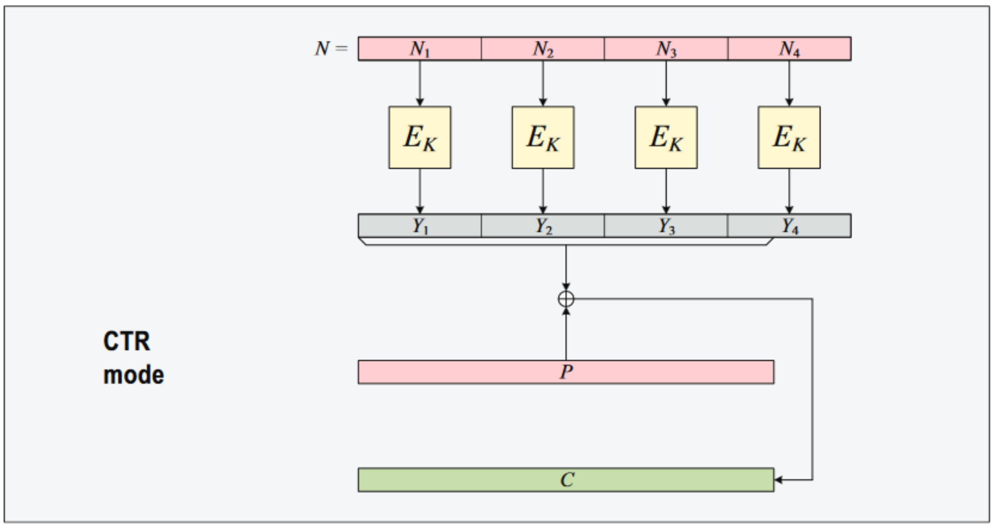
The Advanced Encryption Standard (AES) is deemed a strong secure encryption standard commonly used in many applications and pro-

tocols e.g. for wireless communication Wifi Protected Access II or in Transport Layer Security for secure communication over the World Wide Web.
AES is based on the Rijndael algorithm and is a symmetric block cipher that can process data blocks of 128 Bits. It can use cipher keys with lengths of 128, 192 or 256 Bit. The National Institute of Standards and Technology (NIST) of the United States of America standardized AES.[35]
The AES block cipher is the basis for the used encryption in DHFS.

4.5.6 *Counter Mode (CTR)*

CTR is defined by the NIST in [8]. CTR is an encryption mode that can be used with 128 Bit block ciphers. NIST requires the underlying block cipher to be approved, which means it must be AES with one of its three key lengths. The CTR mode is the simplest and therefore most elegant mode of all the confidentiality-only modes. It was first suggested by the famous Diffie and Hellman at the same time the other four modes Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB) and Output Feedback (OFB) were presented. "The simplicity, efficiency, and obvious correctness of CTR make it a mandatory member in any modern portfolio."[40] CTR can generate ciphertexts with an arbitrary length, so no padding is needed to fill up the current block. CTR mode generates essentially a stream cipher, which is quite an advantage for DHFS.
However, there is one downside of CTR. The used counter value key combination can *never* be reused. These values pass the hands of the encryption user which leaves plenty of opportunity to disobey this critical instruction. Reusing counter values which is in this mode the IV will lead to a catastrophic outcome for the system security. So the IV is defined to be a Number used Once (NONCE). A NONCE guarantees to be unique in the according cryptography context.
The second disadvantage of CTR is that it does not offer any authenticity, chosen-ciphertext attack security or non-malleability. The encryption user can not assume any of these properties and will almost certainly fail in doing so. In Section 4.5.4 are the solutions for authenticity and non-malleability discussed. The mitigation for chosen-ciphertext attacks is discussed in Section 4.5.2.
The encryption and the decryption algorithm are precisely the same, which is a remarkably pleasant property of CTR. Encryption can be fully parallelized. CTR has excellent performance (fastest of the five modes) and its provable security guarantees confidentiality similar to the other four modes.
The CTR mode takes as input a secret key K, a NONCE N and a plaintext P. It returns a ciphertext C as output. The underlying block cipher is E processes a fixed length NONCE part N_x to the intermediate variable Y_x with the same length. Y_x and the plaintext are combined with a logical XOR to arrive at the final result, the ciphertext C. Figure 13 shows a block diagram and a mathematical representation of the CTR

algorithm.



```
algorithm $\mathrm{CTR}_K^N(P)$                                        *CTR mode*
// $K \in \mathcal{K}$,  $N \in (\{0,1\}^n)^+$,  $P \in \{0,1\}^*$,  $|N| = \lceil |P|/n \rceil$
$m \leftarrow \lceil |P|/n \rceil$
$(N_1, \dots, N_m) \leftarrow N$
for $i \leftarrow 1$ to $m$ do $Y_i \leftarrow E_K(N_i)$
$Y \leftarrow \mathrm{MSB}_{|P|}(Y_1 \cdots Y_m)$
$C \leftarrow P \oplus Y$
return $C$
```

Figure 13: Block diagram and algorithmic description of CTR encryption mode.[40]

The *Galois/Counter Mode* (GCM) defined by NIST in [9] is a combination of the CTR mode and Carter-Wegman message authentication, in the style encrypt-then-MAC. GCM therefore already achieves Authenticated Encryption with Associated Data (AEAD) out of the box. For DHFS it is a unfavorable choice, because then the index table data is hard to include in the MAC. By including the index table data (payload file name and file size) payload file meta data can also be checked if they are forged or authentic. With the standard GCM only the payload file content could be checked.
Another alternative would be to use CBC mode in combination with an HMAC. Unfortunately CBC [40] can only produce ciphertexts that are a multiple of the block size e.g. 16 Byte. Thus the last block has to be filled up with padding e.g. PKCS5. DHFS has an arbitrary length of file slacks to fill, hence an arbitrary ciphertext length simplifies the read and write process.

4.5.7 *DHFS Cryptography System*

The Open Web Application Security Project (OWASP) Guide to Cryptography [13] states recommendations for the design of cryptography systems. So design decisions can be well made, respectively typical pitfalls can be avoided. Important topics are the choice of a proper encryption algorithm, key lengths, key storage and transmission of keys. DHFS does not store any keys, nor are keys transmitted, not even a master key. The choice for a strong block cipher fell to AES because it is a well known secure algorithm that is expected to stay secure in the foreseeable future. A key length of 128 Bit for AES is deemed secure. The longest (most secure) key length for AES is 256 Bit so this is used by DHFS. For the application in file systems that typically has one (or very few) users that require both read and write access symmetric keys are used.

The number of atoms contained in our galaxy is estimated with 2^{233}[41]. 2^{256} Bit offer a large search space.

4.5.7.1 *Key Derivation Function and Salt*

The user should be able to encrypt and decrypt his data by using a password. This password is processed into a 256 Bit long key for the data encryption. This is done by a Key Derivation Function (KDF). KDFs are mathematical one way functions, meaning it is easy to calculate the key from a given password, but it is extremely hard to do the reverse, to calculate the password from a given key. It can be attempted to calculate the outputs for all inputs. Such a table is called rainbow table, which can then be used to look up the password for a given key. The password - key space is designed to be enormous, so that this approach would represent a huge computational task that would take thousands of years with today's computers. However for some antiquated algorithms today's computers are fast enough e.g. MD5 was broken this way. If such a KDF takes not only the password as input but also a fixed value, called salt, per user or per application instance, then such a precomputed rainbow table would have to be computed again for each user or application instance. Thus the KDF is customized and a general rainbow table is useless. This represents a significant increase in the KDF's strength and the salt can still be public.

The salt does not need to be a true random number (TRN), a pseudo random, or even a sequence number serves the purpose. Essentially the higher the entropy the better. The TRN is best, because if one could assume a fixed start value and a sequence number for a low number of files then only very little of the large value range would be used. Computing e.g. "the first few hundred" rainbow tables might be feasible in future. A TRN uses the whole range avoiding this assumption of few reused salts. The salt has to be different for each use case, avoiding reuse even with another key provides additional computational security. Nevertheless a TRN is used in DHFS, implemented with the self seeding TRNG `java.security.SecureRandom`. The salt length is set to 128 Bit, so the possible range is $2^{128} = 3{,}4 \cdot 10^{38}$ possibilities which gives a probabilistic guarantee that there are enough different salt values. Precomputing a KDF for all salt-password combinations is infeasible.

The estimated lifespan of the universe is 2^{61} seconds. The number of atoms in the planet earth is about 2^{170} [41]

There are many KDFs, DHFS follows the recommendation of the Internet Engineering Task Force defined in [12] known as Password-Based Key Derivation Function 2 (PBKDF2). PBKDF2 is a pseudo random function that uses a password and a salt to create a key of a cer-

tain length. This is repeated for many iterations. The iteration count is also a parameter of PBKDF2 as well as the algorithm to be used. For DHFS the underlying algorithm is SHA256. The crypto library `javax.crypto.SecretKeyFactory` is used to create the PBKDF2. The iteration count is relatively high with 65536. A few thousand would also be enough, but the more iterations the better is the computational security. The iteration count was intended to be able to increase security post release. Choosing this value is a tradeoff, more iterations are more secure, but require more computing. The number of iterations is configurable so security can be adjusted for different devices. Weak CPU devices may choose a low number of iterations e.g. 2000. Powerful Server hardware may afford a much higher number of iterations, because a lot of computing power is available e.g. 100 000.
The relative high number of iterations was a performance problem for the test device due to the high number of repetitions. If the calculated key is cached and therefore only calculated once, so the iteration count can remain high. The KDF is used both for encryption and decryption as depicted in Figure 14 and Figure 15. At the moment the HMAC and the encryption are derived from the same password. Section 5.5 describes a way to use two separate keys to decouple the integrity check and the encryption.
An alternative PBKDF is Scrypt that requires not only computational power, but also costly memory. The large memory requirements are problematic for specially dedicated cryptography mining hardware such as ASICs but also for off-the-shelf graphic cards with. The Graphics Processing Unit has a high computation power but the connected bus system is in comparison quite slow (the bottleneck) and thus additional high memory requirements slow down the process significantly.

4.5.7.2 *Payload and Index File Encryption*

After the key is derived from the password as Section 4.5.7.1 describes it, the data is encrypted with that key. The data in this case is the current payload file e.g. `MySecret.pdf`. The encryption with AES-CTR is already described in Section 4.5.5 and Section 4.5.6 and also shown in Figure 14. One salt is needed per password. Since there is one password per compartment, one salt per compartment is enough. However the salt is stored in the header of each file due to a lack of a per-compartment storage. For the same reason the IV is also stored in the header of each file. The IV (counter value) for CTR mode needs to be a NONCE, which is a true random number in DHFS instead of an increasing sequence number. The CTR mode requires only that the counter value (and key combination) does not repeat itself. The range of the IV (128 Bit) is a large enough so a coincidental repeated IV is statistically unlikely. Both the salt and the IV can be publicly known without hurting the confidentiality of the encryption. There is a NONCE for each payload and index file fragment. The encryption of payload files includes the authentication and integrity check performed with HMAC. The encryption of DHFS index files does not

In CTR mode the IV (counter value) must be a NONCE. This is extremely important! If it is not a NONCE, the encryption is pointless.

include a HMAC, but most index data is still included in the HMAC for payload files.

4.5.8 *HMAC*

As mentioned in Section 4.5.4 the HMAC is a cryptographic integrity protection checksum. It is used to provide message authentication, in the sense that the user has knowledge of the secret key, but not more. It is created using the secret key and the message. For DHFS, the HMAC-SHA256 algorithm is used. Any change in the message will lead to a (drastic) change in the checksum. Therefore changes in the message can be discovered. A manipulated message is invalid and will not further be processed. There is one HMAC per payload file fragment. The message in this case is the payload fragment itself including the header data and payload file meta data in the index table. Particularly the payload file name (including the extension), the total payload file size, the part number and the payload fragment length are included in the message and hence integrity protected. This is important for various attack scenarios discussed in Section 4.10. The only parameter that is not secured with the HMAC is the carrier path. If that part is wrong the payload data will simply not be found.
For the implementation of the HMAC the Java library `javax.crypto.Mac` was used. If a payload file is stored the HMAC is generated and stored in the index table. Lastly when the compartment is closed, the index is written to the medium. When a compartment is opened, the unencrypted index is available again. Then if a payload file is opened, the according entries in the index are collected. The HMAC for each fragment is calculated from the metadata and the whole payload fragment. If it matches the stored HMAC no unallowed changes were made and the payload file is passed to the user. If the calculated and stored HMAC do not match then an unauthenticated modification was made. Therefore the fragment and consequently the whole payload file is invalid and the user is informed about this incident. Such a change could happen due to normal carrier file system operations e.g. partly or completely overwriting a file slack, but it could also be an attack on DHFS. In both cases the payload file is invalid.
The HMAC is stored in the index with a Base64 encoding so it can be guaranteed that the special characters for separating the index parameters are not included, which are the "colon" and the "greater than" characters.

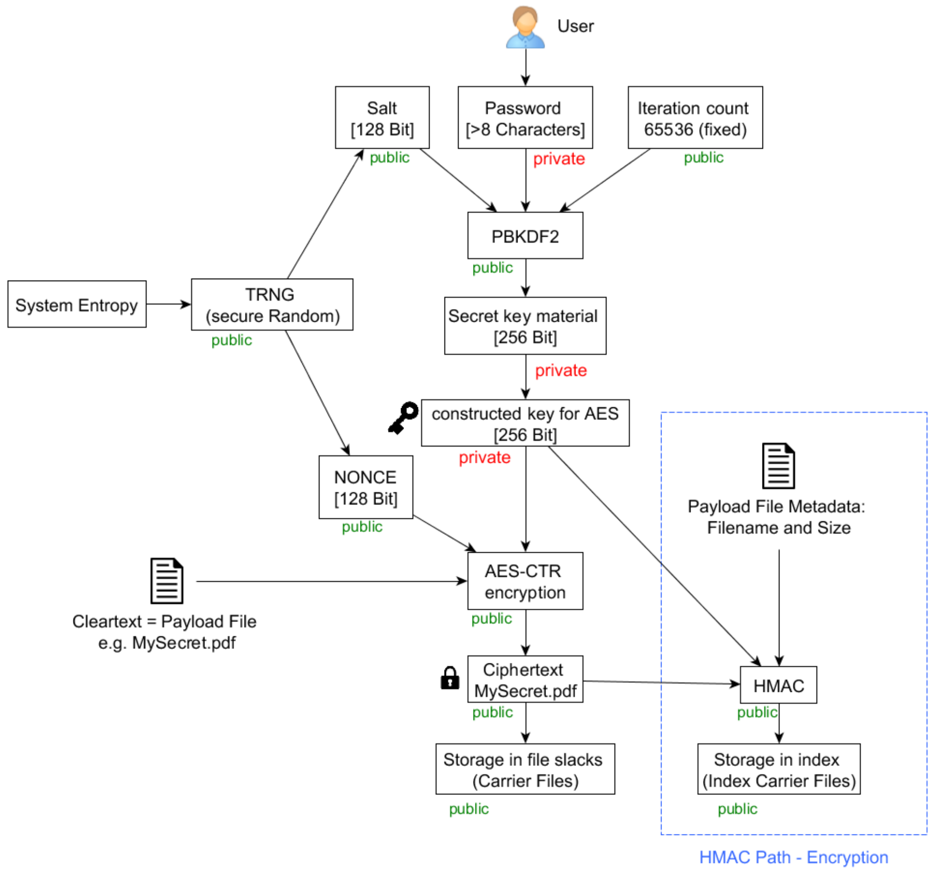

User
Salt [128 Bit]
public
Password [>8 Characters]
private
Iteration count 65536 (fixed)
public
PBKDF2
public
System Entropy
TRNG (secure Random)
public
Secret key material [256 Bit]
private
constructed key for AES [256 Bit]
private
NONCE [128 Bit]
public
Payload File Metadata: Filename and Size
AES-CTR encryption
public
Cleartext = Payload File e.g. MySecret.pdf
Ciphertext MySecret.pdf
public
HMAC
public
Storage in file slacks (Carrier Files)
public
Storage in index (Index Carrier Files)
public
HMAC Path - Encryption

Figure 14: DHFS encryption block diagram. The HMAC is only generated for payload files and not for DHFS index files.

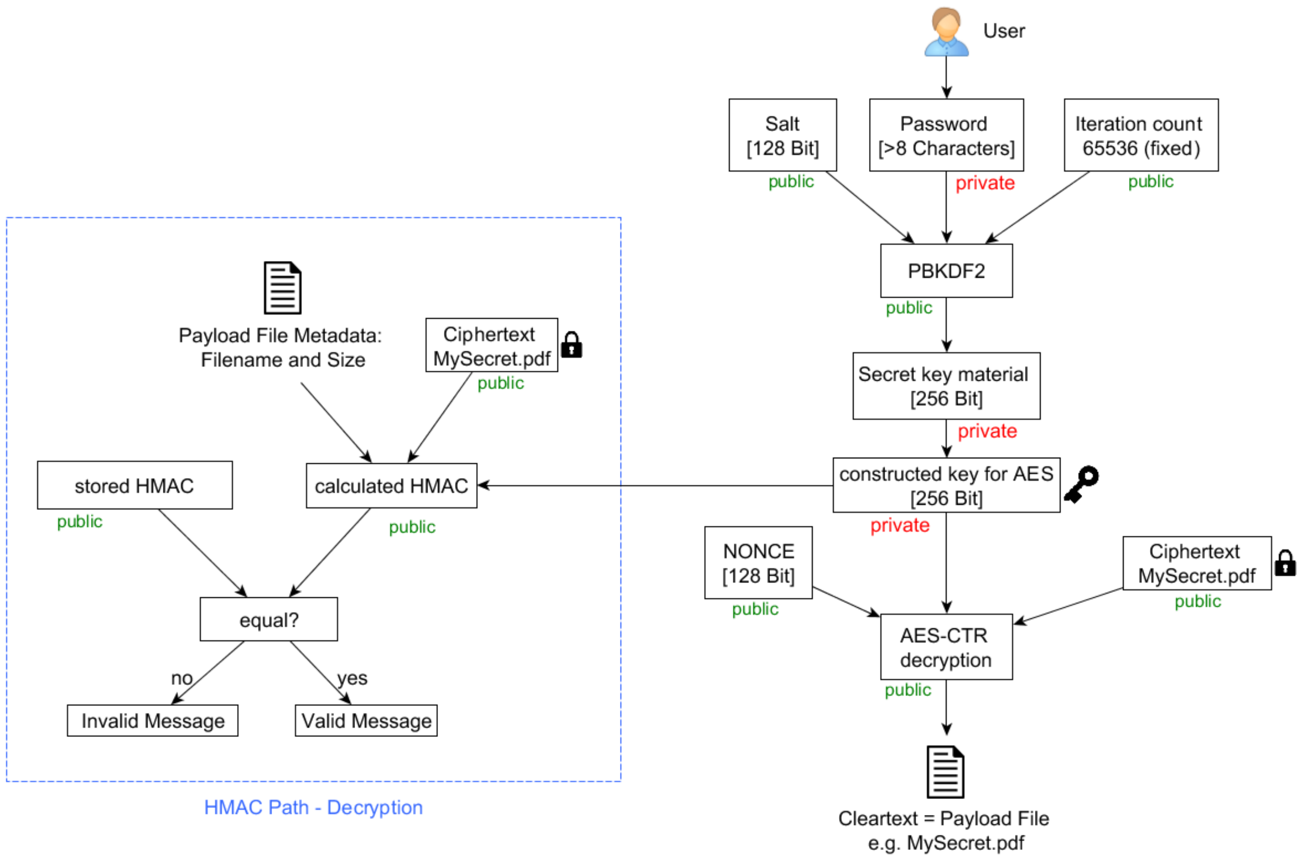


Figure 15: DHFS decryption block diagram. The HMAC is only generated and compared for payload files and not for DHFS index files.

4.5.9 *JDK Key Length Restriction*

Unfortunately the Java encryption library [6] comes with a default policy that only allows a key length up 128 Bit. If one wants to use key longer than that e.g. 256 Bit an `InvalidKeyException` with the comment "`Illegal key size or default parameters`" occurs. This is *not* a programming mistake, but a non-technical restriction (policy) of Java with default settings. The *Cipher* class permits only up to 128 Bit long key. This restriction can be removed by replacing the according policy file found at %JRE_HOME%\lib\security.
This limitation was implemented due to legal limitations in some countries on encryption strengths. Android does not impose such a policy that limits encryption key lengths.

4.6 memory layout and header data

Figure 16 reveals the order and composition of the file slack data format. The index bit is first, because before one can access payload data the index must be loaded. The CRC32 checksum identifies if valid data resides in the current file slack, otherwise further processing can be skipped.
Assume the data container holds index data. So the data container must be read into a compartment. Then there can only be one GZip fragment for index data. In the next step the index data can be decompressed and loaded into memory.
Otherwise the data container holds payload data. In this case the compartment number is irrelevant. The compressed data could be a fragment of a larger compressed file or it could be small enough to fit in one slack and have only one fragment. The index table holds that information. Accordingly fragments have to be concatenated in the correct order and can then be decompressed. If there is only one fragment no concatenation needs to take place.

The case where index data is enclosed in the data container is depicted in Figure 12. One data container will typically hold many serialized index entries as CSVs. One compartment index can and will likely need multiple data containers. That means multiple index carrier files. More detailed information on index entries can be found in Section 4.4.2. Figure 17 presents the case where the data container is filled with payload data. In this example file content of the payload file `MySecret.pdf` is stored. The payload data could use all the available space of the data container or just a part of it. The length of the container depends on the slack size. Self-evidently the size of the slack header has to be deducted from the slack space to arrive at the correct maximum net data container size. This overhead has to be kept in mind when fragmentation takes place. A 1000 Byte payload file needs net 1000 Byte container length. If that payload file is split in three fragments then the gross space need is 1000 Byte plus three times the header length (38 Byte), which results in 1114 Byte. The

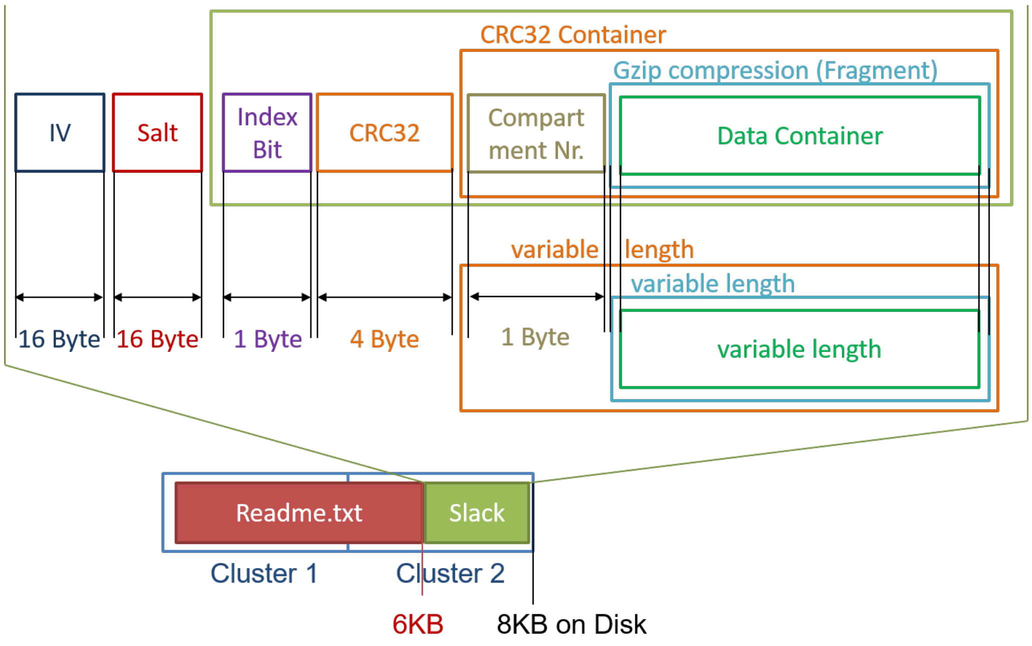


Figure 16: Overview of the memory layout and the slack header with header parameter sizes

needed space for the index is not included. Depending on the available carrier files, the fragment number will vary and therefore the additional overhead. The more fragments the more overhead. With a cluster size of 32 KByte the overhead of 38 Bytes becomes negligible in terms of wasting space.

Figure 17: A data container that stores payload data.

Figure 16 shows the header parameters in detail including header parameter sizes. The figure shows that first the IV is stored. This encryption parameter can be public just as the salt that is stored right after the IV. The salt and the IV are needed to decrypt the encrypted data that follows them. So only the two absolutely necessary parameters are stored unencrypted. The relatively long size with 16 Byte each is necessary to create a large enough range of values, so it is statistically unlikely that values would be repeated by the TRNG. The meaning and need of the IV and salt are discussed in Section 4.5
The index bit is actually one Byte long and contains the magic number `0x0A` (ASCII Line Feed). The approach with an encrypted magic number is simpler to implement, but one Bit would also be enough.
The CRC32 checksum is here the perfect trade-off between checksum length and integrity verification quality, see Section 4.4.4
The compartment number can be expected to be in the range of 0 to

255, which is $2^8 - 1$, so one Byte (8 Bit) is enough. The corner case of zero compartments is negligible. The system is not usable with zero compartments. The number of compartments is a constant, fixed to 8 and starts counting at 1 not 0. Future version of DHFS might increase this value.
Since the file slack varies, also the available data container length is variable. There is a minimum requirement for file slack sizes. It has to be able to store at least the size of the header including IV and salt plus one Byte of payload data. Otherwise it is too short and will be ignored by DHFS. An even larger minimum file slack length requirement might be useful because it might not be worth the processing effort to gain just one Byte of payload space.
The compression is only applied to index files because there the contained index data is well defined and can be proved to be compressed effectively. Payload data compression might result in more data if e.g. the data is already compressed.

4.7 DHFS INITIALIZATION

The assumptions mentioned earlier in Section 3.2 must hold (reasonably enough file slack). Essentially a decent amount of carrier files that have to offer at least some slack space. Then using DHFS for the first time, some initialization has to be done.
DHFS finds an amount of carrier files N. Let N e.g. be 8 000. These carrier files are distributed among the 8 compartments equally. So each compartment gets 1 000 carrier files. The file slack of each carrier file will vary, so the slack space distributions is statistically equal for large N. So each compartment has a similar capacity. After the initialization each compartment has a list of carrier files that are only associated with the current compartment. Each carrier file can only be associated to one compartment, otherwise conflicts would occur. This would be trivial if the indexes of the carrier would not also reside in the same carrier files. The index carriers can not be payload carriers.

4.7.1 *Index Table Example*

This section contains an example of an index table, see Table 4. It should clearly visualize the complex index structure.
An index table as described in Section 4.4.1 contains index entries. The individual index entries are shown in Section 4.4.2. Each index table describes one compartment only. There are allocated entries like the one in the first line. This entry states that the first part of the payload file `MySecret.pdf` is stored in the slack of the carrier file `/path/to/file.exe`. The whole payload file `MySecret.pdf` has 1 MByte which is stored as the number of Bytes. Assuming an average slack size of 2 KByte, the `MySecret.pdf` would need 512 fragments. The total size of the payload is stored redundantly for each fragment.

1 MByte = 1024 · 1024 Byte = 1048576 Byte

In contrast to the length of the payload fragment which is stored only once. The *payload fragment size* describes how much of the payload data is stored in the current file slack. The fragment size is very important. It states how much of the available data in the file slack must be processed. This can only be stored in the index because the file slack data, including the header, is still encrypted. Before decryption the whole file slack is read and therefore the length of the data has to be known.
The slack fragment size has two Bytes so a non-negative integer can be saved in an appropriate range. Two Bytes (16 Bit) allow to store numbers from including 0 up to including 65535, which is $2^{16} - 1$. Unfortunately in Java, primitive data types are stored in the two's compliment representation. So they always have a (almost) symmetric range of positive and negative numbers. The only exception is `char` which is too small in this case. In order to do the conversion from two's complement into unsigned binary numbers, the class `ByteConvertUtil` was created as part of DHFS. The fragment size corresponds to a 8 Byte `unsigned short` in C.

Table 4 is decrypted and loaded into memory. Compartment 1 is therefore opened. Other compartments may not be opened or empty (and therefore can not be opened).
Notice that there are no restrictions on the carrier file type or format. Also any payload file can be stored. There are no payload directories, only payload files without hierarchy are supported by DHFS. The various file lengths are stored in Byte. Each payload fragment has one row in the index table. The fragment length varies due to varying slack sizes and also according to the payload file length. The HMAC is a cryptographic integrity checksum that is described in Section 4.5.8. Empty compartments are still initialized and encrypted with a random password.
Table 5 represents an unopened compartment (password not known). It could contain random data or encrypted data. DHFS does not know, until the correct password is entered. Neither could an inspector know. Table 5 is an illustration that encrypted compartments can not be read and carrier files allocated to this compartment can not be recognized. Neither for the purpose of allocating new carrier files, nor for protecting existing carriers of overwriting, nor for an inspector to find out about secret payload data. Actually DHFS could not even create a table like Table 5 because needed parameters are still encrypted, the index entry separator for breaking the table rows is only available in the unencrypted version. After decryption, DHFS would not attempt to create an index table unless the correct password was provided.

Carrier path	Payload file name	Total payload file size	Part number	Payload fragment size	HMAC (Base64)
/path/to/file.exe	MySecret.pdf	1048576	1	412	bQk8Y2T4x5JhYm1B4ya+ro6tDe64phhnCEUtIaYQYZs=
/an/empty/file.doc					
/path/to/a/file.pdf	Illuminati.xls	1337	1	165	Z4g3MhW4sFg/3FRZSYbKHonKDyDIrGBBByE9oxq5tMY=
/another/empty/file.jpg					
/path/to/anotherfile.png	MySecret.pdf	1048576	2	283	hf4JS1Htp+IoR8Mzkpq+anrjQKF1DqdtW+fJFNunahM=
/path/to/a/Readme.txt	MySecret.pdf	1048576	3	42	Prl2R6d+CFm+fXhfrB+sAmfvEH5nJ5itQQDLkv3PCyo=
/emptyRootDirFile.mp3					
...	...	...	...	...	...

Table 4: Example of an unencrypted index table of a single compartment, e.g. compartment 1 with explanatory entries.

Carrier path	Payload file name	Total payload file size	Part number	Payload fragment size	HMAC (Base64)
tkHpCamCIlgomsVcKOqMFPewmYYJTYvOywNDBCd6v66XRV8agnHqTArwwu5hJOzZH78pUufIUfz95Bp					
vHxtreWcv4QWKDHuLhS7eCyf3sp4fZM7YjVVateS0hrVFiRJXDXjbycSRIHpTuhTm4PoxekVbcYM7					
lvr9qK7eNdilTdCcdYBk424WLUZCopOnSJBQZgxUuzrWkNs4ucDikBUFmysv6yj1BTU6AZei9exVY					
IS3Z6imTkc6tIiiiTzCt8ENuxJOLe70zyrDR7ROkDHHjwAW73lPgwu4L4LidXdeWQJyvJDiUZEn7vvUd					
KJ412zFfP9WpiCJgfOl8cGarUeOjQX13nYjf8bk4s97dH2LWoev1mnDmvXWkc5lfmPwxC2l2KeMNBhOnQ					
...					

Table 5: Fictional illustration of an encrypted index table of e.g. compartment 2. This table is not readable without decryption (with the correct password) first.

4.7.2 *Index Entry Distribution Algorithm*

Only complete index entries are written to index carriers. No index entry fragments are allowed. Each carrier is filled with index entries, which then are compressed. If the compressed data still fits into the carrier another index entry can be added and then the whole index entry list is compressed again and checked again. Finally the point is reached where the compressed data is too large, then the previous result that barely fitted is written to the index carrier.

4.7.3 *Carrier File and Index Carrier File Allocation*

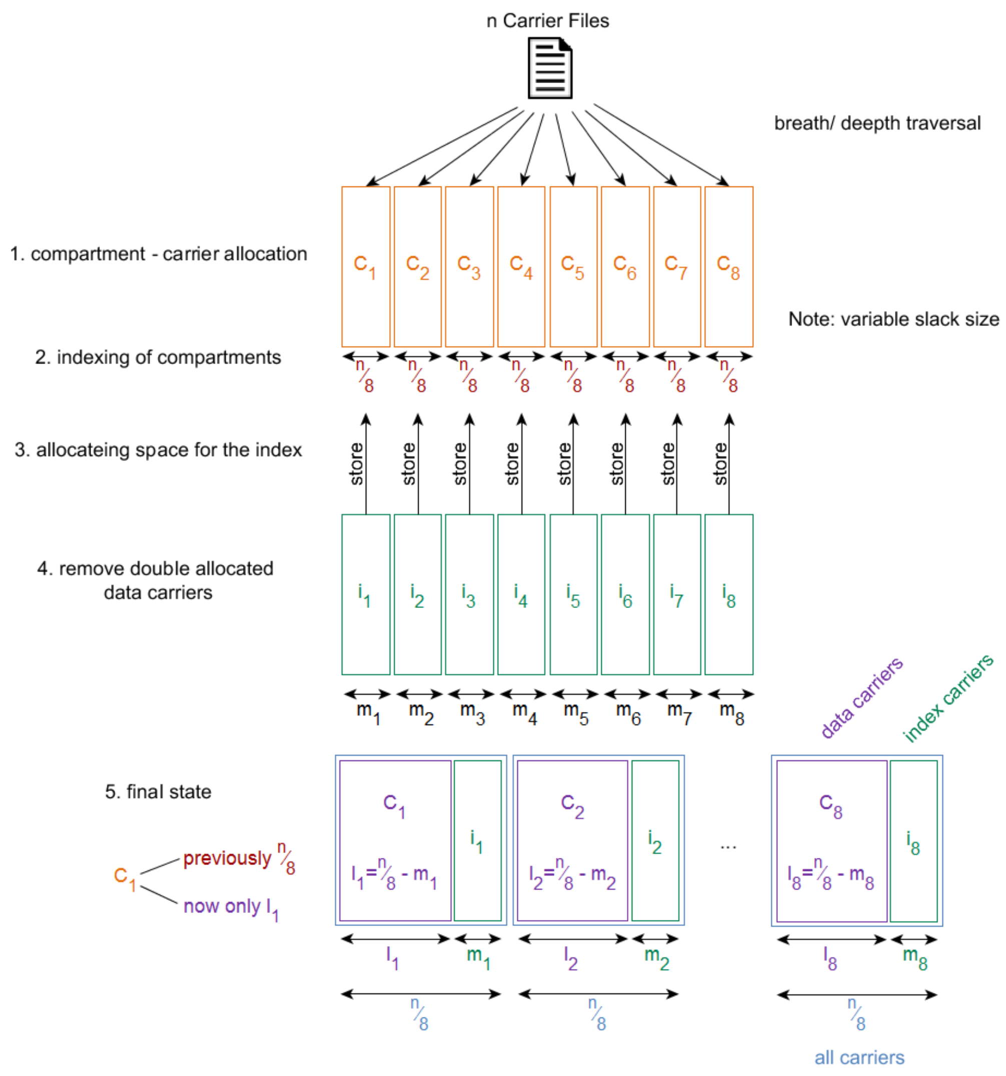

n Carrier Files
breath/ deepth traversal
1. compartment - carrier allocation
C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8
Note: variable slack size
2. indexing of compartments
$n/8$
3. allocateing space for the index
store
4. remove double allocated data carriers
i_1 i_2 i_3 i_4 i_5 i_6 i_7 i_8
m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8
data carriers
index carriers
5. final state
C_1 previously $n/8$
now only l_1
$l_1 = n/8 - m_1$
$l_2 = n/8 - m_2$
$l_8 = n/8 - m_8$
...
all carriers

Figure 18: Illustration of data carrier and index carrier allocation process

Figure 18 visualizes the the process of the carrier allocation. In the first step of the initialization DHFS finds a medium with *n* carrier files. They are identified by their path and file name. So DHFS browses the carrier file system (directory tree) in a *deepth traversal first* manner and collects all carrier files in a list. This could also be done in a breath traversal first. This would have the consequence the neighboring files (same directory) would be neighbors in the traversal file list. The compartments are allocated in order of that list, so depending on the traversal strategy neighboring files would be in the same compartment or not. This could be important for an redundancy concept, see Section 5.10.

In the second step the carrier file list has to be distributed among the compartments. This is a typical distribution problem. So each compartment gets the same amount of carrier files, distributed in a *round robin* manner.

Round Robin is a scheduling algorithm that gives away resources, here carrier files, one each and repeats until all files are distributed.

Then all compartment indices are filled with the carrier file paths. Now the indices have to be stored in carrier files. So the size of each compartment index is determined and written into a list. With this list the method `findExtractAllIndexCarriers(...)` in the `Index` class selects carrier files out of all N carrier files to hold index data. They are selected randomly, which is described in the next paragraph. The selection works compartment wise. For example if compartment one needs 500 Byte for its (uncompressed) index then the selection of carriers continues until at least 500 Byte worth of carrier files are found. Of course the selected carrier files have to be removed from the index they are listed in. Therefore all compartment indices are searched for the carrier files and finally removed. This removal of a few index carrier files reduces the already determined size of the indices for a few index entries. This reduction of the index size can cause the number of needed index carrier files to drop by a small amount. This will particularly happen if the offered file slack by the selected index carrier files is low. In that case there are e.g. one or two index carrier files that are not needed. They are then not filled with any data and will therefore not be used at all. They are orphaned files because they were already removed from the indices. As a result they are neither carrying index data nor can they be filled with payload data. Such orphaned files can be reintegrated by doing a balancing of DHFS which takes care of unseen files, see Section 5.8.

DHFS assumes that there is enough slack space available to store all the indices. This is the defacto lower space bound that DHFS needs. This lower bound depends strongly on the number of files and the amount of offered slack space.

The random selection must not allow duplicates, so a carrier file can only be chosen once. For development a PRNG (`java.util.Random.Random(long seed)`) with a fixed seed (one) is used. As a result always the same sequence is generated and the DHFS driver can be repeatedly tested under the same conditions. For operational use a true random number generator is better, because then the carrier files for the index are different for each initialization. As a True Random Number Generator `java.security.SecureRandom.SecureRandom()` is used. `SecureRandom()` is self-seeding, so a proper initial value for the Random Number Generator is chosen automatically each time in a secure way.

If a randomly selected carrier file does not offer any slack space, it is disqualified and not added to the index carriers.

For strongly inhomogeneous file slack sizes the distribution of slack space among the compartments would vary a lot. In terms of slack space this distribution does not give a fairness guarantee, only the number of carrier files is evenly distributed. Starvation is possible, but an unlikely case if the number of files is large, which is also required for slack size reasons. In mobile environments this is not a problem.

The carriers are spread over the compartments in the same order they are found. The combination of deepth traversal first and round robin will likely create a mixed pattern of compartments. For example if there is a carrier file system directory containing eight carrier files, each carrier file would be associated with a different compartment. If instead of round robin, all the N carrier files are split in eight blocks, the eight carrier files will likely be associated with the same compartment. These patterns of which carrier files attends to which compartment can be important if data is overwritten or redundancy is introduced into DHFS.

The calculated space must be adjusted by another factor, when it is split into multiple index fragments. Each fragment resides in a carrier file that needs a header. If the total index size of a compartment is 500 Byte and it is stored in three index carrier files the total space needed on the medium is 500 Byte plus three times the header size. This is considered by DHFS. Depending on the available file slack the fragments are created "on the fly". The number of fragments is not predetermined. Such a predetermination would be tricky, because the assumed needed space rises with each fragment. The initially determined number of fragments would be insufficient. Then an additional carrier would have to be allocated. In the worst case this could happen multiple times.

4.7.4 *Storing Indices into Index Carrier Files*

Now all the index carrier files are prepared. In the next step the indices have to be written into the index carrier files. This is done by the method `writeIndexOfCompartment(...)` in the `Index` class for each compartment.

An according number of index carrier files that provide enough space for all the index entries is provided already as Section 4.7.3 describes. The method `writeIndexOfCompartment(...)` can rely on having enough space. So each carrier file receives one serialized index entry after another. The serialized index entries (CSVs) are provided by the index compartments. Only complete CSV entries are stored in an index carrier file. Therefore the order in which the carriers are read is irrelevant. Then the index carrier path is traversed to get to the actual index carrier file and then the CSV index entries are written.

4.8 DHFS ACCESS

This section is about the process of reading a DHFS that was already correctly written and is undamaged. Here is described how a stored payload file can be recovered.

The first task of DHFS is to read the index one compartment at a time.

When the index of a compartment is read it goes to the opened state. Then the content of the opened compartment is displayed and the stored files can be accessed. the compartments are closed again.

4.8.1 *Coincidential Fuzzy Testing*

Due to the plausible deniability characteristic of DHFS, the file system itself does not know if a compartment is filled with data or if the compartment is only random data. DHFS will have both. A compartment that is not a compartment at all, but only random data is essentially fuzzing input for DHFS.
Fuzzy testing is a well known and extremely successful testing approach to find bugs and security issues. A program is given random data as input, which leads in many cases to application crashes. Implementation often make conscious or unconscious assumptions about the input data e.g. assume a counter only to have positive values. If a counter then has negative values the program is not prepared to handle this. That is a cause why strange errors may appear.
Any file slack could be random data. This is a consequence of plausible deniability. This fuzzy testing aspect was initially not anticipated and it uncovered several implementation problems that were then solved. For example huge fragment size that were longer than the file slack triggering various buffer overflows. The solution was then a sanity check. The fragment size must be between one and the slack size of the current carrier file.
Fuzzy testing also appears in DHFS if data is decrypted with a wrong password. Then the resulting plain text is random data salad as mentioned in Section 4.8.2.

4.8.2 *Index Search*

The user "opens" a compartment x and provides a password for this compartment. In order to read the index, an *exhaustive search* is carried out. DHFS gathers a list of all files on the file system (e.g. SD card). Then every file slack is attempted to be decrypted with the compartment user provided password, the IV and the salt that are read from the slack header. If and only if all three parameters are correct, the decryption will provide the correct unencrypted data. If any of those parameters are wrong, the decryption will produce randomw data.
In this case DHFS is trying to process random data, which is coincidentally the same as fuzzy testing.
After the decryption the plaintext (or trash data) is tried to be processed. First the index Bit (magic number) is checked, see also Figure 16. If it is the index magic number it is further processed, if not, DHFS continues with the next carrier. This is an early knock out criterion to avoid unnecessary processing in an already resource intensive search.
Next is an error-detection check. The CRC32 checksum is read and

then compared with a newly created checksum from the data. Accidental data changes can be recognized. If the checksums match, DHFS assumes valid data. This is a check to detect accidental errors in the data. CRC32 can easily be forged by a malicious user, an HMAC not, see Section 4.5.8. This way the index Bit is not integrity protected, but still confidentiality protected.
In the following step the read compartment number is compared to the user requested compartment number. If they match processing continues. It could happen that the user is using the same password for two independend compartments, then the decryption could result in the valid plaintext data for two compartments. Of course it is recommended not to reuse a password, but DHFS would handle it correctly.
Thereafter the compressed data fragment is uncompressed. Then a CSV list of index entries emerges, that can be parsed and loaded into RAM. The order in which the those index fragment are recovered does not matter. Each index fragment only holds complete index entries. The ordering of index entries in the index table can be arbitrary. When all files in the file system are processed once, the whole index of compartment x is found. If one index file is damaged, it is not read and the index entries of that index fragment are missing. Nonetheless the valid index fragments are read correctly, and the index still works for the available entries. Whether or not a index fragment is missing can not be determined by DHFS. Some additional mechanism would be needed for that.

4.8.3 *Payload File Search*

After the index was successfully read, the user gets a compartment listing, a list of all payload files that are stored in the opened compartment x.
For this DHFS iterates through the compartment index entries, checking for inhabited index entries and adds them to a result set without duplicates. Each fragment has one index entry. One file could have many fragments, but is only be listed once. This result list is then displayed to the user.

4.8.4 *Payload File Retrieval*

Assume the user has already opened a compartment x and got a list of available payload files. Then a payload file can be selected for retrieval from that list.
DHFS now gets the payload file name and the compartment number x of the file to be opened. The index of compartment x is iterated for all occurrences of the given payload file name. Each occurrence represents one fragment of the payload file and is collected in a fragment list. This list is sorted first by ascending part numbers, so the fragments can already be processed in the right order. There has to be at least one entry per payload file.

The fragment list is worked through as follows. The first carrier file's file slack is read. Before anything else happens with the data, the cryptographic HMAC checksum is checked. The metadata from the index, the still encrypted file slack and the secret key are used to generate a HMAC. If it does not match with the stored HMAC from the index, this fragment is invalid. The user gets an error message. If the HMACs match, valid data is at hand.
Then the password, IV and salt are used to decrypt the rest of the data. This is exactly the same process as with index files. Then the header is next. First the index Bit (magic number) must match "P" for Payload, otherwise this fragment is not seen as payload and skipped. Then the CRC32 checksums and the compartment numbers must match, also exactly as for index files. There is no decompression because payload data is not compressed.

CRC32 checks for payload could be omitted, because the integrity is already checked.

Finally the first payload file fragment is extracted. This data fragment is stored in a byte buffer. The next iteration of the fragment list returns an additional fragment, which is appended to the buffer. Since the fragment list is sorted the order of the fragments is correct. The fragment list is worked through until all fragments are found. Then the complete payload file byte buffer can be handed back to the user. The user now has a copy of the payload file in memory, not as a temporary file because this file would need protection.

4.8.5 *Adding a Payload File*

This section describes the process of adding a payload file to a compartment x. The compartment x has to be opened already, as described in Section 4.8.2.
The user provides DHFS with a payload file to be stored in the compartment x. If the payload file is too large, DHFS will fill all available space of the compartment and then finally return with an error.

On error cases DHFS might store left over data. A balancing step could neutralize this behavior. Anyway the written data would be protected just like normal data.

The procedure of adding a payload file is as follows. The whole payload file is read into a byte array. The payload file has to fit into the memory, otherwise DHFS can not handle it. Then the index of compartment x is iterated until an unused carrier is found. This is a linear allocation strategy. However, the order of the table depends on the sequence in which it is loaded. After reinitialization or balancing the ordering could be different, particular if the number or locations of carrier files change.
The free index entry carrier file is checked for how much space is available in its file slack. At least the IV, salt and a header plus 1 Byte payload data have to fit into that slack to be further considered.
Assuming the slack is large enough, the available net slack space is filled with the first part of the payload file. DHFS stores the current offset where the not-yet-written payload byte begins. Then the index entry is updated and the file slack including header, IV and salt are written. The first fragment is finished. After that this procedure is repeated as often as needed. This way the fragments utilize the maximum slack space of the current carrier. However, the file's last frag-

ment is special. It is unlikely that the last fragment also fits exactly in the last file slack. In general it will be shorter than the space that is available. DHFS only writes the needed amount of Bytes to the slack. Whatever data is stored in the file slack remains there. This will most likely be random data from the initialization. It could also be left over data from a previously stored payload file. This would still be encrypted and then partly or completely overwritten. Fragments of encrypted data look like random data and can anyway not be recovered.
After the last fragment was written, the job is done and DHFS returns to the user. The process of removing an added payload file is discussed in Section 5.6.

4.8.6 *Example of DHFS State in Memory*

Section 4.8 describes the procedures that DHFS uses to implement file operations e.g. index carrier allocation, payload carrier allocation, payload fragmentation etc. These operations generate memory artifacts in the file slack of carrier files.
Figure 19 illustrates a possible arrangement of different elements that could be stored by DHFS. This figure continues with example data used in the example index table Table 4.
Figure 19 shows three fragments of `MySecret.pdf`. The first two use all the available slack space, the third and last does not as it is typical. The unused space is file-slack slack. Also notice that the ordering in which the fragments are stored in the carriers is arbitrary.
The `Illuminati.xls` has just one fragment and does not even fill the Slack2. DHFS only allows a maximum of one fragment per carrier, even if there is still space left.
Figure 19 also exhibits nicely that both payload data and index data are stored in file slacks. This lack of separation of index and payload data is not typical in file system and leads to implementation challenges. Payload data and index could potentially overwrite each other, without separated address spaces. However DHFS handles this nicely.

4.9 ACCESSING INTERNAL AND EXTERNAL MEMORY IN ANDROID

This section describes how to program with the Android `File` interface for file systems. It is very similar to disk-based file systems used on other platforms. Memory location choices and basic file operations are discussed from the view of an app. Particularly how to save a file to internal or external memory, free space and file deletion are discussed.

The Android `File` API was designed for writing data in a linear start-to-finish manner e.g. images or network traffic. Reading and writing files is a basic task of an application (app). This API is re-

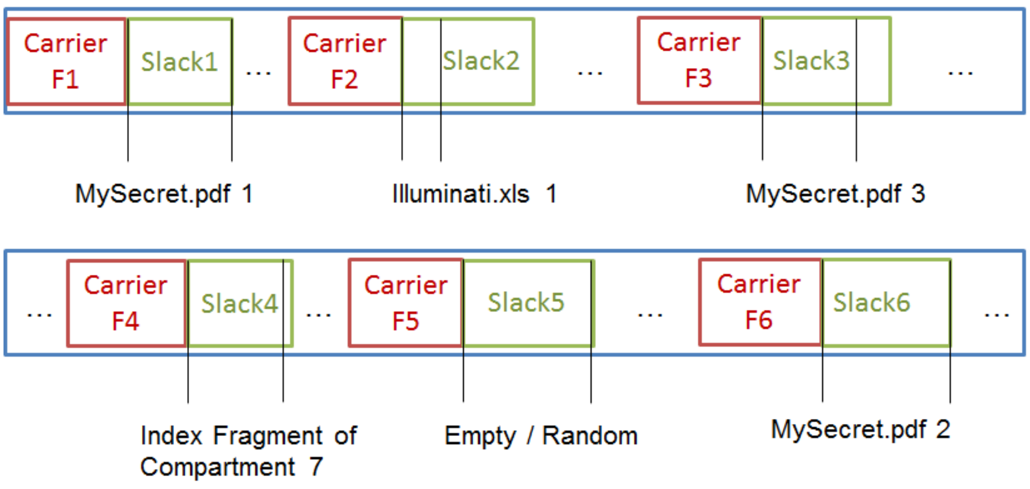


Figure 19: Example of memory state of a carrier file system with DHFS.

lated to the standard Java library `java.io`.

4.9.1 *Internal and External Storage*

Android distinguishes between internal and external storage. This is true for all Android devices. The naming stems from early times of Android devices, when most devices still offered one internal non-volatile, non-removable memory and additionally a removable (therefore external) medium typically a micro SD card. Other devices omit a removable memory, but then still have one partition used as internal memory and another partition as external memory. So there are always two types of memory available.
Internal storage ...

1. is always available,
2. ensures that files are by default only accessible for the app that created,
3. garantees that all files associated with an app are deleted, when the app is uninstalled.

Internal Storage encapsulates app data and thus supports the app to keep data private. Neither other apps nor the user is able to access those files. There are only very few exceptions e.g. if the device is rooted the user gains access also to the app internal data.
External storage ...

1. is not always available. Depending on the device, external memory can be physically removed. It could also be accessed via USB and is therefore unmounted. Then only the remote device can access it,

2. is publicly readable (until Android 4.4). Other apps or the user may delete, modify or read the data. The data is not exclusively under control of the creating app,

3. may leave back data after the app is uninstalled. The system only deletes data from the external memory if the app places it in a predefined location given by `getExternalFilesDir()`.

External storage is best if there is no need to keep data private or the app wants to provide the data to other apps or to allow the user to access it with another computer. Also size considerations play a role. Depending on the internal and external memory sizes, users might prefer to use rather external memory than internal. Typically older and low cost devices were built with little internal memory.
Apps are by default installed in internal memory e.g. `/data/data/com.myapp.appname/` , but an app can specify in the app's manifest file the attribute `android:installLocation` which allows "internal only", "auto" or "prefer external". In the latter case a large .apk file can be stored in the external memory. Other data will still be stored in the internal memory.
Internal memory is using an underlying file system driver of the extended family, most commonly Ext4, which is device specific. In contrast to external storage, which is pre-formated with FAT32. External storage could also be reformated to an extended file system or another file system that is supported by the Android kernel.

4.9.2 *Permissions for External Storage*

File permissions for internal memory are implicitly given (for a very limited per-app space) for an apps, while external storage permission have to be specifically declared.
Access has to be declared in the app's manifest, which can be found in the root of the app with the exact name `"AndroidManifest.xml"`. As seen in Listing 6 the write permission has to be defined if needed. Note that write permissions implicitly include read permissions. Generally for access an declaration has to be made. Depending on the API level this permission can be left out if and only if `getExternalFilesDir()` are used. For Android 4.4 (API level 19) this is true. For Android 4.3 (API level 18) and lower the declaration is needed anyway. There is also a corresponding read permission, which should be used if read access is sufficient for an app.

Listing 6: Android write permission for external storage

```
<manifest ...>
        <uses-permission android:name="android.permission.WRITE_
            EXTERNAL_STORAGE" />
        ...
</manifest>
```

There is no permission to request access to a block device.

4.9.3 *File Access to Internal Storage*

An app has two predefined file locations (directories), one for caching files returned by `getCacheDir()` and one for regular persistent files returned by `getFilesDir()`. Those are per default /data/data/com.myapp.appname/cache and /data/data/com.myapp.appname/files. Other directories can generally not be accessed from the app. In Listing 7 two standard ways for opening a file are depicted. The first version returns a `File` type, which allows several meta operations e.g. to check for write permissions. For reading from or writing to the file additional classes are needed. In the second version the return type `FileOutputStream` directly allows file access. Furthermore the context mode is specified which version one does not specify. If `MODE_PRIVATE` is used the file created by the app are only accessible by the app. An app programmer can allow read and write permissions for other apps, however this is rather considered a security flaw than a communication mechanism. Another app does not have the right to browse directories, for access it would have to know the correct path already. The correct path is a fixed base directory `/data/data/` and the app package including folder e.g. `com.myapp.appname/files` and the correct filename e.g. hello.txt.

If system memory runs low, the system may decide to delete cache files without warning!

Listing 7: Examples for opening a file in internal memory

```
Version 1:
File file = new File(context.getFilesDir(), filename);

Version 2:
FileOutputStream outputStream = openFileOutput(filename, Context.
    MODE_PRIVATE);
outputStream.write("Hello World!");
```

4.9.4 *File Access to External Storage*

As mentioned in Section 4.9.1 the external storage can be removed or used via a PC and could therefore be unmounted. An app should always verify whether or not the external memory is used otherwise. If the function `getExternalStorageState()` returns `MEDIA_MOUNTED`, there are no obsticals for further use.
The external storage is modifiable by the user and other apps, still there are two categories of files, public files and private files. Public files are freely available to the user as well as other apps. If an app is uninstalled public files stay at the user's disposal e.g. pictures or downloaded files.
Private files are associated with an app and do not offer any use without the app. These files are deleted by the system when the app is uninstalled, given that they can be found in the app's external private directory. E.g. temporary media files or additional downloads are private files. Technically these files are not protected from outside manipulation.

Android offers the method `getExternalStoragePublicDirectory()` for getting the appropriate directory for public files. Moreover one can pass a parameter such as `DIRECTORY_MUSIC` or `DIRECTORY_PICTURES` (and more) in order to get the collected data according to its purpose. The function `getExternalFilesDir()` delivers the correct path for private app data, which of course is for each app a different one. Unfortunately some apps hardcode paths that are not universally adaptable. This may lead to inconsistencies and is bad programming style.

4.9.5 *Free Space and File Deletion*

If it is known how much space a file allocation takes, it makes sense to check if the required space is available. The methods `getFreeSpace()` and `getTotalSpace()` help in this regard. These methods allow in advance checking for space without causing an `IOException` due to too little free space. Also filling up memory beyond a certain threshold may be desirable for performance and the possibility of makeing use of swap space. The system does not guarantee that all of the free space is indeed be used up. If the storage is used less than 90% writing is very likely still safe.
If the file size is not known in advance an `IOException` should be caught by the app by default or better additionally.
File deletion is straight forward by using the delete method of the file interface, respectively the `deleteFile()` method of the app's context.

4.10 SECURITY ANALYSIS

The first barrier for an attacker is that the data is hidden. So the attacker has to find out that there is data hidden in the file slacks. This is only an *obscurity* measure. It can be expected that (only) simple superficial searches for data can be fooled that do not check the file slacks. The data in the file slacks appears random, but is ciphertext. DHFS has this obscurity component to hinder attackers in their reconnaissance, so they would not know where to start attacking DHFS.

The second barrier is a classic encryption solution. Even if the attacker finds out DHFS is used and has detailed knowledge of the system, the encryption has to be broken first before any plaintext data can be retrieved. However the attacker could read and write to the medium and thus has a chance to interfere with the system. There are generally three security goals: Confidentiality, Integrity and Availability. Confidentiality is protected by strong encryption, Availability can not be protected if the attacker has physical access to the device. If the attacker securely deletes all data on the device or destroys it, all data is gone. The DHFS encryption system protects the confidentiality of all the payload and index data including metadata (header and index). The only exceptions are the IV and the salt, which need to be available for decryption. They can be by definition public knowledge. Section 5.2 suggests a obscurity measure to hide these parame-

ter from inspectors without knowledge about DHFS. Integrity is protected with an HMAC (not CRC). The following paragraphs describe what would happen if data of DHFS was changed by an attacker.
First the manipulation of the index is discussed and then manipulation of payload header data and index header data.

4.10.1 *Index Parameter Manipulations*

The index data is encrypted, but with knowledge of the data format the parameters would be changed. All possible parameters are listed below with the according outcome of an integrity attack.
The term "HMAC protected" applies so several parameters. It means the parameter is part of the cryptographic checksum, HMAC. If HMAC protected data were changed or spoofed DHFS will detect this and give an error message instead of accepting spoofed data.

- spoofing the *carrier path*
  The carrier path could be changed undetected. It is not protected. This is intentional, because a redundancy solution still has to be found, which may allow multiple copies of an fragment. If it were maliciously changed, the original data fragment would simply not be found. Instead of the original a forged fragment could be used from a different location. However for forging an encrypted fragment, the secret key would have to be known, otherwise the decryption would produces random data. Random data or even spoofed data would be detected due to the HMAC checksum. As a result changing the carrier path does not allow forged fragments, only duplicated fragment would be valid.

- spoofing *payload file name* and/or file extension
  HMAC protected, only the valid parameter will be processed

- spoofing the *total payload file size*
  HMAC protected, only the valid parameter will be processed

- spoofing the *part number*
  HMAC protected, only the valid parameter will be processed

- spoofing the *ordering of payload fragments*
  HMAC protected, only the valid parameter will be processed

- spoofing/changing the HMAC
  The HMAC is stored in the index table, which is stored in file slack (fragments) with a header. The header contains a CRC check, so if the HMAC was accidentally changed the CRC checksum for the according index fragment would be invalid. This index fragment with all its entries would not be processed. The other valid fragments would still work independently.
  The attacker could attempt to manipulate the CRC checksum.

This is generally possible due to the lack of cryptographic guarantees of the CRC checksum. However the CRC checksum of the index fragment that contains the changed HMAC is also encrypted with the secret key. A changed HMAC would result in an invalid fragment. Even if the attacker would generate an adjusted CRC checksum he could only have it as plaintext, but the header is encrypted. In order to create the correct ciphertext CRC the secret key is needed.
An intentional spoof of the HMAC is not possible. The secret key is part of the HMAC input data, so the secrete key would have to be known to successfully forge the HMAC. If the secret key is known to the attacker he has complete access anyway.

4.10.2 *Header Parameter Manipulations*

Another attack surface are the header parameters. If the attacker has knowledge about the header format, he could attempt to change parameter to influence DHFS. The complete fragment including all parameters are input the cryptographic HMAC checksum for payload fragments, hence if any *payload data or payload header parameters* were changed the DHFS would *detect* it and the according payload file fragment would be invalid.
The HMAC is stored in the index table. There is *no* additional *HMAC* for the *index fragments*, yet. However, used index entries are already HMAC protected, only unused entries and index header are not HMAC checked. An attacker could attempt to change the index fragments with all of its parameters and data. The consequences of a change of each *index data and index header* modification are listed here: The index header and data are encrypted, so changing it in a meaningful way requires the knowledge of the secrete key. Nevertheless a change of the encrypted data is possible, spoofing is very hard or not possible. Changing the ciphertext at a specific location will result into a change of the plaintext at about the same location. The change would not propagate though the complete ciphertext like it would in Cipher Block Chaining mode, where each block depends on the former block. A change would stay within the block boundaries, which is the IV (or key) length (16 Byte in DHFS). Due to the diffusion property of the underlying block cipher a change of one Bit of the input would result into a drastic change of the output block. Therefore a change of only one or several bits is hard to reach.
For the sake of this analysis it is assumed that single parameters or data could be changed:

- changing the index Bit (currently magic number)
  If the attacker changed the magic index number it is very likely that it will be processed to an arbitrary number. Most likely magic number would be nether of the two valid magic numbers. Then the fragment would be ignored as random data.
  If it were luckily recognized as payload it would be uncovered during the access by the an invalid HMAC for payload frag-

ments. If it were luckily recognized as index, due to a rare collision in the encryption, the fragment is processed as a index fragment. This requires the to have the correct format.

- changing the CRC32 checksum
  A change in this value would indicate that the fragment is invalid. A spoofed CRC check could allow spoofed index data. However the checksum is encrypted is therefore hard to change in a meaningful way.

- changing the compartment number
  A change of this parameter (1 Byte) would likely result in a non-existing compartment number, which means DHFS would not associate it with one of the valid 8 compartments. Therefore DHFS would never access it. If the compartment number were luckily a valid compartment number, it would not have it listed in the correct compartment index and never try to access it.

- changing the compressed index data (fragment)
  If the compressed data were changed, it would very likely result in a damaged uncompressed data. Then the index format (separators) would not appear as DHFS expects them. This would result in a parsing error. DHFS would run into an Exception, because it relies on the index data format, which is checked by the CRC checksum.

- changing the index data of the index fragments
  The index data is stored compressed and is only uncompressed if an compartment is in the state "opened". Then the index is held in memory. For reaching the opened state the secret key is needed. Then the data of that compartment is plaintext read- and write-able. The user has to avoid the szenario of having the device inspected if any of the compartments are still opened. Even more the DHFS app should be uninstalled before a foreseeable check happens. For unforeseeable checks an automatic close of all compartments could take place after a certain time e.g. 5 min. If the device were powered down with an opened compartment, all changes of this session could be lost, because they were not persisted to the medium.

All payload file data and meta data is HMAC protected (exception carrier file path). Only the unused index entries and the index header are not HMAC protected. So only information about the index (meta meta data) might be changed undetected. An additional HMAC could protect this data too, see Section 5.4
DHFS does not suffer from a boot-time vulnerability as MobiHydra [52] does. There are no known attacks on DHFS that can lead to harm for the confidentiality or integrity of the data. Availability can still be improved (see Section 5.10), but provably not guaranteed, because the use case assumes physical access of the device for an inspector.

4.11 POTENTIAL ABUSE

DHFS could be abused to transfer illegal data e.g. child pornography. DHFS is merely a tool with security emphasis.
Steganography is a topic that is interesting for malware producers. Trojans generally want to be and stay hidden from the user, a virus scanner or a forensic investigator. Standard file system drivers can not access the file slack. If a virus scanner is limited to standard drivers, file slacks can not be inspected. Maleware could be stored in DHFS. Only a loader component would be needed additionally that reads and writes from/to DHFS. Such a loader could easily avoid signature based virus scanners by using randomizing components.

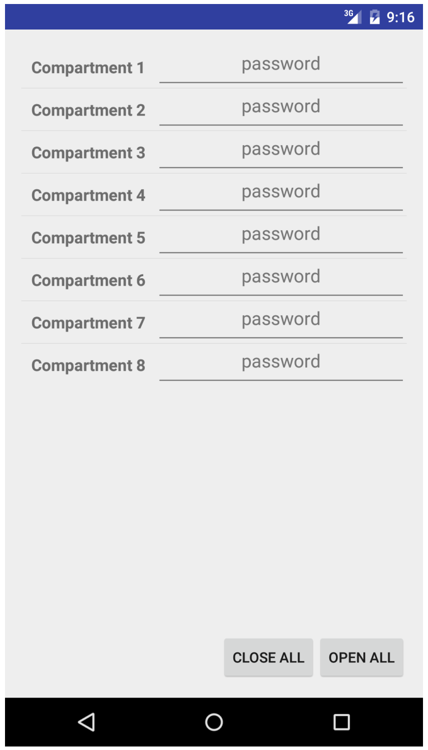
4.12 ANDROID APP CONCEPT

This section suggests an Android User Interface (UI) and provides a conceptional solution as well as implementation hints. The DHFS Android app would use DHFS as Java library, imported as an archive (.jar). The class `HFileSystem` represents the class with which the library user interacts. The app does *not* require root permissions, only access to the SD-card is requested by the app in order to be able to read and write to the FAT32 carrier file system image, see Listing 6. Any Java enabled platform (Servers, PCs, notebooks, tablets) could potentially implement a client.
The Android app could use the library to provide a `Service` limited only to the client app by setting `android:exported="false"` in the service declaration in the manifest file.
The different `Activities` (screens the user sees) could call service methods such as open/close compartments, list compartment's payload files etc. This service has performance intense tasks e.g. searching all file slacks for indices and decrypting them, therefor the service should not run in the Graphical User Interface (GUI) thread but in an separated thread. So the GUI does not freeze when the thread is busy, instead a circular loading bar could be displayed.
The `Activities` in the DHFS app will often work with lists e.g. of compartments or payload files. The communication between `Activities` works with *Intents* that allow to pass on *custom lists* via the `intent.putExtra()` respectively `intent.getExtras()`.
The app would need to interact with other apps. New payload files are an input to the DHFS app and stored payload files e.g. MySecret.pdf are outputs. After storing a file, the user may want to open or export them again e.g. open the stored MySecret.pdf with an PDF viewer app. As a result the DHFS app has to provide the PDF to a viewer of the user's choice. For the DHFS app this is not trivial.
Inter-app-communication typically uses `Intents`, but they use Uniform Resource Identifiers (URIs) to pass on files, these are references of files. DHFS can not provide an URI (due its hidden files) that could be opened by an external app. Instead it would directly pro-

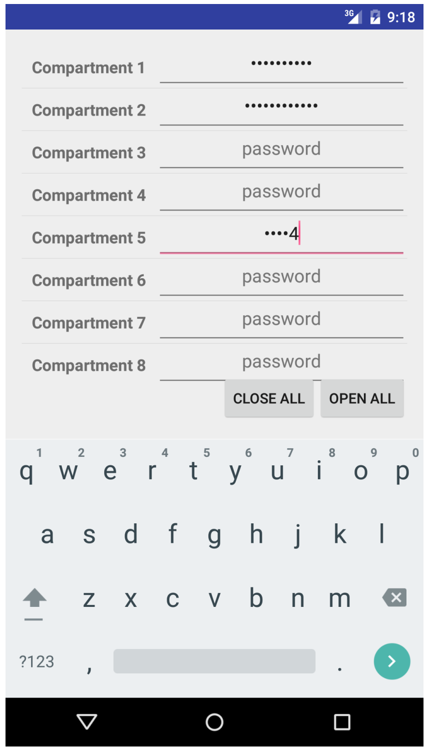
vide the file content as `ByteArray`. This is possible with *Intents*, but external apps would (likely) not expect this form. They would expect an URI and try to open it as a file. Since Android 4.4 (API level 19) the Storage Access Framework (SAF) [18] which is a *content provider* is available. This is an interface for cloud file providers e.g. Google Drive to browse and access remote files on Android. SAF offers a file picker (`Intent.ACTION_OPEN_DOCUMENT`) that can be used for selecting input (payload) files. Additionally DHFS could implement a `DocumentsProvider` as described by Google documentation [18] to make payload files available to other apps. A `DocumentsProvider` class would wrap the `ByteArray` of the payload data. In this case the file data is not provided with an URI, but an `ParcelFileDescriptor` to other apps. This avoids the problem of using `Intents` that work with URIs.

In contrast to cloud file providers DHFS does not need to transfer files over the Internet to the user, but only from the locally hidden file system to the user. Nevertheless the interface works for this purpose.

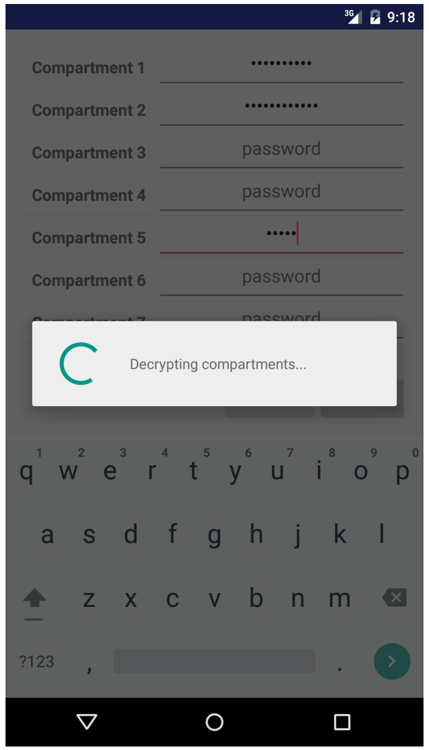
It is important that after a use of DHFS all opened compartments are closed again. For convenience the app could support this by automatically closing all compartments if it thinks the use is over e.g. after 5 minutes of the last user action or when the display is turned off (using a android broadcast receiver for this event). Figure 20 and Figure 21 show how an Android client app could look like. These screen shots were taken on the Android emulator.



(a) Start- and logon- screen

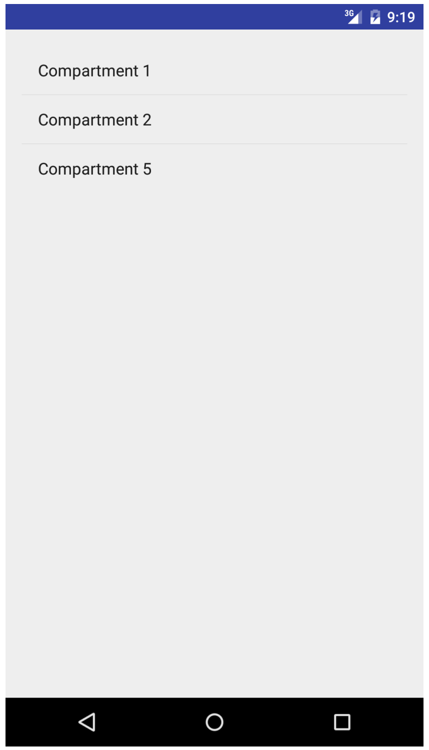


(b) Entering compartment passwords



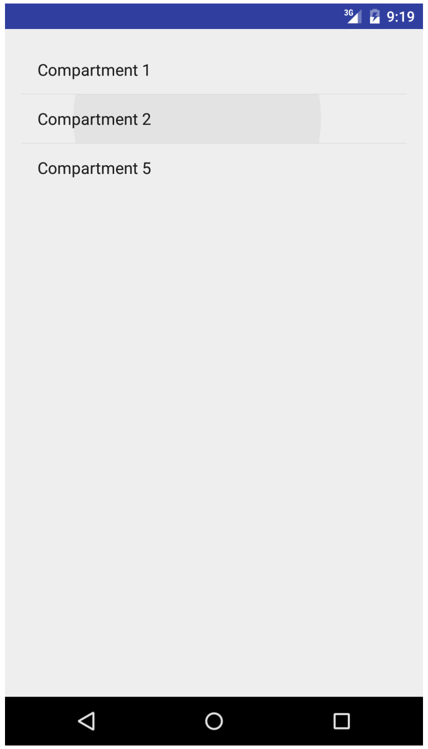
Decrypting compartments...

(c) Opening compartments in an additional thread

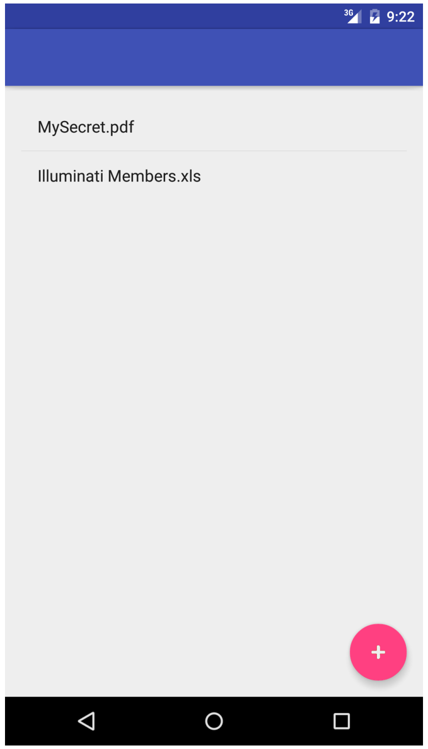


(d) List of opened compartments

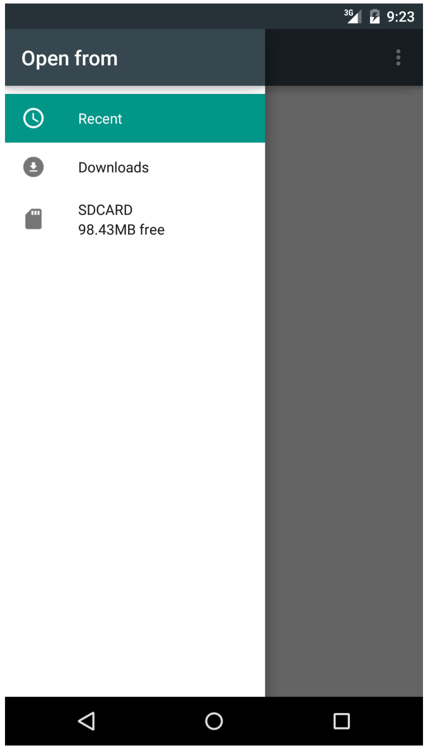
Figure 20: Screen shot series of DHFS Client App - part 1



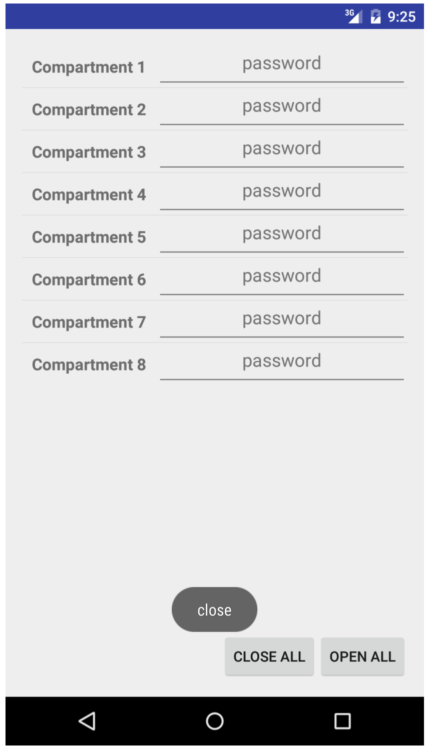
(a) Tap compartment 2 to open it



(b) Payload file list of compartment 2 - tapping a file would display app(s) to open it



(c) Tapping the "plus" in the right bottom triggers the *file picker* to add a new payload file



(d) Tap the "back" button in the left bottom until the initial screen reappears and tap "close all" to end the app

Figure 21: Screen shot series of DHFS Client App - part 2

Part IV

FUTURE WORK, SUMMARY AND CONCLUSION

5 FUTURE WORK

5.1 DIRECT BLOCK DEVICE ACCESS

Currently the FAT32 carrier file system is located in an image file that is e.g. on a mobile phones's SD-card. This image file is visible and easy to inspect, but for a proof-of-concept this disadvantages are accepted. The data would be better hidden if instead of the image file the actual SD-card file system were used. Section 4.2 discusses an attempt to do this, but SE Linux policies forbid it. For an Android firmware manufacturer it would be possible to simply adjust the SE Linux policies to allow access to a block device for the use case of DHFS, probably by creating a request able permission (similar to Section 4.9.2) for this access. There are several Android firmware alternatives that claim to have an security emphasis e.g. DarkMatter Phone, Guardian Rom, Tails Mobile etc. It would be simple for them to introduce the needed change of the SE-Linux policy.

5.2 ENCRYPTION PARAMETER MASKING

Each file slack header begins with the IV and the salt values. They can not be encrypted. The IV is the same for each payload file fragment and the salt is the same for data and index fragment in a compartment. In the case of an inspector checking the file slack, he might notice those repetitive values. This might be a hint for further investigation. From the point of view from the encryption these parameter can be public. In order to hide these parameters better they could be combined with an HMAC (or another any data checksum) of the encrypted data and subsequently only store the combination e.g. the Xor operation is easily reverse able. As a result no reappearing values would occur. This makes the IV and Salt parameters less visible. An attacker without knowledge of the system will not have access to the original parameters.

5.3 KEY CACHING

It would be a significant performance improvement if the key derived via the PBKDF2 would be cached within one session, otherwise each operation would again derive the needed key from the given password. The PBKDF2 uses performance intense calculations hence reducing the number of calls to it will have a noticeable positive impact.

5.4 INTEGRITY CHECKS

The HMAC protects payload file fragments and payload file meta data, but the index carriers are only CRC32 protected. The index data could also be HMAC protected, by e.g. perpending an HMAC to the complete index or even each index fragment. Then the index would also be adequately integrity checked with cryptographic guarantees that CRC32 lacks. In fact then the CRC32 could be completly omitted, because it is substituted with a better solution.

5.5 ENCRYPTION AND HMAC KEY DECOUPLING

At the moment the same key is used for the encryption and for the creation of the HMAC. This is not a security hole, though using independent keys provides additional security. If one key is compromised the other key is not automatically compromised as well. One would need to have knowledge of both secret keys to trick the encryption and the integrity check. If the encryption is broken the data can be read, but not changed without detection. If the HMAC key is broken, the ciphertext can be changed without detection, but not read it in plaintext.
There is a simple way to implement this decoupling as shown in Figure 22. The Key Derivation Function returns one key. In order to create n new keys the key could be combined (XOR) the SHA-256 hash value of the constant 1. A second new key could be derived by using the constant 2 instead of 1. Then one new key is used for the encryption and the other for the HMAC.

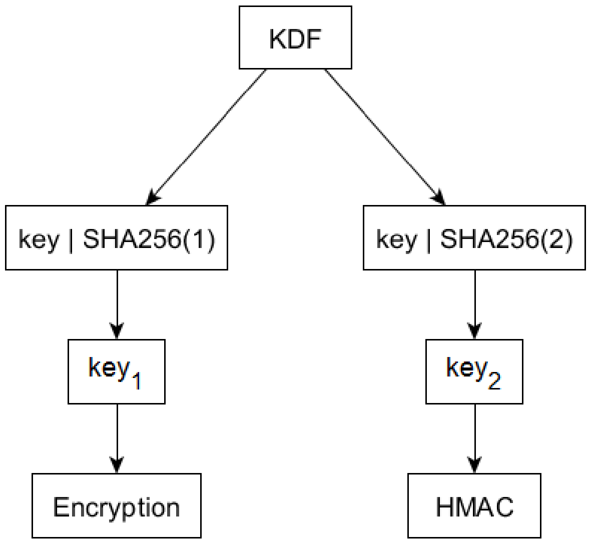


Figure 22: Decoupling encryption and HMAC keys

5.6 removing a payload file

If the user wants to remove a payload file there are (at least) two ways to do it: mark deleted and overwrite.
For both cases the compartment has to be opened. The user would define which file to delete as an input to the delete method. It would search the compartment index for all entries that match the payload file name. The read method already implements this. In order to mark the file deleted, the according index entries would empty the file meta data (payload file name, size and fragment number) and then store the index entries. Then there are no references to the data left, but the data it self would still be there. This is a common way of deleting a file.
Going one step further would be to overwrite the data as well. This leaves less traces of the previously stored data but takes additional write operations (worse performance).
The start is the same as for the mark-deleted methodology. But before the list of index entries of the to be deleted payload file is emptied, it has to be iterated. For each index entry a method has to be called that overwrites this file slack with random data. The file slack write method can be reused for this. After that again the references in the index entries should be removed as described for the mark-deleted method.

5.7 in place payload file editing

At the moment DHFS can read and write data to file slacks. It would be an improvement if a change in a payload file could be persisted without the need to rewrite unchanged fragments.

5.8 balancing dhfs

DHFS uses the available carrier files at the time of its initialization. However new files could be added or existing files could be deleted. When the carrier file system changed DHFS could be balanced, so unused files are included in the index and thus will be usable for DHFS. This process could be implemented similar to the initialization, see Section 4.7.
Balancing will be different for used and unused compartments. Unused compartments could be reinitialized (overwritten). The unused compartment's indices can not be accessed because the indices are encrypted with an unknown key.
The used compartments can only be balanced if they are opened. So the first step would be to ask the user to open all used compartments. The system does not know due the plausible deniability characteristic. Then all the indices should be checked if carrier files were deleted. If so the according index entry should be deleted. This could be done by traversing the carrier file system and creating a complete list of all available files. Then each carrier file path from the compartment

indices entries is looked up and removed in the new carrier file system listing. Then the carrier files of the unused compartments and the newly added carrier files remain in the list.
In the next step the unused compartment should get carrier files associated up to the average of the used compartments. Hence all compartment will have roughly the same amount of carrier files. Either there are less carrier files available then the unused compartments will have less carriers than the used onces, or there are more carrier files available then all compartments should get an equal amount of carrier files. This distribution is the same as in the initialization process. In the case that there are fewer carrier files, the unequal distribution could be accepted (recommended option) or carrier files from the used compartment would have to be reassigned to unused compartments. This could be a problem if used compartments are heavy occupied by payload files and no free carriers are available.

5.9 PAYLOAD DIRECTORIES

DHFS currently only supports to store payload files, no payload directories. Some users might also want to be able to store payload files in payload directories to have a clean hierarchical structure (directory tree). This could be done by extending the payload file name in the index to a payload file path. It would be performance wise desirable that payload files of the same directory would be adjacent in the index or at least near.

5.10 REDUNDANCY

File slacks of carrier files could be overwritten by append operations of processes that are unaware of DHFS. So there is a need for protection in case some file slacks are overwritten. One solution is to introduce redundancy.
There are (al least) three levels on which redundancy could be implemented: Compartments, payload files and file slack fragments.
Each compartment could be stored redundantly e.g. three times. This implies to increase the fixed number of compartment also by a factor three. Alternatively when a payload file is written it would not just be written once but three times. This would use more index entries. The index processing would have to be adjusted by e.g. appending a short string to the payload file name indicating which redundant file it belongs to. Another way would be to create (exact) *duplicates* of the payload fragments. This could be problematic because the fragment size depends on the available file slack size. Fragment duplicates would have to be exact duplicates with the same size, otherwise the used HMAC would detect changes and deny the use of changed data. Therefore the fragment layer can not be recommended due to inefficient use of space. The other two layers (compartment and payload files) would not have a problem with the HMAC check. The payload file level redundancy is preferable because redundancy is only cre-

ated if it is needed, at the time it is needed (at payload file storage), not before. Compartment redundancy would also redundantly store unused (unallocated) data multiple times and consequently waste space, that could be used otherwise.

5.11 PAYLOAD FILE COMPRESSION

Index data is currently stored compressed. An optional compression for payload fragments or files could be implemented almost identically to the index fragment compression. This may help to solve slack space issues and reduce the load on a redundancy concept.

6 SUMMARY AND CONCLUSION

6.1 SUMMARY

Files are stored in blocks. Those blocks are not always completely used. In this thesis this space (file slack) is used to hide data. Hiding files (Steganography) improves the chances that an adversary remains unaware that data is stored at all. Additionally a Plausible Deniability construction with a fixed number of compartments is used and of course the data is encrypted (AES-256-CTR with HMAC) as well. A detailed rationale of the Deniable Hidden File System for design and encryption decisions is given that shows how its challenges are met.
The result is a proof-of-concept implementation of a file slack file system with a security emphasis. Moreover an Android app concept and user interface for DHFS is presented. Furthermore the design and particularly the encryption are reviewed in an security analysis.

6.2 CONCLUSION

This thesis proves that a file system based on file slacks is indeed possible and practical. It offers up to several GBytes of space. DHFS offers strong encryption in addition to a chance that an inspector will overlook that there is data hidden.
Even if it should come to a confrontation e.g. in a court proceeding the user can surrender some mildly compromising data and plausibly deny possessing more data than what was provided. While full cooperation is perceived by the inspector, secret data will remain private. Private means that the confidentiality and integrity of the hidden data is guaranteed by the encryption system. A proposed security analysis gave no indication of security holes being present.
In order to improve availability the introduction of redundancy can be recommended. However, availability *can not* be guaranteed presuming an inspector has physical access to the device.
Future research could aim at bringing a file slack file system into a release stadium, providing more features, implementations for more platforms, improving performance and usability. Then integration in a secure Android firmware could be achieved.

Part V

APPENDIX

A ADDITIONAL MATERIAL

A.1 SIMPLE USE CASE AND TEST

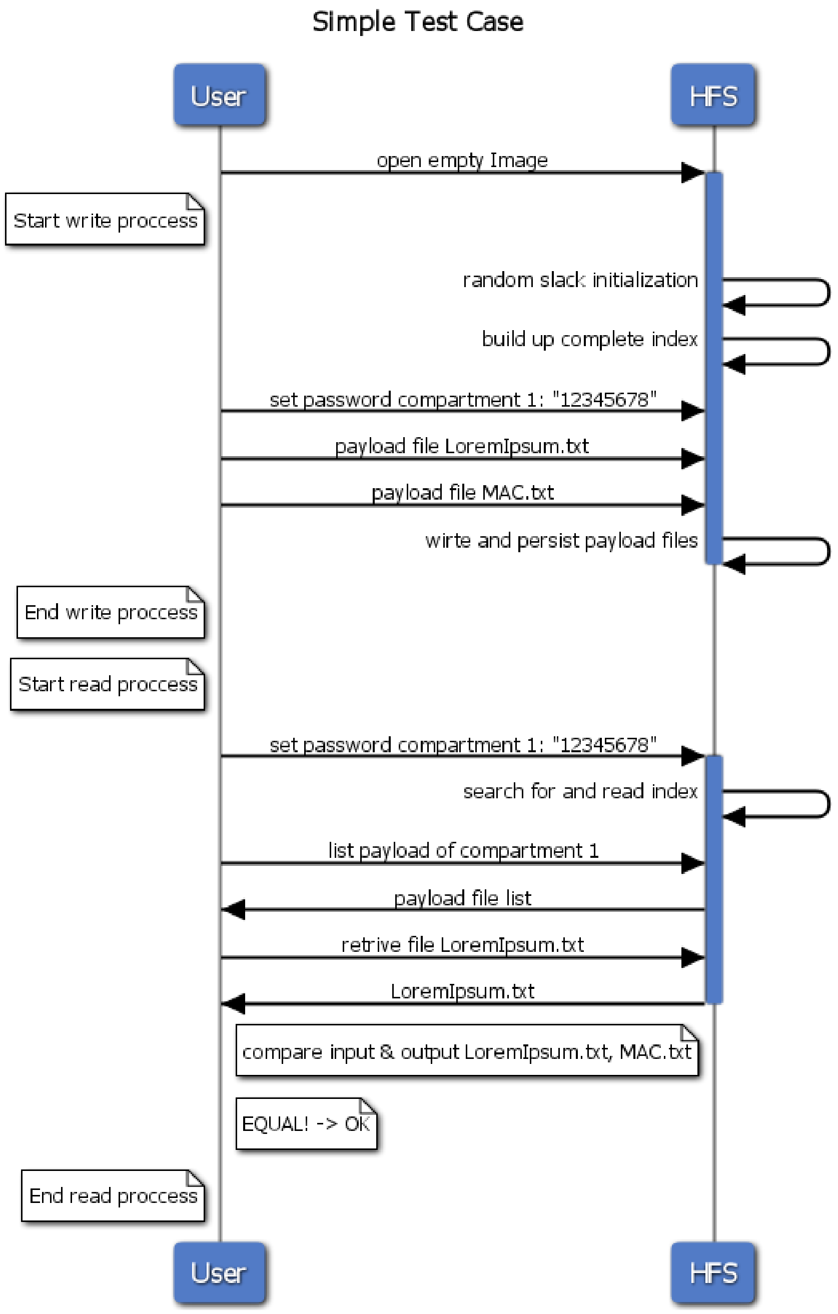

Simple Test Case
User
HFS
open empty Image
Start write proccess
random slack initialization
build up complete index
set password compartment 1: "12345678"
payload file LoremIpsum.txt
payload file MAC.txt
wirte and persist payload files
End write proccess
Start read proccess
set password compartment 1: "12345678"
search for and read index
list payload of compartment 1
payload file list
retrive file LoremIpsum.txt
LoremIpsum.txt
compare input & output LoremIpsum.txt, MAC.txt
EQUAL! -> OK
End read proccess
User
HFS

Figure 23: Simple Use Case of DHFS

Listing 8: Verbose debugging output of the test case depicted in Figure 23 useing the image file `fat32test.img`

```
127 Files found with slacks
--- Compartment 1:
index carrier path: /word documents/bar.docx
offers 478 summedCarrierSlackspace (nett) 473
index carrier path: /libfuse-master/LIB/cuse_lowlevel.c
offers 401 summedCarrierSlackspace (nett) 869
--- Compartment 2:
index carrier path: /libfuse-master/LIB/FUSE.C
offers 424 summedCarrierSlackspace (nett) 419
index carrier path: /added
offers 507 summedCarrierSlackspace (nett) 921
--- Compartment 3:
index carrier path: /libfuse-master/UTIL/fusermount.c
offers 141 summedCarrierSlackspace (nett) 136
index carrier path: /libfuse-master/LIB/FUSE_I.H
offers 427 summedCarrierSlackspace (nett) 558
index carrier path: /Tor exit node/Peham-ISP_und_die_
    Anonymisierung_seiner_Kunden.pdf
offers 99 summedCarrierSlackspace (nett) 652
index carrier path: /libfuse-master/LIB/mount_util.h
offers 301 summedCarrierSlackspace (nett) 948
--- Compartment 4:
index carrier path: /WEBSITE/WISSEN/CONTENT/E16607/E16332/E
    257286/E257772/employee_groups_wiss257774/employees258536/
    SabrinaWeigl_ger.jpg
offers 291 summedCarrierSlackspace (nett) 286
index carrier path: /sample.txt
offers 506 summedCarrierSlackspace (nett) 787
--- Compartment 5:
index carrier path: /libfuse-master/DOC/KERNEL.TXT
offers 247 summedCarrierSlackspace (nett) 242
index carrier path: /Tor exit node/Hoermanseder-Konfiguration_und
    _Betrieb_eines_Tor_Servers.pdf
offers 428 summedCarrierSlackspace (nett) 665
index carrier path: /New Textdocument.txt
offers 507 summedCarrierSlackspace (nett) 1167
--- Compartment 6:
index carrier path: /word documents/U.docx
offers 478 summedCarrierSlackspace (nett) 473
index carrier path: /libfuse-master/INCLUDE/FUSE_OPT.H
offers 131 summedCarrierSlackspace (nett) 599
index carrier path: /ECLIPSE/ECLIPSE.INI
offers 53 summedCarrierSlackspace (nett) 647
index carrier path: /libfuse-master/UTIL/mount.fuse.c
offers 21 summedCarrierSlackspace (nett) 663
index carrier path: /WEBSITE/WISSEN/CONTENT/E16607/E16332/JKU-
    Banner-forschen_ger.jpg
offers 23 summedCarrierSlackspace (nett) 681
index carrier path: /ROOTDIR2/SUBDIR5/unsuspicious File.html
offers 465 summedCarrierSlackspace (nett) 1141
--- Compartment 7:
index carrier path: /libfuse-master/AUTHORS
offers 464 summedCarrierSlackspace (nett) 459
```

```
index carrier path: /libfuse-master/LIB/fuse_signals.c
offers 240 summedCarrierSlackspace (nett) 694
index carrier path: /libfuse-master/LIB/fuse_lowlevel.c
offers 55 summedCarrierSlackspace (nett) 744
index carrier path: /libfuse-master/EXAMPLE/FIOC.H
offers 169 summedCarrierSlackspace (nett) 908
--- Compartment 8:
index carrier path: /libfuse-master/DOC/.gitignore
offers 487 summedCarrierSlackspace (nett) 482
index carrier path: /WEBSITE/ZONA/CONTENT/index.html
offers 134 summedCarrierSlackspace (nett) 611
index carrier path: /libfuse-master/EXAMPLE/Makefile.am
offers 56 summedCarrierSlackspace (nett) 662
index carrier path: /libfuse-master/INCLUDE/cuse_lowlevel.h
offers 499 summedCarrierSlackspace (nett) 1156

carrier space available 478 of file /word documents/bar.docx
    cSVFragmentlen: 690
write checksum 1896714378 with len 324 to carrier bar.docx
    neededspace 332 of fragment_/New Long Directory/Neues
    Textdokument.txt::0:0:0: >/PODCAST/Aerger und
    Selbsterkenntnis.m4a::0:0:0: >/WEBSITE/newsfeed.xml.html
    ::0:0:0: >/WEBSITE/WISSEN/CONTENT/E16607/E16332/E16044/
    employee_groups_wiss108511/employees119729/MS_688.JPG::0:0:0:
     >/WEBSITE/ZONA/CONTENT/E41510/employee_groups_wiss104378/
    employees297320/100Zahnrad2.jpg::0:0:0: >/libfuse-master/.
    gitignore::0:0:0: >/libfuse-master/README.MD::0:0:0: >/
    libfuse-master/DOC/Makefile.am::0:0:0: >/libfuse-master/
    EXAMPLE/FSEL.C::0:0:0: >/libfuse-master/EXAMPLE/NULL.C
    ::0:0:0: >/libfuse-master/LIB/BUFFER.C::0:0:0: >/libfuse-
    master/LIB/FUSE_MT.C::0:0:0: >/libfuse-master/LIB/mount_bsd.c
    ::0:0:0: >/libfuse-master/TEST/TEST.C::0:0:0: >_
carrier space available 401 of file /libfuse-master/LIB/cuse_
    lowlevel.c cSVFragmentlen: 0
carrier space available 424 of file /libfuse-master/LIB/FUSE.C
    cSVFragmentlen: 762
write checksum 3090469600 with len 361 to carrier FUSE.C
    neededspace 369 of fragment_/ROOTDIR1/SUBDIR2/NOTE.TXT
    ::0:0:0: >/ECLIPSE/.eclipseproduct::0:0:0: >/Tor exit node/
    Bergauer-Strafrechtliche_Aspekte_der_Providerhaftung.pdf
    ::0:0:0: >/WEBSITE/content.html::0:0:0: >/WEBSITE/WISSEN/
    CONTENT/E16607/E16332/E16044/employee_groups_wiss16041/
    employees136345/GeorgZenz_ger.png::0:0:0: >/WEBSITE/WISSEN/
    CONTENT/E16607/E16332/E257286/E257772/employee_groups_wiss
    257774/employees258537/GeorgZenz_ger.png::0:0:0: >/WEBSITE/
    ZONA/CONTENT/E41510/employee_groups_wiss195896/employees
    279316/Unbenannt.png::0:0:0: >/libfuse-master/README.NFS
    ::0:0:0: >/libfuse-master/DOC/mount.fuse.8::0:0:0: >/libfuse-
    master/EXAMPLE/fselclient.c::0:0:0: >/libfuse-master/LIB/FUSE
    _OPT.C::0:0:0: >/libfuse-master/LIB/mount_util.c::0:0:0: >/
    libfuse-master/UTIL/.gitignore::0:0:0: >_
carrier space available 507 of file /added cSVFragmentlen: 0
carrier space available 141 of file /libfuse-master/UTIL/
    fusermount.c cSVFragmentlen: 58
```

```
write checksum 2336604096 with len 52 to carrier fusermount.c
    neededspace 60 of fragment_/ECLIPSE/artifacts.xml::0:0:0: >/
    WEBSITE/im.html::0:0:0: >_
carrier space available 427 of file /libfuse-master/LIB/FUSE_I.H
    cSVFragmentlen: 432
write checksum 2675982989 with len 242 to carrier FUSE_I.H
    neededspace 250 of fragment_/WEBSITE/WISSEN/CONTENT/E16607/E
    16332/E16044/employee_groups_wiss16041/employees172829/
    SabrinaWeigl_ger.jpg::0:0:0: >/WEBSITE/ZONA/MS_9178.html
    ::0:0:0: >/WEBSITE/ZONA/CONTENT/E41656/index.html::0:0:0: >/
    libfuse-master/configure.ac::0:0:0: >/libfuse-master/DOC/
    IMAGES/490px-FUSE_structure.svg.png::0:0:0: >/libfuse-master/
    EXAMPLE/FUSEXMP.C::0:0:0: >/libfuse-master/INCLUDE/FUSE.H
    ::0:0:0: >/libfuse-master/LIB/fuse_session.c::0:0:0: >_
carrier space available 99 of file /Tor exit node/Peham-ISP_und_
    die_Anonymisierung_seiner_Kunden.pdf cSVFragmentlen: 0
carrier space available 301 of file /libfuse-master/LIB/mount_
    util.h cSVFragmentlen: 0
carrier space available 291 of file /WEBSITE/WISSEN/CONTENT/E
    16607/E16332/E257286/E257772/employee_groups_wiss257774/
    employees258536/SabrinaWeigl_ger.jpg cSVFragmentlen: 423
write checksum 3456944037 with len 243 to carrier SabrinaWeigl_
    ger.jpg neededspace 251 of fragment_/ECLIPSE/ECLIPSE.EXE
    ::0:0:0: >/Tor exit node/Mayrhofer.M-Anonymisierung_im_Web _
    und_Grundrechte.pdf::0:0:0: >/WEBSITE/MS_6.html::0:0:0: >/
    WEBSITE/WISSEN/CONTENT/E16607/E16332/E16055/JKU-Banner-
    forschen_ger.jpg::0:0:0: >/WEBSITE/ZONA/CONTENT/e41510.html
    ::0:0:0: >/libfuse-master/COPYING::0:0:0: >/libfuse-master/
    DOC/Doxyfile::0:0:0: >/libfuse-master/EXAMPLE/.gitignore
    ::0:0:0: >/libfuse-master/EXAMPLE/fusexmp_fh.c::0:0:0: >_
carrier space available 506 of file /sample.txt cSVFragmentlen:
    134
write checksum 3722112691 with len 91 to carrier sample.txt
    neededspace 99 of fragment_/libfuse-master/INCLUDE/fuse_
    common.h::0:0:0: >/libfuse-master/LIB/MODULES/ICONV.C::0:0:0:
     >/libfuse-master/UTIL/init_script::0:0:0: >_
carrier space available 247 of file /libfuse-master/DOC/KERNEL.
    TXT cSVFragmentlen: 263
write checksum 2331923815 with len 182 to carrier KERNEL.TXT
    neededspace 190 of fragment_/adddir/Another long file.txt
    ::0:0:0: >/Tor exit node/Mayrhofer.R-Anonym_im_Internet_
    mittels_Tor.pdf::0:0:0: >/WEBSITE/MS_10393.html::0:0:0: >/
    WEBSITE/WISSEN/CONTENT/E16607/E16332/E16099/E16086/fodok_6_
    eng.jpg::0:0:0: >/WEBSITE/ZONA/CONTENT/imprint.html::0:0:0:
    >_
carrier space available 428 of file /Tor exit node/Hoermanseder-
    Konfiguration_und_Betrieb_eines_Tor_Servers.pdf
    cSVFragmentlen: 452
write checksum 3225083542 with len 202 to carrier Hoermanseder-
    Konfiguration_und_Betrieb_eines_Tor_Servers.pdf neededspace
    210 of fragment_/word documents/Hengstberger Martin 0755731.
    docx::0:0:0: >/libfuse-master/fuse3.pc.in::0:0:0: >/libfuse-
    master/DOC/fusermount.1::0:0:0: >/libfuse-master/EXAMPLE/
    CUSEXMP.C::0:0:0: >/libfuse-master/EXAMPLE/fuse_lo-plus.c
    ::0:0:0: >/libfuse-master/INCLUDE/fuse_kernel.h::0:0:0: >/
    libfuse-master/LIB/fuse_loop.c::0:0:0: >/libfuse-master/LIB/
    fuse_versionscript::0:0:0: >/libfuse-master/LIB/MODULES/
    SUBDIR.C::0:0:0: >/libfuse-master/UTIL/Makefile.am::0:0:0: >_
carrier space available 507 of file /New Textdocument.txt
    cSVFragmentlen: 0
carrier space available 478 of file /word documents/U.docx
    cSVFragmentlen: 553
write checksum 201692076 with len 261 to carrier U.docx
    neededspace 269 of fragment_/adddir/Neues Textdokument.txt
    ::0:0:0: >/ECLIPSE/ECLIPSEC.EXE::0:0:0: >/WEBSITE/WINIE/
    content.html::0:0:0: >/WEBSITE/WISSEN/CONTENT/E16607/E16332/E
    16099/E16086/fodok_6_ger.jpg::0:0:0: >/word documents/foo.
    docx::0:0:0: >/libfuse-master/MAKECONF.SH::0:0:0: >/libfuse-
    master/DOC/how-fuse-works::0:0:0: >/libfuse-master/EXAMPLE/
    FIOC.C::0:0:0: >/libfuse-master/EXAMPLE/HELLO.C::0:0:0: >/
    libfuse-master/INCLUDE/fuse_lowlevel.h::0:0:0: >/libfuse-
    master/LIB/fuse_loop_mt.c::0:0:0: >/libfuse-master/LIB/HELPER
    .C::0:0:0: >/libfuse-master/TEST/.gitignore::0:0:0: >_
carrier space available 131 of file /libfuse-master/INCLUDE/FUSE_
    OPT.H cSVFragmentlen: 0
carrier space available 53 of file /ECLIPSE/ECLIPSE.INI
    cSVFragmentlen: 0
carrier space available 21 of file /libfuse-master/UTIL/mount.
    fuse.c cSVFragmentlen: 0
carrier space available 23 of file /WEBSITE/WISSEN/CONTENT/E
    16607/E16332/JKU-Banner-forschen_ger.jpg cSVFragmentlen: 0
carrier space available 465 of file /ROOTDIR2/SUBDIR5/
    unsuspicious File.html cSVFragmentlen: 0
carrier space available 464 of file /libfuse-master/AUTHORS
    cSVFragmentlen: 512
write checksum 1206031124 with len 270 to carrier AUTHORS
    neededspace 278 of fragment_/ECLIPSE/README/readme_eclipse.
    html::0:0:0: >/Tor exit node/Sonntag-Registrierung_bei_der_
    RTR-Statistik.pdf::0:0:0: >/WEBSITE/WISSEN/CONTENT/small_Foto
    _Kooperationen_Keplergeb_ger.jpg::0:0:0: >/WEBSITE/WISSEN/
    CONTENT/E16607/E16332/E16219/forschen1a_eng.jpg::0:0:0: >/
    WEBSITE/ZONA/CONTENT/MS_9172.html::0:0:0: >/libfuse-master/
    Makefile.am::0:0:0: >/libfuse-master/EXAMPLE/HELLO_LL.C
    ::0:0:0: >/libfuse-master/LIB/Makefile.am::0:0:0: >/libfuse-
    master/TEST/Makefile::0:0:0: >/libfuse-master/UTIL/udev.rules
    ::0:0:0: >_
carrier space available 240 of file /libfuse-master/LIB/fuse_
    signals.c cSVFragmentlen: 0
carrier space available 55 of file /libfuse-master/LIB/fuse_
    lowlevel.c cSVFragmentlen: 0
carrier space available 169 of file /libfuse-master/EXAMPLE/FIOC.
    H cSVFragmentlen: 0
carrier space available 487 of file /libfuse-master/DOC/.
    gitignore cSVFragmentlen: 622
write checksum 1401573861 with len 337 to carrier .gitignore
    neededspace 345 of fragment_/New Long Directory/Another long
    file.txt::0:0:0: >/PODCAST/argumentieren.m4a::0:0:0: >/Tor
    exit node/Tischlinger-Schwierigkeiten_bei_der_Strafverfolgung
    _durch_Anonymisierung.pdf::0:0:0: >/WEBSITE/WISSEN/CONTENT/E
    16607/E16332/E16219/forschen1a_ger.jpg::0:0:0: >/WEBSITE/ZONA
    /CONTENT/MS_9227.html::0:0:0: >/word documents/J.docx::0:0:0:
     >/libfuse-master/NEWS::0:0:0: >/libfuse-master/DOC/MAINPAGE.
    DOX::0:0:0: >/libfuse-master/EXAMPLE/fioclient.c::0:0:0: >/
    libfuse-master/INCLUDE/Makefile.am::0:0:0: >/libfuse-master/
    LIB/fuse_misc.h::0:0:0: >/libfuse-master/LIB/MOUNT.C::0:0:0:
    >/libfuse-master/TEST/stracedecode.c::0:0:0: >_
carrier space available 134 of file /WEBSITE/ZONA/CONTENT/index.
    html cSVFragmentlen: 0
carrier space available 56 of file /libfuse-master/EXAMPLE/
    Makefile.am cSVFragmentlen: 0
carrier space available 499 of file /libfuse-master/INCLUDE/cuse_
    lowlevel.h cSVFragmentlen: 0
compressed frag. length: 324
uncompressed frag. length: 690
read 1896714378 bar.docx read len 324 len parameter324_/New Long
    Directory/Neues Textdokument.txt::0:0:0: >/PODCAST/Aerger und
     Selbsterkenntnis.m4a::0:0:0: >/WEBSITE/newsfeed.xml.html
    ::0:0:0: >/WEBSITE/WISSEN/CONTENT/E16607/E16332/E16044/
    employee_groups_wiss108511/employees119729/MS_688.JPG::0:0:0:
     >/WEBSITE/ZONA/CONTENT/E41510/employee_groups_wiss104378/
    employees297320/100Zahnrad2.jpg::0:0:0: >/libfuse-master/.
    gitignore::0:0:0: >/libfuse-master/README.MD::0:0:0: >/
    libfuse-master/DOC/Makefile.am::0:0:0: >/libfuse-master/
    EXAMPLE/FSEL.C::0:0:0: >/libfuse-master/EXAMPLE/NULL.C
    ::0:0:0: >/libfuse-master/LIB/BUFFER.C::0:0:0: >/libfuse-
    master/LIB/FUSE_MT.C::0:0:0: >/libfuse-master/LIB/mount_bsd.c
    ::0:0:0: >/libfuse-master/TEST/TEST.C::0:0:0: >_Cnr 1
CRC match!
read into compartment1 /New Long Directory/Neues Textdokument.txt
    ::0:0:0:     from bar.docx
read into compartment1 /PODCAST/Aerger und Selbsterkenntnis.m4a
    ::0:0:0:     from bar.docx
read into compartment1 /WEBSITE/newsfeed.xml.html::0:0:0:
    from bar.docx
read into compartment1 /WEBSITE/WISSEN/CONTENT/E16607/E16332/E
    16044/employee_groups_wiss108511/employees119729/MS_688.JPG
    ::0:0:0:     from bar.docx
read into compartment1 /WEBSITE/ZONA/CONTENT/E41510/employee_
    groups_wiss104378/employees297320/100Zahnrad2.jpg::0:0:0:
        from bar.docx
read into compartment1 /libfuse-master/.gitignore::0:0:0:
    from bar.docx
read into compartment1 /libfuse-master/README.MD::0:0:0:     from
     bar.docx
read into compartment1 /libfuse-master/DOC/Makefile.am::0:0:0:
        from bar.docx
read into compartment1 /libfuse-master/EXAMPLE/FSEL.C::0:0:0:
        from bar.docx
read into compartment1 /libfuse-master/EXAMPLE/NULL.C::0:0:0:
        from bar.docx
read into compartment1 /libfuse-master/LIB/BUFFER.C::0:0:0:
    from bar.docx
read into compartment1 /libfuse-master/LIB/FUSE_MT.C::0:0:0:
    from bar.docx
read into compartment1 /libfuse-master/LIB/mount_bsd.c::0:0:0:
        from bar.docx
read into compartment1 /libfuse-master/TEST/TEST.C::0:0:0:
    from bar.docx
```

```
Payload file: C:\Dropbox\uni\BachelorArbeit\workspace\fat32-lib\
    LoremIpsum.txt 574
carrier file has 490 available
writting CRC 2894306100 of length=453
payload write to LoremIpsum.txt part#1 to carrier /New Long
    Directory/Neues Textdokument.txt
carrier file has 143 available
writting CRC 900908554 of length=106
payload write to LoremIpsum.txt part#2 to carrier /PODCAST/Aerger
     und Selbsterkenntnis.m4a
carrier file has 484 available
writting CRC 1726646595 of length=15
payload write to LoremIpsum.txt part#3 to carrier /WEBSITE/
    newsfeed.xml.html
payloadlist (compartment 1):
Name: LoremIpsum.txt len: 574 parts: 3
HMAC: ojtcw/3ha347ZXrjnQ+0CBStomTfb46Fb/V0KSaBrMA=
HMAC OK
HMAC: iF2AROeiascXMuRSpwI9ULziI9QzTw4VjetW4A15QT8=
HMAC OK
HMAC: NGFr2tclkC0YOocXepUEnxMiEmoFHrnXQ6BjMqfTvcg=
HMAC OK
original length:574 payloadbinary length:574
in and out binary EQUAL!
Payload file: C:\Dropbox\uni\BachelorArbeit\workspace\fat32-lib\
    MAC.txt 259
carrier file has 22 available
carrier file has 204 available
writting CRC 467222172 of length=167
payload write to MAC.txt part#1 to carrier /WEBSITE/ZONA/CONTENT/
    E41510/employee_groups_wiss104378/employees297320/100Zahnrad
    2.jpg
carrier file has 487 available
writting CRC 1689367499 of length=92
payload write to MAC.txt part#2 to carrier /libfuse-master/.
    gitignore
payloadlist (compartment 1):
Name: LoremIpsum.txt len: 574 parts: 3
Name: MAC.txt len: 259 parts: 2
HMAC: V58LvKRLmRGFrASvi/bjz8LJNL90zT4h9zaQxyZ6ziE=
HMAC OK
HMAC: SlCJNZeXverl0ZN9SYHuus67PiqWr5OF+hB90EJjQI8=
HMAC OK
original length:259 payloadbinary length:259
in and out binary EQUAL!
```

A.2 DHFS CLASS STRUCTURE

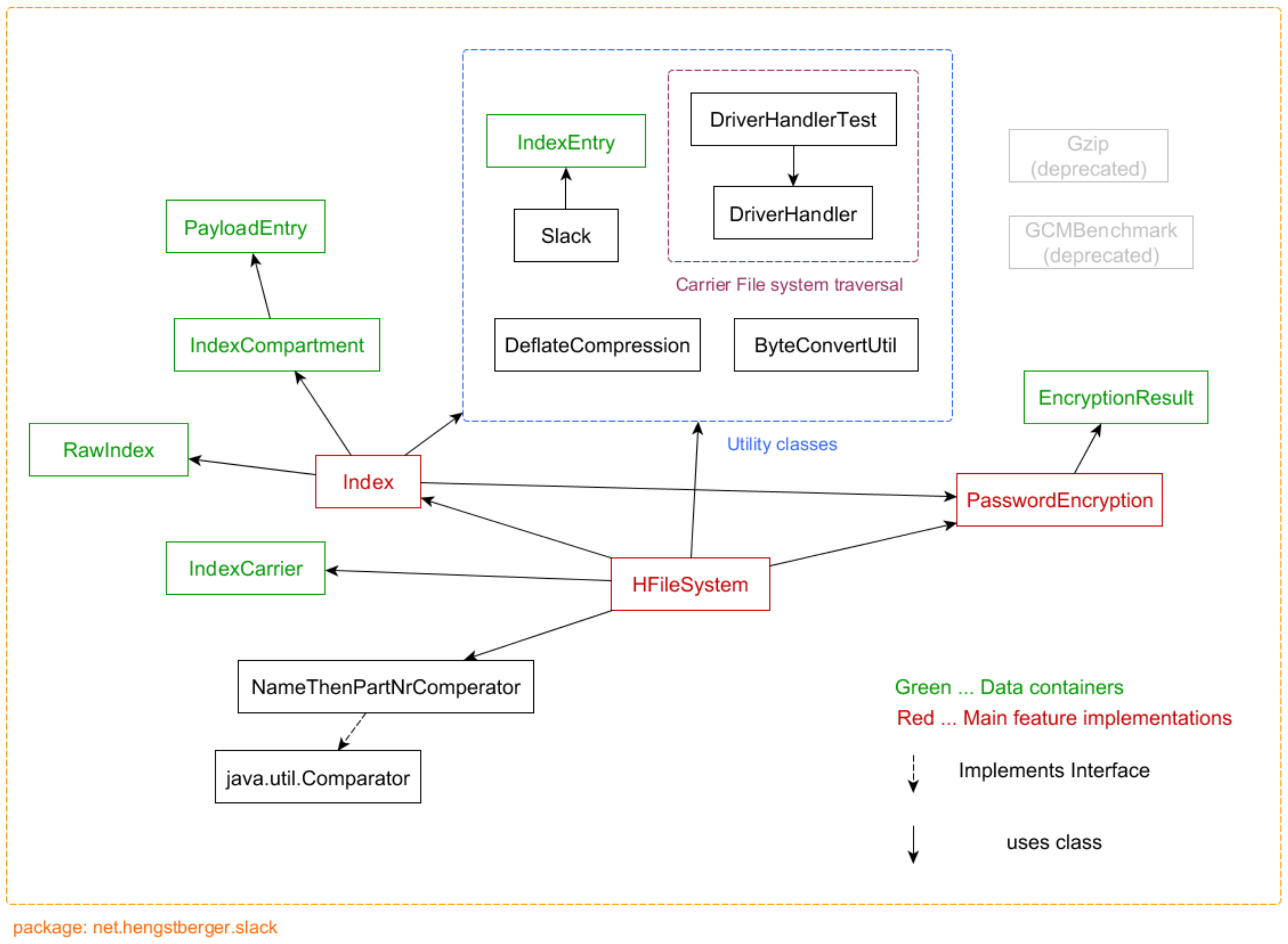


Figure 24: DHFS Class Structure

BIBLIOGRAPHY

[1] Tom Ritter Alex Balducci, Sean Devlin. Open Crypto Audit Project TrueCrypt, 2015. URL http://opencryptoaudit.org/reports/TrueCrypt_Phase_II_NCC_OCAP_final.pdf. Online; accessed May-2016.

[2] Ross Anderson, Roger Needham, and Adi Shamir. The Steganographic File System. In *Information Hiding*, volume 1525 of *Lecture Notes in Computer Science*, pages 73–82. Springer Berlin Heidelberg, 1998. ISBN 978-3-540-65386-8. doi: 10.1007/3-540-49380-8_6. URL http://dx.doi.org/10.1007/3-540-49380-8_6.

[3] Mihir Bellare and Chanathip Namprempre. *Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm*, pages 531–545. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000. ISBN 978-3-540-44448-0. doi: 10.1007/3-540-44448-3_41. URL http://dx.doi.org/10.1007/3-540-44448-3_41.

[4] Mathieu Blondel and Zephaniah E. Loss-Cutler-Hull. WikipediaFS, 2015. URL https://sourceforge.net/projects/wikipediafs/. Online; accessed November-2015.

[5] M. Borowski and M. Lesniewicz. Modern usage of old one-time pad. In *Communications and Information Systems Conference (MCC), 2012 Military*, pages 1–5, Oct 2012.

[6] Neil Coffey. Removing the 128-bit key restriction in Java, 2012. URL http://www.javamex.com/tutorials/cryptography/unrestricted_policy_files.shtml. Online; accessed April-2016.

[7] Jean-Daniel Dodin. Device Major and Minor Numbers, 2009. URL http://www.tldp.org/HOWTO/Partition-Mass-Storage-Definitions-Naming-HOWTO/x183.html. Online; accessed May-2016.

[8] Morris Dworkin. NIST SP-800-38A Recommendation for Block Cipher Modes of Operation, 2001. URL http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. Online; accessed April-2016.

[9] Morris Dworkin. NIST SP-800-38D Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007. URL http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf. Online; accessed April-2016.

[10] Matthias Treydte "Waldheinz" et al. FAT32 Driver Java Library. URL https://android.googlesource.com/platform/external/fat32lib/. Online; accessed March-2016.

[11] Milan Broz et al. Cryptsetup/LUKS/Dm-Crypt, 2011. URL https://gitlab.com/cryptsetup/cryptsetup/. Online; accessed May-2016.

[12] B. Kaliski for the Internet Engineering Task Force. PKCS #5: Password-Based Cryptography Specification Version 2.0, 2000. URL https://tools.ietf.org/html/rfc2898. Online; accessed April-2016.

[13] OWASP Foundation. Guide to Cryptography, 2015. URL https://www.owasp.org/index.php/Guide_to_Cryptography. Online; accessed April-2016.

[14] TrueCrypt Foundation. TrueCrypt - Encryption for the masses, 2014. URL http://truecrypt.sourceforge.net/. Online; accessed Feb-2016.

[15] Google. Android Security 2014 Year in Review, . URL https://static.googleusercontent.com/media/source.android.com/de//security/reports/Google_Android_Security_2014_Report_Final.pdf. Online; accessed June-2016.

[16] Google. Android JOBB Development Tool, . URL http://developer.android.com/tools/help/jobb.html. Online; accessed March-2016.

[17] Google. Preview Android N for Developers, . URL https://developer.android.com/preview/api-overview.html. Online; accessed June-2016.

[18] Google. Content Providers Storage Access Framework, 2013. URL https://developer.android.com/guide/topics/providers/document-provider.html. Online; accessed June-2016.

[19] Google. Android API 23 Reference, 2015. URL http://developer.android.com/reference/. Online; accessed Nov-2015.

[20] K. Gopalan. Audio steganography using bit modification. In *Multimedia and Expo, 2003. ICME '03. Proceedings. 2003 International Conference*, volume 1, pages I–629–32 vol.1, July 2003. doi: 10.1109/ICME.2003.1220996.

[21] "Guillaume". A glimpse of ext4 filesystem-level encryption. URL http://blog.quarkslab.com/a-glimpse-of-ext4-filesystem-level-encryption.html. Online; accessed June-2016.

[22] Robert Hackett. Most Common Passwords Found in the Hacked LinkedIn Data, 2016. URL http://fortune.com/2016/05/18/linkedin-breach-passwords-most-common/. Online; accessed June-2016.

[23] Michael Sthultz Hal Berghel, David Hoelzer. Data Hiding Tactics for Windows and Unix File Systems, May 2006. URL http://www.berghel.net/publications/data_hiding/data_hiding.php. Online; accessed May-2016.

[24] Dave Hansen. GMail Filesystem over FUSE, 2015. URL http://sr71.net/projects/gmailfs/. Online; accessed November-2015.

[25] Tyler Hicks and Dustin Kirkland. eCryptfs - The enterprise cryptographic filesystem for Linux, 2012. URL http://ecryptfs.org/. Online; accessed May-2016.

[26] The IEEE and The Open Group. The Open Group Base Specifications Issue 7, 2013. URL http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html#tag_03_276. Online; accessed May-2016.

[27] Devyn C Johnson. FUSE, 2014. URL http://www.linux.org/threads/fuse.6211/. Online; accessed May-2016.

[28] Hans Kratz. Stackoverflow article "how do i set extended user attributes on android files?", 2013. URL http://stackoverflow.com/questions/17784158/how-do-i-set-extended-user-attributes-on-android-files. Online; accessed Nov-2015.

[29] Thorsten Leemhuis. Mein Geheimnis, dein Geheimnis - Verschlüsselungsfunktion des Linux-Dateisystems Ext4, May 2015. URL http://www.heise.de/ct/ausgabe/2015-20-Verschluesselungsfunktion-des-Linux-Dateisystems-Ext4-2793204.html. Online; accessed May-2016.

[30] Florin Malita. FTP File System, 2015. URL http://ftpfs.sourceforge.net/. Online; accessed November-2015.

[31] Andrew D. McDonald and Markus G. Kuhn. StegFS: A Steganographic File System for Linux. In Andreas Pfitzmann, editor, *Information Hiding*, volume 1768 of *Lecture Notes in Computer Science*, pages 463–477. Springer Berlin Heidelberg, 2000. ISBN 978-3-540-67182-4. doi: 10.1007/10719724_32. URL http://dx.doi.org/10.1007/10719724_32.

[32] Nick Mead. TrueCrypt 7.2 Free open-source disk encryption software. URL http://screenshots.en.sftcdn.net/en/scrn/34000/34872/truecrypt-16.jpg. Online; accessed May-2016.

[33] Mühlbacher. *Betriebssysteme - Grundlagen*. Trauner Verlag, Linz, March 2011. ISBN 978-3-85499-843-3.

[34] Nikratio. SSHFS, 2015. URL https://sourceforge.net/projects/wikipediafs/. Online; accessed November-2015.

[35] NIST. FIPS197: Advanced Encryption Standard (AES), 2001. URL http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf. Online; accessed April-2016.

[36] Timothy M. Peters. DEFY: A Deniable File System for Flash Memory, 2014. URL http://digitalcommons.calpoly.edu/theses/1230.

[37] W. W. Peterson and D. T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, Jan 1961. ISSN 0096-8390. doi: 10.1109/JRPROC.1961.287814.

[38] Thomas & Erin Ptacek. You Don't Want XTS, 2015. URL http://sockpuppet.org/blog/2014/04/30/you-dont-want-xts/. Online; accessed June-2016.

[39] Daniel Ridge. Hiding Data in Slack Space using bmap. URL https://www.computersecuritystudent.com/FORENSICS/HIDING/lesson1/index.html. Online; accessed Feb-2016.

[40] Phillip Rogaway. Evaluation of Some Blockcipher Modes of Operation. Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan, 2011. URL http://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf.

[41] Josef Scharinger. Cryptography Lecture: Introduction to Cryptography, Spring term 2014. Department of Computational Perception, Johannes Kepler University (JKU) Linz.

[42] Dennis Schirrmacher and Jürgen Schmidt. TeslaCrypt 2.0 decryption, 2016. URL http://heise.de/-3094987. Online; accessed April-2016.

[43] IEEE Computer Society. Portable Operating System Interface, 2001. URL http://standards.ieee.org/develop/wg/POSIX.html. Online; accessed Nov-2015.

[44] Jospin Software. Deepsound steganography tool. URL http://jpinsoft.net/DeepSound/. Online; accessed Feb-2016.

[45] Syvaidya. Openstego steganography tool, 2007. URL http://www.openstego.com/. Online; accessed Feb-2016.

[46] Miklos Szeredi. libFUSE - File system in User space, 2001. URL https://github.com/libfuse/. Online; accessed April-2016.

[47] Theodore T'so. Ext4 File system Development Repsitory ext2/e2fsprogs.git. URL https://git.kernel.org/cgit/fs/ext2/e2fsprogs.git/. Online; accessed June-2016.

[48] Wikipedia user "Sven". Structural diagramm of File System in User Space, 2007. URL https://en.wikipedia.org/wiki/

Filesystem_in_Userspace#/media/File:FUSE_structure.svg. Creative Commons 3.0; Online; accessed May-2016.

[49] In Wikipedia. Cryptography, 2016. URL https://en.wikipedia.org/wiki/Cryptography. Online; accessed May-2016.

[50] In Wikipedia. Plausible Deniability, 2016. URL https://en.wikipedia.org/wiki/Plausible_deniability. Online; accessed May-2016.

[51] In Wikipedia. Steganography, 2016. URL https://en.wikipedia.org/wiki/Steganography. Online; accessed May-2016.

[52] Xingjie Yu, Bo Chen, Zhan Wang, Bing Chang, Wen Tao Zhu, and Jiwu Jing. *Information Security: 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, chapter MobiHydra: Pragmatic and Multi-level Plausibly Deniable Encryption Storage for Mobile Devices, pages 555–567. Springer International Publishing, Cham, 2014. ISBN 978-3-319-13257-0. doi: 10.1007/978-3-319-13257-0_36. URL http://dx.doi.org/10.1007/978-3-319-13257-0_36.

[53] ZLib. RFC Gzip Format, 1991. URL http://www.zlib.org/rfc-gzip.html. Online; accessed Dec-2015.

Martin Hengstberger

Curriculum Vitae

Personal Data

Born:	1986 in St.Pölten, Austria
Address:	Freistädter Straße, 4040 Linz, Austria
Mobile:	(0043) 676 6903583
Email:	martin@hengstberger.net
Nationality:	Austria

Education

2015–2016	**Master in Computer Science**, *Johannes Kepler University*, Linz, Focus on **Networks and Security**.
2012–2013	**Master in Computer Science**, *Norwegian University of Science and Technology NTNU* , Trondheim, Norway. Exchange year (ERASMUS)
2010–2015	**Bachelor in Computer Science**, *Johannes Kepler University*, Linz.
2007–2010	**Bachelor in Mechatronics**, *Johannes Kepler University*, Linz, *incomplete*.
2000–2006	**General qualification for university entrance**, *Höhere Technische Bundeslehr- und Versuchsanstalt (HTL)*, St.Pölten, Focus on **Information Technology**. Deparment for Electronics and Computer-engineering

Experience

Vocational

2014–Present	**IT Technician**, *Johannes Kepler University JKU*, Linz, Austria. Maintainance of server infrastructure & services as well as computer laboratories; Security incident handling; Assesment and roll out of software solutions; Technical supervision of computer examinations
2013	**Internship**, *Uninett AS - Norwegian research and education network* , Trondheim, Norway. Development of an IPv6 web application

2011–2012	**Software Developer**, *Intel Mobile Communication DMCE*, Linz, Austria. Development of technical documentation software.
<2011	**Various Internships**.

Miscellaneous

2006–2007	**Mandatory Civil Service**, *Traisen, Austria*, Ambulance driver and medic.

Languages

German	**Mothertongue**	
English	**Advanced (CEFR: C1)**	*fluent, working language*
Spanish	**Basic (CEFR: A2)**	*Routine conversations*
Norwegian	**Basic (CEFR: A2)**	*Routine conversations*

Interests

- Sport competitions: Triathlon & Marathon
- Mountaineering
- Computer games
- Cooking

Freistädter Straße – 4040 Linz, Austria
(0043) 676 6903583 • *martin@hengstberger.net*

SWORN DECLARATION

I hereby declare under oath that the submitted Master's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, July 2016

Martin Hengstberger