DIPLOMARBEIT

STEGANOGRAPHISCHE VERFAHREN ZUR VERSTECKTEN DATENSPEICHERUNG

ausgeführt zum Zweck der Erlangung des akademischen Grades eines
„Diplom-Ingenieurs für technisch-wissenschaftliche Berufe“
am Masterstudiengang Telekommunikation und Medien
der Fachhochschule St. Pölten

unter der Erstbetreuung von
Univ.Doz. Dipl.-Ing. Dr.tech. Ernst Piller

Zweitbegutachtung von
Ing. Mag. Helmut Kaufmann MAS

ausgeführt von
Jürgen Wurzer, BSc
tm071071

St. Pölten, 1. März 2009

Ehrenwörtliche Erklärung

Ich versichere, dass

- ich diese Diplomarbeit selbständig verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und mich auch sonst keiner unerlaubten Hilfe bedient habe.
- ich dieses Diplomarbeitsthema bisher weder im Inland noch im Ausland einem Begutachter/einer Begutachterin zur Beurteilung oder in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Diese Arbeit stimmt mit der von den Begutachtern beurteilten Arbeit überein.

Ort, Datum *Unterschrift*

Kurzfassung

In dieser Arbeit werden Verfahren und die dazu benötigten Protokolle designt und erforscht, die eine allgemeine und standardisierte Möglichkeit bieten, Daten mit Hilfe der Steganographie auf einem bzw. mehreren Covers (Trägermedium) zu speichern. Das Verfahren soll eine Abstraktionsschicht zu bestehenden oder auch zu neuen Steganographie-Technologien bilden. Das bedeutet, dass es auf der Steganographie-Technologie aufsetzen soll und somit eine leicht anwendbare und technologieunabhängige Methode bietet, um Daten steganographisch zu speichern. Des Weiteren wird geregelt, auf welche Art und Weise der Stego-Schlüssel aufbewahrt bzw. gespeichert wird. Ziel dieser Arbeit ist es, durch ein standardisiertes Verfahren die Interoperabilität zwischen Programmen, die steganographische Technologien zur Speicherung von Daten verwenden, zu gewährleisten bzw. zu ermöglichen.

Abstract

The goal of this research is to design and explore a method which stores information into one or more covers with the help of steganography. This technique should standardize the distribution of the hidden information on one or more covers. The method should build an abstraction layer on top of the underlying steganography technology. This steganography technology should be replaceable without making changes to the overlying method. Which means, that the new method provides a technology independent practice for saving data. Furthermore the best practical way to store the stego-key will be researched. The final goal of this thesis is to provide a standardized technique to guarantee the interoperability between programs which use steganography technologies to store information.

Danksagung

In erster Linie möchte ich diese Arbeit meinen Eltern widmen, die mich während meiner Studienzeit immer unterstützten. Vor allem möchte ich mich dafür bedanken, dass sie mir während des Studiums immer finanziell zur Seite standen.

Mein ganz besonderer Dank geht auch an meinen Bruder, der dieselbe Studiengangsrichtung wie ich absolviert und daher immer für spannende technische Diskussionen zu haben ist.

Weiters möchte ich mich bei meinem Diplomarbeits-Betreuer Ernst Piller bedanken, der mir auf der Suche nach Lösungswegen immer Rede und Antwort stand.

Ein herzliches Dankeschön geht auch an alle Studienkollegen und Freunde, die den Studiumsalltag so richtig lebenswert machten.

Jürgen Wurzer
März 2009

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **x**

1 Einleitung **1**
1.1 Steganographie . . . 2
1.1.1 Pure Steganography . . . 4
1.1.2 Secret Key Steganography . . . 4
1.1.3 Public Key Steganography . . . 5
1.2 Einbettung . . . 6
1.2.1 Methoden zum Speichern einer geänderten versteckten Nachricht . . . 8
1.2.2 Maximale Einbettungsgröße . . . 9
1.3 Überblick zum Konzept der Arbeit . . . 10
1.3.1 Verteilte versteckte Datei, Verzeichnisse, Dateisysteme . . . 11
1.4 Physische Datenträger und Sektoren . . . 12
1.5 Dateisysteme und Clusters . . . 13
1.6 Forschungsleitende Fragestellungen . . . 14
1.7 Kurzbeschreibung der Kapiteln . . . 14

2 Speicherung der versteckten Datei in verteilte Blöcke **16**
2.1 Variable und fixe Blockgröße . . . 16
2.2 Block-Referenzen . . . 17
2.3 Dateioperationen . . . 18
2.4 Buffering der versteckten Datei . . . 19
2.5 Zugriffsproblematik der versteckten Datei . . . 20
2.6 Indextabelle für Blockreferenzen . . . 21

3 Detailiertes Konzept zur Speicherung **23**
3.1 Datensätze, Relationales Datenbankmodell . . . 25
3.1.1 Directory-Tabelle . . . 26
3.1.2 StorageFile- und StorageDirectory-Tabelle . . . 28
3.1.3 KryptoOptions und StegoOptions-Tabelle . . . 31
3.1.4 DistributedFile- und BlockGroup-Tabelle . . . 33
3.2 ASN.1-Stukturen . . . 35
3.2.1 DistributedFile Struktur . . . 38
3.2.2 Directory Struktur . . . 39
3.2.3 StegoOptions Struktur . . . 39

3.2.4 KryptoOptions Struktur . . . 39
3.2.5 HashData Struktur . . . 40
3.2.6 StorageFile Struktur . . . 40
3.2.7 StoredBlocks Struktur . . . 41
3.2.8 NextEntries Struktur . . . 42
3.3 Möglichkeiten durch dieses Konzept . . . 43
3.3.1 Versteckte Speicherung mit Hilfe der Steganography . . . 43
3.3.2 Externe Speicherung auf Webaccounts als Backuplösung . . . 43
3.3.3 Externe Speicherung auf Webaccounts zur versteckten Übertragung . . . 43
3.4 Sicherheit dieses Konzepts . . . 44

4 Distributed File Storage (DFS) 45
4.1 Verwaltung und Konfiguration eines DistributedFiles . . . 46
4.2 DUSE . . . 47
4.3 File Operation Protocol (FOP) . . . 47
4.3.1 FOP Sessions . . . 48
4.3.2 Aushandlung der FOP Version (Übergang Phase 1 zu 2) . . . 48
4.3.3 Open Request (Anfrage vom Client zum Server) . . . 49
4.3.4 Open Response (Antwort vom Server zum Client) . . . 51
4.3.5 Close Request (Anfrage vom Client zum Server) . . . 52
4.3.6 Close Response (Antwort vom Server zum Client) . . . 53
4.3.7 Read Request (Anfrage vom Client zum Server) . . . 53
4.3.8 Read Response (Antwort vom Server zum Client) . . . 54
4.3.9 Write Request (Anfrage vom Client zum Server) . . . 55
4.3.10 Write Response (Antwort vom Server zum Client) . . . 56
4.3.11 Resize Request (Anfrage vom Client zum Server) . . . 56
4.3.12 Resize Response (Antwort vom Server zum Client) . . . 57

5 Problem des ersten Schlüssels 58
5.1 Parametersatz für das erste StorageFile . . . 58
5.2 Chipkarte, RFID . . . 59
5.3 Steganographische Speicherung des Parametersatzes . . . 60
5.4 Aufbewahrung des Parametersatzes im Internet . . . 61
5.4.1 HTTPS und Authentifizierung . . . 61
5.4.2 Steganographische Speicherung in Foren, Free-Webspaces, etc. 62

6 Möglicher Existenzbeweis durch Forensik 63
6.1 Zeitstempel einer Datei . . . 63
6.2 Existenz des steganographischen Programms . . . 64

7 Fazit und Ausblick 66
7.1 Standardisiertes Verfahren . . . 66
7.2 Sichere Aufbewahrung des Parametersatzes . . . 67
7.3 Beseitigung von forensischen Spuren . . . 67
7.4 Ausblick . . . 67

A ASN.1 Definitionen **68**

Literaturverzeichnis **71**

Abbildungsverzeichnis

1.1 Pure Steganography 4
1.2 Secret Key Steganography 4
1.3 Public Key Steganography 5
1.4 Cover, Stego-Objekt, Differenzbild x50 7
1.5 Test für mehrfaches Überschreiben 7
1.6 Maximale Einbettungsgröße 10
1.7 Prinzipielle Funktionsweise 11
1.8 Steganographisches Dateisystem 12

2.1 Variable und fixe Blockgröße 17
2.2 Einfach verkettete Liste von Blöcken und Blockgruppen 18
2.3 Indextabelle zur Verwaltung der Datenblöcke 21

3.1 Verteilte Datei (DistributedFile) gespeichert in mehreren StorageFiles 23
3.2 Möglicher Verwendungszweck der einzelnen StorageFiles 24
3.3 Relationales Datenbankmodell 25
3.4 Pfad-Rekonstruktion aus Directory-Tabelle 28
3.5 Objektdarstellung eines StorageFiles 31
3.6 Beispiel BlockGroup-Tabelle 35
3.7 Gespeicherte Struktur-Instanzen und Datenblöcke 37
3.8 Beispiel StoredBlocks Struktur 42

4.1 Überblick: Distributed File Storage 45
4.2 FOP Versionsaushandlung 49
4.3 Anfrage zum Öffnen einer Datei 49
4.4 Antwort vom Öffnen einer Datei 51
4.5 Anfrage zum Schließen einer Datei 52
4.6 Antwort vom Schließen einer Datei 53
4.7 Anfrage zum Lesen einer Datei 53
4.8 Antwort vom Lesen einer Datei 54
4.9 Anfrage zum Schreiben in eine Datei 55
4.10 Antwort vom Schreiben in eine Datei 56
4.11 Anfrage zum Ändern der Größe einer Datei 56
4.12 Antwort der Größenänderung einer Datei 57

Tabellenverzeichnis

1.1 Voreingestellte Clustergrößen für FAT32-Datenträger 13
1.2 Voreingestellte Clustergrößen für NTFS-Datenträger 14

3.1 Beispiel für StorageFile-Dateinamen 27
3.2 Beispiel Directory-Tabelle 27
3.3 Beispiel StorageFile-Datensatz 30
3.4 Beispiel StorageDirectory-Datensätze 31

Kapitel 1

Einleitung

Wäre es nicht wunderbar seine persönlich kritischen Daten, wie beispielsweise Gesundheitsdaten, Firmengeheimisse, Passwörter, Source Code von wichtigen Programmen oder seinen kritischen Bericht in diktatorischen Ländern, der nicht in falsche Hände geraten soll, auf einfachste Art und Weise sicher und unauffindbar auf seinem eigenen PC speichern zu können?

Hier stellt sich gleich die Gegenfrage, ob die Kryptographie nicht genau dieses Problem löst. Jedoch die Kryptographie verhindert nur den unbefugten Zugriff zum Lesen der Daten, verschleiert jedoch nicht die Existenz von Daten.

Wieso kann die Existenz von Daten problematisch werden?

Erstens wenn verschlüsselte Daten gefunden werden, stellt sich sofort die Frage wieso die Daten geheim gehalten werden und somit existiert auch gleichzeitig ein Anreiz die Daten entschlüsseln zu wollen. Zweitens ist man in einigen Ländern dazu verpflichtet den Schlüssel bei einer Beschlagnahmung seines Laptops preis zu geben und in anderen Ländern ist Kryptographie erst gar nicht erlaubt und somit die Verwendung strafbar.

Die Lösung des Problems bietet die Steganographie. Mit Hilfe der Steganographie soll das einfache und problemlose Speichern und Bearbeiten von versteckten Daten realisiert werden. Da diese Arbeit auf bestehende steganographische Technologien aufbaut, wird zuerst ein kurzer Überblick über Steganographie und auf deren relevanten Eigenschaften eingegangen.

1.1 Steganographie

Steganographie ist die Kunst Informationen verborgen zu speichern oder zu übertragen. Dabei wird die geheime Information in einem Trägermedium (Cover) versteckt. Die Existenz der geheimen Nachricht darf für eine außenstehende dritte Person nicht erkennbar sein. Ein steganographisches Verfahren zum Verstecken von Informationen gilt als unsicher bzw. geknackt, wenn die Existenz der geheimen Informationen bewiesen werden kann. Ob die extrahierten geheimen Informationen gelesen bzw. entschlüsselt werden können, ist dabei nur von sekundärer Bedeutung.

Das berühmteste steganographische Beispiel führt zurück in die Antike. 499 v. Chr. rasierte Histiaios seinem vertrauenswürdigsten Sklaven die Haare und ließ ihm eine Nachricht auf dem Kopf tätowieren. Anschließend wartete er, bis die Haare des Sklaven nachgewachsen waren und schickte ihn zu Aristagoras, der dem Sklaven die Haare erneut rasierte um die geheime Nachricht zu lesen. (vgl. Church, 2009)

Eine weitere einfache steganografische Methode wurde im zweiten Weltkrieg von einem deutschen Spion verwendet, um eine versteckte Nachricht zu verschicken. Er verwendete dazu folgende Null-Cipher Nachricht (vgl. Vitaliev, 2007):

> A**p**parently n**e**utral's p**r**otest i**s** t**h**oroughly d**i**scounted a**n**d i**g**nored. **I**sman h**a**rd h**i**t. B**l**ockade i**s**sue a**f**fects p**r**etext f**o**r e**m**bargo o**n** b**y**-products, e**j**ecting s**u**ets a**n**d v**e**getable o**i**ls.

Zum Dekodieren der geheimen Nachricht muss von jedem Wort der zweite Buchstabe extrahiert werden. Anschließend werden die Buchstaben aneinander gereiht und zusätzlich Leerzeichen an den richtigen Stellen eingefügt. Die versteckte übermittelte Nachricht lautet somit:

> Pershing sails from NY June 1.

Die Steganographie wird oft als Teilgebiet der Kryptographie eingestuft, jedoch wird sie oft auch als eigenständiger Wissenschaftsbereich angesehen. Für Letzteres spricht, dass die Steganographie und Kryptographie nicht ganz die selben Ziele verfolgen. Die Kryptographie versucht die Nachricht zu verschlüsseln, sodass sie von einem Dritten nicht gelesen werden kann. Hingegen die Steganographie versucht die Nachricht zu verstecken, sodass sie von einem Dritten erst gar nicht gefunden bzw. entdeckt werden kann.

Steganographie kann weiters als Teilbereich von *Information Hiding* eingestuft werden. Ein anderes großes Teilgebiet von Information Hiding ist Digital Watermarking. Hierbei werden mit Hilfe von steganographischen Techniken Urheberrechtshinweise

und beispielsweise eindeutige Seriennummern versteckt, um bei Urheberrechtsverletzungen auf den Täter schließen zu können.

Information Hiding kann in folgende Punkte unterteilt werden:

- Covert Channels
- Anonymity
- Steganography
- Watermarking

Für diese Arbeit ist der Teilbereich Steganographie von Bedeutung, da für die Datenspeicherung steganographische Techniken zum Einsatz kommen sollen. In dieser Arbeit werden keine neuen Techniken erforscht bzw. erfunden, sondern sie baut auf bestehenden Techniken auf. Hierzu muss auf die relevanten Eigenschaften der steganographischen Techniken eingegangen werden.

Durch die Digitalisierung von Texten, Bildern, Videos usw. nahm auch die Steganographie Einzug in die digitale Welt. Nicht nur durch die digitalen Speichermedien, sondern auch vor allem durch die Kommunikationsmöglichkeiten per Internet wird die Verwendung von steganographischen Techniken immer attraktiver. So bietet zwar auch die Kryptographie die Möglichkeit Daten geschützt über das unsichere Internet zu übertragen, jedoch erst durch die Steganographie ist eine unentdeckbare und unauffällige Datenübertragung möglich.

Die Steganographie kann grundsätzlich in drei Typen unterteilt werden, dies sind die Pure Steganographie, die Secret Key Steganographie und die Public Key Steganographie, wobei die Letztere auf einer Public/Private Verschlüsselung basiert und nicht auf einer Public/Private steganographischen Einbettung. Um eine steganographische Methode anwenden so können, wird immer ein Trägermedium (Cover, c) und eine zu versteckende Nachricht (Message, m) benötigt. Außerdem kann je nach verwendeter Methode noch ein zusätzliches Geheimnis (Stego-Key, k) zum Einsatz kommen. Durch einen bestimmten Algorithmus wird die geheime Nachricht in das Trägermedium eingebettet. Das veränderte Trägermedium, das die geheime Nachricht beinhaltet, wird Stego-Objekt (Stego-object, s) genannt. Das Stego-Objekt unterscheidet sich nur minimal vom originalen Trägermedium. Die Änderungen sollten nicht mit den menschlichen Sinnesorganen wahrgenommen werden können. Des Weiteren sollte es nicht möglich sein, nur anhand des Stego-Objekts mithilfe von statistischen Methoden, die Existenz der geheimen Nachricht zu beweisen. Damit eine gute steganographische Einbettung möglich ist, muss die zu einbettende Nachricht um ein vielfaches kleiner sein als das eigentliche Trägermedium. Oft wird vor der Einbettung noch die Nachricht komprimiert und verschlüsselt, um die zu einbettenden Daten klein zu halten und eine Streuung bzw. Permutation der Nachricht zu erhalten. Diese Zusatzmaßnahmen erschweren das Auffinden eines Steganogramms. (vgl. Katzenbeisser und Petitcolas, 2000, S. 1-11) (vgl. Johnson und Jajodia, 1998)

1.1.1 Pure Steganography

Bei der Puren Steganographie wird kein zusätzliches Geheimnis verwendet. Das bedeutet, dass kein Stego-Schlüssel zum Einsatz kommt. Durch diesen Umstand steckt die Sicherheit dieses Verfahren einzig und alleine im Einbettungsalgorithmus. Daher sollte der Algorithmus geheim bleiben und nur dem Sender und Empfänger bekannt sein. Abbildung 1.1 zeigt links die Einbettung einer Nachricht in ein Cover, das zu einem Stego-Object resultiert. Rechts in der Abbildung ist das Extrahieren der Nachricht dargestellt.

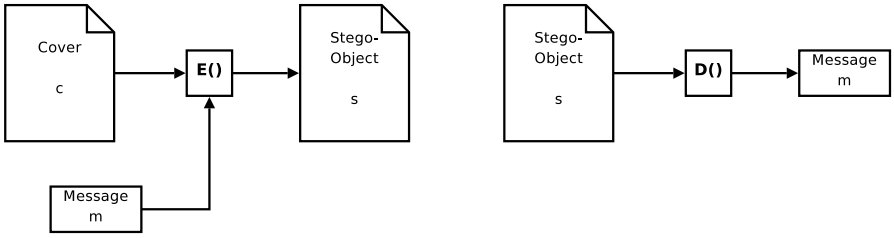

Abbildung 1.1: Pure Steganography

(vgl. Katzenbeisser und Petitcolas, 2000, S. 20)

1.1.2 Secret Key Steganography

Bei der Secret Key Steganographie wird zusätzlich ein Stego-Schlüssel verwendet. Die Sicherheit beruht alleine auf dem Stego-Schlüssel. Ohne Stego-Schlüssel sollte es nicht möglich sein, die Nachricht aus dem Stego-Objekt extrahieren zu können, beziehungsweise deren Existenz beweisen zu können. Dies entspricht dem Kerckhoffs' Prinzip, wo die Sicherheit einzig und alleine im Schlüssel liegen darf und nicht in der Geheimhaltung des Algorithmus. Wenn bei dieser Methode ein unbefugter Dritter den Algorithmus kennt, jedoch nicht den passenden Schlüssel, so darf es nicht möglich sein, die Nachricht zu extrahieren. Die Abbildung 1.2 zeigt den Ablauf der Secret Key Steganographie.

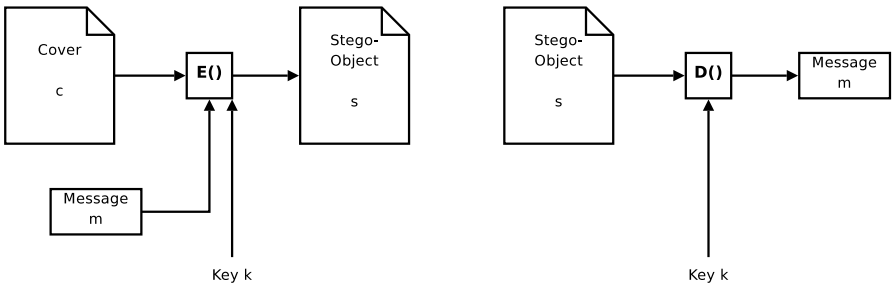

Abbildung 1.2: Secret Key Steganography

(vgl. Katzenbeisser und Petitcolas, 2000, S. 22)

1.1.3 Public Key Steganography

Bei der Public Key Steganography wird grundsätzlich die Pure Steganographie mit einer zusätzlichen Public/Private Verschlüsselung der Nachricht verwendet. Die Sicherheit (das Nicht-Auffinden der Nachricht) beruht bei dieser Methode darauf, dass eine verschlüsselte Nachricht sich im Idealfall bzw. fast nicht von dem ”Rauschen” eines leeren Stego-Objekts unterscheidet, zumindest wird diese Eigenschaft vorausgesetzt. Wird zum Beispiel bei einem nicht infizierten Bild versucht eine steganographische Nachricht zu extrahieren, so erhält man eine Nachricht, die einem zufälligen Bytestream entspricht. Dieser Bytestream ist im Idealfall jedoch nicht von einem Bytestream, der eine verschlüsselte Nachricht enthält, unterscheidbar. Somit darf sogar der steganographische Algorithmus, der keinen Stego-Schlüssel benötigt, für einen unbefugten Dritten verfügbar sein, da er die extrahierte Nachricht nicht zwischen einer sinnvollen verschüsselten Nachricht und keiner sinnvollen Nachricht unterscheiden kann. Im konkreten Fall kann diese Methode dazu verwendet werden, um einen Stego-Schlüssel für die anschließende Secret Key Steganographie auszutauschen. Um eine Nachricht per Public Key Steganographie versenden zu können, muss Bob zuerst ein Public/Private Schlüsselpaar erstellen. Anschließend muss dafür gesorgt werden, dass Alice den öffentlichen Public Key unverändert erhält. Wenn Alice den Public Key besitzt, kann sie beispielsweise den zu übertragenden Stego-Schlüssel mit dem Public Key verschüsseln. Durch die asymmetrische Verschlüsselung ist gewährleistet, dass nur Bob in der Lage ist den zu verwendeten Stego-Schlüssel mit Hilfe des Private Keys zu extrahieren. Die Abbildung 1.3 zeigt den Ablauf einer Public Key Steganographie. Eine zweite Möglichkeit ein asymmetrisches steganographisches Verfahren zu entwickeln wäre ein Verfahren, das auf zwei unterschiedliche Stego-Schlüssel aufbaut. Für die steganographische Einbettung und für die Extraktion muss dann jeweils der andere Stego-Schlüssel verwendet werden. Jedoch existiert derzeit kein solches Verfahren.

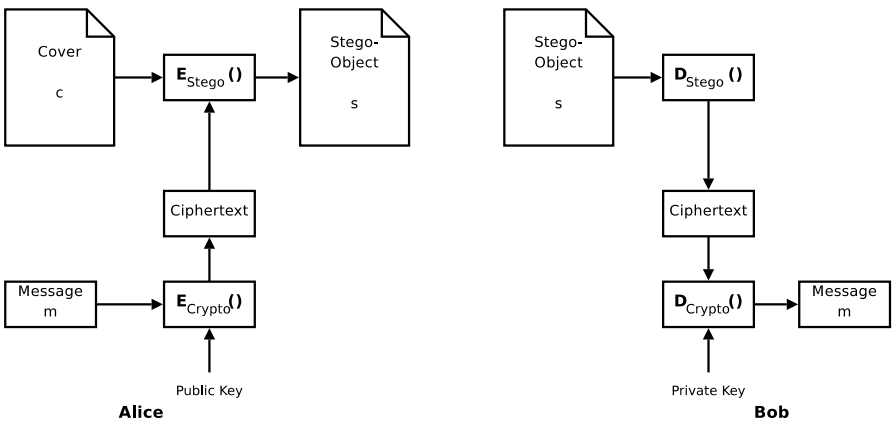

Abbildung 1.3: Public Key Steganography

(vgl. Katzenbeisser und Petitcolas, 2000, S. 23 ff)

1.2 Einbettung

Für die Einbettung der Nachricht existieren verschiedene Einbettungsalgorithmen. Die meisten Algorithmen sind stark vom Trägermedium abhängig beziehungsweise nur für bestimmte Covertypen geeignet. So existieren beispielsweise spezielle Algorithmen zum Einbetten von Nachrichten in JPEG Bilddateien. Weiters existieren Algorithem für Windows Bitmap (BMP), Graphics Interchange Format (GIF), MPEG-1 Audio Layer 3 (MP3) und vielen anderen Dateitypen, die als Cover zur Einbettung verwendet werden können.

Welche Bits der Algorithmus im Cover Material verändert, beziehungsweise welche Bits zum Speichern der Nachricht ausgewählt werden, ist bei den meisten Algorithmen maßgeblich durch folgende Eigenschaften bestimmt.

- Inhalt des Covers
- Inhalt der Nachricht
- Stego-Schlüssel

Zur Veranschaulichung sind einige Tests mit den Programmen *steghide* und *outguess* durchgeführt worden. Speziell werden die Auswirkungen auf das mehrfache Überschreiben eines Covers bzw. Stego-Objekt getestet.

Bei den Tests werden gleich lange Nachrichten mit unterschiedlichem Inhalt in jeweils einem Cover eingebettet. Somit wird getestet, in wieweit der Inhalt der zu versteckenden Nachricht die Auswahl der zu manipulierenden Coverdaten beeinflusst.

Als Cover dient eine Jpeg-Bilddatei mit einer Auflösung von 3264x2448 Bildpunkten und einer Dateigröße von 1837712 Bytes (entspricht ca. 1,75 MB). Die einzubettende Nachricht beträgt mit 18569 Bytes ca. 1% der Cover-Dateigröße. Als Nachricht stehen zwei verschiedene Binärdateien zu Verfügung. Die zwei Dateien besitzen einmal nur Nullen (00000000b) und einmal nur Einser (11111111b) als Bitwerte pro Byte.

Für die Tests wird das Programm Steghide Version 0.5.1 verwendet, bei dem die Komprimierung, Verschlüsselung der Nachricht und Einbettung des Dateinamens für die Tests deaktiviert werden. Zuerst wird mit diesen Optionen die Eigenschaften des Stego-Objekts und des Differenzbildes erforscht. Im nächsten Schritt wird die Auswirkung des mehrfachen Einbettens in ein bereits existierendes Stego-Objekt untersucht.

Die Abbildung 1.4 zeigt die Differenz zwischen dem Cover und dem Stego-Objekt, das als Nachricht die Datei mit den Nullen eingebettet besitzt. Die Farbe Schwarz bedeutet bei dem Differenzbild keine Änderung. Damit die Änderungen erkennbar sind, ist der Kontrast um den Faktor 50 erhöht. Insgesamt sind im Stego-Objekt 21.5% der Bildpunkte, bezogen auf das Originalcover, leicht verändert. Wäre der Kontrast

nicht um den Faktor 50 verstärkt, so würde das Differenzbild als schwarze Fläche wargenommen werden.

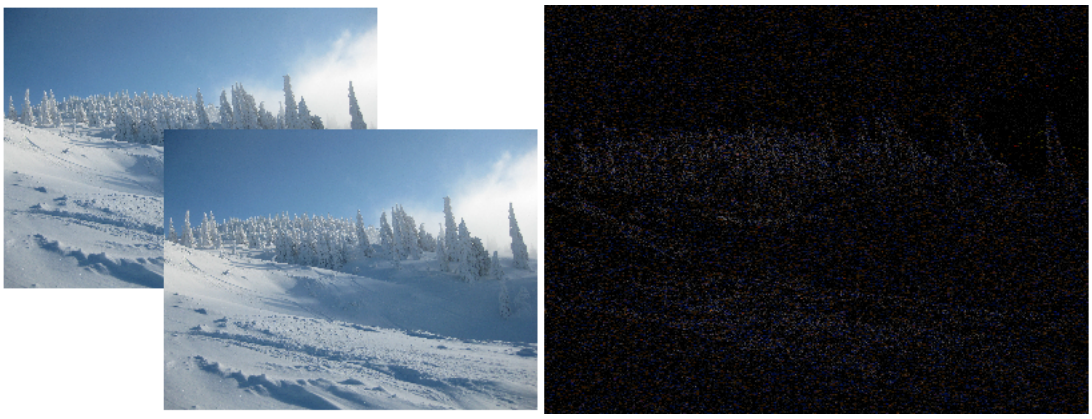

Abbildung 1.4: Cover, Stego-Objekt, Differenzbild x50

Der Test, wie in Abbildung 1.5 dargestellt, kann das Verhalten bei mehrfachen Überschreiben bzw. Einbetten in ein Stego-Objekt getestet werden. Wenn sich die Stego-Objekte s1 und s3 nicht unterscheiden, so bedeutet das, dass die eingebettete Nachricht des Stego-Objekts s2 durch eine neue Nachricht ersetzt werden kann. Solch ein steganographischer Einbettungs-Algorithmus dürfte den Einbettungspfad nur anhand des Stego-Schlüssels unabhängig von der Nachricht und den Inhalt des Covers bestimmen. Jedoch weisen die Einbettungs-Algorithmen im Normalfall dieses Verhalten nicht auf, denn wenn der Inhalt der Nachricht und des Covers berücksichtigt werden, können effektivere Einbettungen vorgenommen werden.

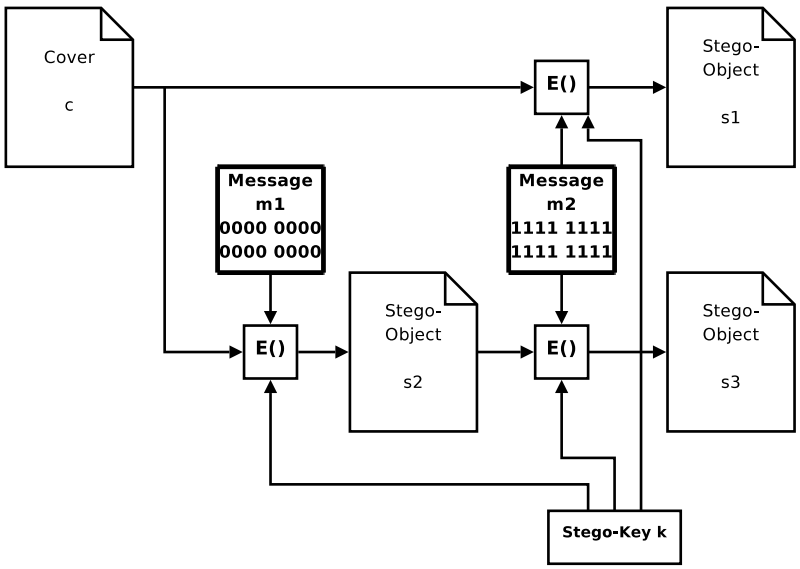

Abbildung 1.5: Test für mehrfaches Überschreiben

Ein Test mit den Programmen Steghide und Outguess laut Abbildung 1.5 führt sehr

wohl zu Differenzen zwischen den Stego-Objekten s1 und s3. Das bedeutet, dass bei jeder weiteren eingebetteten Nachricht sich die Artefakte des Stego-Objekts bezogen zum Original Cover immer stärker auswirken. Die fortlaufende Verschlechterung des Stego-Objekts bei wiederholter Einbettung bedeutet, dass eine andere Lösung für das Bearbeiten beziehungsweise Überschreiben einer steganographischen Nachricht gefunden werden muss.

1.2.1 Methoden zum Speichern einer geänderten versteckten Nachricht

Für das Speichern einer geänderten steganographischen Nachricht können verschiedene Methoden angewendet werden, die unterschiedlich dafür geeignet sind. Bei diesen Methoden wird angenommen, dass in einem bereits verwendeten Cover (aktuelles Stego-Objekt) keine neue Nachricht wegen der Verschlechterung des Stego-Objekts eingebettet werden darf, denn dadurch wird eine erfolgreiche Steganalyse leichter möglich.

1.2.1.1 Geänderte Nachricht im Original Cover einbetten

Die einfachste Möglichkeit wäre das Stego-Objekt mit der alten Nachricht zu verwerfen und das originale Cover, das für die Einbettung verwendet wurde, wieder für die neue Nachricht zu verwenden. Das Problem bei dieser Methode ist, dass das Stego-Objekt und das Cover verfügbar sein müssen. Das Stego-Objekt muss für das Auslesen der Nachricht und das Cover für eine eventuelle neue einzubettende Nachricht vorhanden sein. Durch diesen Umstand ist der Zugriff auf das Cover und auf das Stego-Objekt möglich und somit kann durch ein einfaches Differenzbild die Verwendung von Steganographie nachgewiesen werden. Aus diesem Grund sollte immer das Cover vernichtet werden beziehungsweise die Cover-Datei zum Stego-Objekt überschrieben werden.

1.2.1.2 Geänderte Nachricht in einem neuen Cover einbetten

Eine weitere Möglichkeit wäre es, die geänderte Nachricht in einem neuen Cover einzubetten. Das jeweilige Programm besitzt zum Beispiel eine Vielzahl von Covers. Wenn nun eine Nachricht in einem Stego-Objekt geändert werden soll, dann wird das Stego-Objekt verworfen und die geänderte Nachricht wird in einem neuen Cover gespeichert. Durch diese Methode ist auch kein Differenzbild für den Nachweis möglich, selbst wenn zum Beispiel mit forensischen Mitteln das alte Stego-Objekt wiederhergestellt werden kann, denn das alte und neue Stego-Objekt beruhen auf zwei unterschiedlichen Covers. Es muss nur dafür gesorgt werden, dass das vernichtete Cover auf keinen Fall wiederhergestellt werden kann. Deshalb sollte die Cover-Datei mit dem Inhalt des Stego-Objekts überschrieben werden und nicht eine eigene Datei für das Stego-Objekt angelegt werden. Der Nachteil dieser Methode ist, dass bei einer

häufigen Änderung der eingebetteten Nachricht eine hohe Anzahl von neuen Covers benötigt wird.

1.2.2 Maximale Einbettungsgröße

Die maximal verfügbare Größe, die in ein Cover eingebettet werden kann, ist von mehreren Faktoren abhängig. Unter anderem von folgenden Faktoren:

- Größe des Trägermediums (Covergröße)
- Größe des Cover-"Rauschens"
  Zum Beispiel besitzt ein stark unterbelichtetes Bild ein höheres Rauschen als ein optimal belichtetes Bild.
- Einbettungs-Algorithmus
- Robustheit des Steganogramms
  Hiermit ist gemeint, wie leicht bzw. wie schwer die steganographische Nachricht zerstört werden kann.
- Grad der Unauffindbarkeit
  Prinzipiell gilt, um so weniger Daten eingebettet werden sollen, um so schwerer ist der Nachweis per Steganalyse für ein Steganogramm zu bewerkstelligen.

Durch eine zusätzliche Komprimierung der Nachricht kann die Länge der Nachricht größer als die zu eingebetteten Daten ausfallen. Ein weiterer Faktor für die Bestimmung der maximalen Einbettungsgröße kann der Inhalt der zu einbetteten Nachricht sein. Wenn jedoch die maximale Einbettungsgröße für das jeweilige Cover bestimmmt werden soll, ohne die Nachricht zu kennen, kann das zu einem Problem führen.

Generell kann pro Cover bei Bekanntheit der oben aufgelisteten Einbettungsparameter mehr oder weniger genau die maximale Kapazität bestimmt werden. Zur einfacheren Veranschaulichung kann man sich vorstellen, dass von der Größe des Covers ein gewisser Prozentsatz bzw. Block für die Einbettung von Daten zur Verfügung steht. Die Abbildung 1.6 zeigt die Abstraktion, in der die verfügbare Größe für die Einbettung als geschlossener Block dargestellt wird. In Wirklichkeit sind natürlich die einzelnen Bits der Nachricht über das ganze Cover je nach Einbettungsverfahren verteilt.

Abbildung 1.6: Maximale Einbettungsgröße

Auf dieses Prinzip setzt die Arbeit auf. Jedes Cover stellt eine gewisse Kapazität zur Verfügung, die zum Verstecken der zu speichernden Daten verwendet werden kann. Wenn die Einbettungs-Kapazität des Covers nicht ausreicht, so sollen mehrere Covers verwendet werden, die sich die gesamte Nachricht untereinander aufteilen.

1.3 Überblick zum Konzept der Arbeit

Prinzipiell soll ein Verfahren entwickelt werden, das in der Lage ist eine beliebig große Datei in mehrere Covers aufzuteilen. Je nach Dateigröße der steganographischen Nachricht werden dann mehr oder weniger Coverdateien benötigt. Der Schwerpunkt liegt darin, dass anhand der ersten Stego-Objektdatei und dem nötigen Stego-Schlüssel alle weiteren Stego-Objektdateien gefunden werden und die extrahierten Nachrichten in der richtigen Reihenfolge zusammengestoppelt werden. Das bedeutet, dass eine Stego-Objektdatei nicht nur einen Teil der eigentlichen Nachricht beinhaltet, sondern auch eine Referenz mit den nötigen Parametern für die nächste Stego-Objektdatei. Ein weiterer Schwerpunkt der Arbeit stellt die Aufbewahrung des ersten Stego-Schlüssels dar. Es sollen auch die Möglichkeiten erforscht werden, die Steganogramme ins Internet auszulagern, indem beispielsweise Gratis-Webspaces und dergleichen verwendet werden. In diesem Punkt sollte auch eine Redundanz der Speicherung möglich sein. Hiermit ist gemeint, dass Teile der Nachricht nicht nur in einem Stego-Objekt auf einem Gratis-Webspace gespeichert sind, sondern zusätzlich bei einem anderen Anbieter. In diesem Fall sollten die Stego-Objekte über die Redundanz Bescheid wissen. Die Abbildung 1.7 zeigt einen groben Überblick über das Konzept der verteilten steganographischen Speicherung. In dieser Abbildung ist die Verkettung der Stego-Objekte als einfache lineare Liste dargestellt, jedoch werden später noch effektivere Methoden beschrieben.

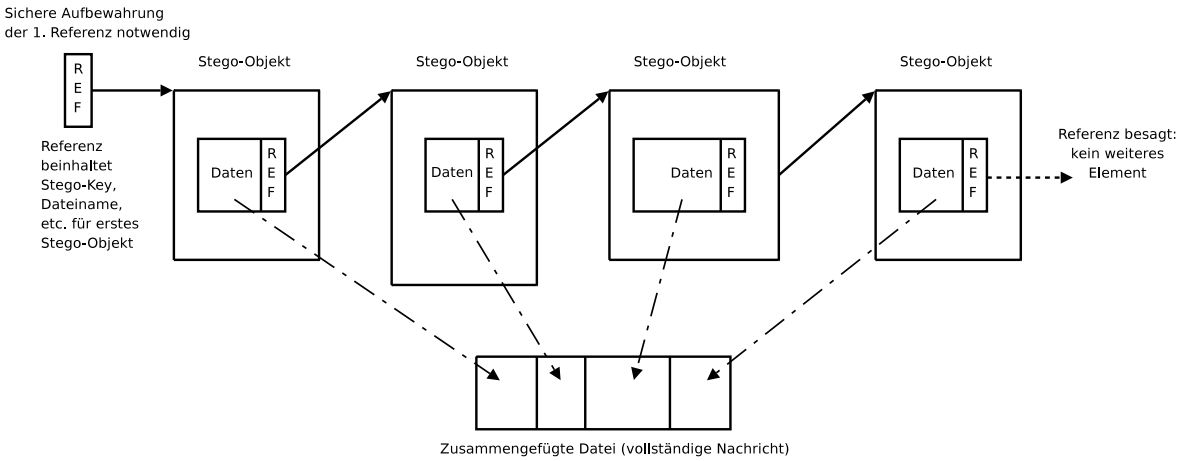

Abbildung 1.7: Prinzipielle Funktionsweise

Falls die steganographisch versteckte Datei, die in mehreren Stego-Objekten aufgeteilt ist, nicht nur gelesen sondern auch beschrieben werden soll, dann müssen weitere Punkte beachtet werden.

- Verfügbare Covers zum Austausch der geänderten Blöcke
- Keinen inkonsistenten Zustand der Daten bei Schreibvorgängen
  Vergleichbar wie bei einem Journaling-Dateisystem muss dafür gesorgt werden, dass kein inkonsistenter Zustand der Referenzen bei einem möglichen Absturz des Rechners bzw des Programms zustande kommen kann. Das bedeutet, der Austausch eines Blockes muss als Transaktion durchgeführt werden. Falls der Austausch des neuen Stego-Objekts fehlschlägt, muss die begonnene Transaktion rückgängig gemacht werden.

1.3.1 Verteilte versteckte Datei, Verzeichnisse, Dateisysteme

Die oben dargestellte Abbildung 1.7 zeigt eine Möglichkeit, die dazu verwendet werden kann, um eine Datei in einem oder mehreren Stego-Objekten zu verstecken. Wenn mehrere Dateien versteckt werden sollen, dann kann diese Methode mehrfach angewendet werden. Es muss nur dafür gesorgt werden, dass jede zu versteckende Datei seine eigenen Stego-Objekte verwendet. Jedoch besitzt diese Methode viele Einschränkungen. So werden grundsätzlich keine Ordner bzw. Verzeichnisse unterstützt. Des weiteren stehen viele Eigenschaften von Dateisystemen nicht zur Verfügung, wie beispielsweise die Eigenschaften Besitzer, Gruppe, Zugriffsrechte, ACL, usw. Prinzipiell gibt es nun zwei Möglichkeiten, die Eigenschaften eines Dateisystems zu unterstützen. Die erste Möglichkeit wäre ein eigenes steganographische Dateisystem zu entwickeln. Jedoch wäre dieses Vorhaben sehr komplex und zeitintensiv. Die zweite sehr einfache Möglichkeit ist, in der steganographisch verstecken Datei ein Dateisystem-Image zu

speichern. Auf diese Image-Datei kann mit einem speziellen Treiber zugegriffen werden. Somit würde beispielsweise unter Windows das steganographische Dateisystem als zusätzliches Laufwerk im Arbeitsplatz aufscheinen. Diese Methode ist vergleichbar mit einem Loop Device unter Linux, welches den Zugriff auf eine Datei als Block Device realisiert. Wenn dieses steganographisch versteckte Dateisystem auch noch geändert werden kann, dann können alle Funktionen eines Dateisystems unterstützt werden. Die Abbildung 1.8 zeigt die Möglichkeit eines steganographischen Dateisystems anhand einer versteckten Image-Datei.

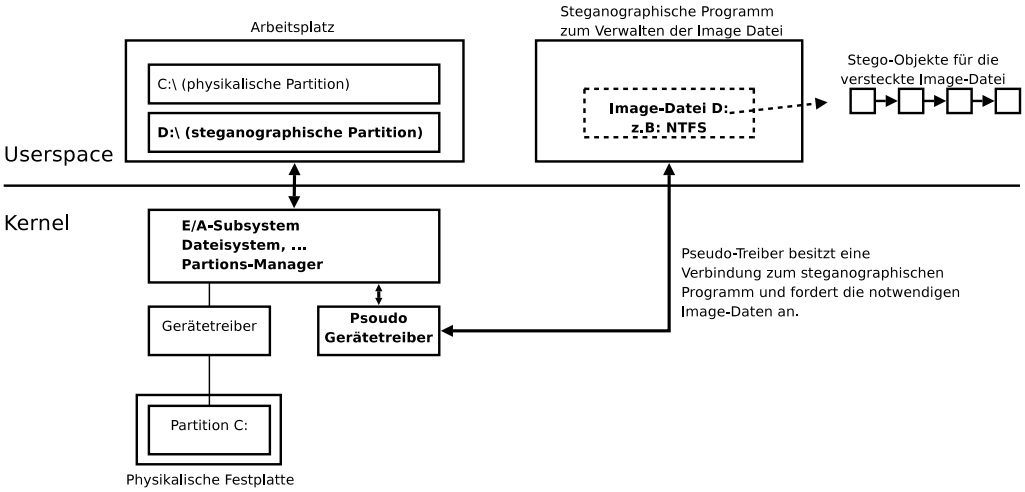

Abbildung 1.8: Steganographisches Dateisystem

Diese Arbeit konzentriert sich darauf, eine Datei verstecken und bearbeiten zu können. Zusätzlich wird ein Treiber implementiert und getestet, der ein steganographisches Dateisystem nach dem Schema wie in Abbildung 1.8 ermöglicht. Auch wenn kein eigenes Dateisystem entworfen wird, so sind trotzdem die grundsätzlichen Eigenschaften wie beispielsweise Blockgröße, Clustergröße, etc. von Relevanz, denn die Schreiboperationen einer versteckten Image-Datei sollten für Änderungen einer Blockgröße optimiert sein. Aus diesem Grund wird noch ein kleiner Einblick in Datenträger und Dateisysteme gegeben.

1.4 Physische Datenträger und Sektoren

Zu den physischen Datenträgern zählen Speichermedien wie beispielsweise CD-ROM, Festplatten (disks), Disketten, USB-Sticks usw. Die meisten Datenträger werden in Sektoren unterteilt. Bei einem Sektor wiederum handelt es sich um einen adressierbaren Block mit fixer Größe, welche üblicherweise vom Datenträger abhängig ist. Bei Festplatten beträgt die Größe eines Sektors im Normalfall 512 Byte und bei CD-ROM-Sektoren 2048 Byte. Ein Sektor ist die kleinste Einheit, die von einem bzw. auf einen

Datenträger gelesen oder geschrieben werden kann. Jeder Sektor besitzt seine eigene Speicheradresse und ist somit unabhängig von seinen anderen Sektoren adressierbar. Ein Datenträger besteht aus vielen zusammenhängenden Sektoren. Der Datenträger kann in eine oder mehrere Partitionen unterteilt sein. Eine Partition wiederum besteht aus mehreren zusammenhängenden Sektoren. Der Startsektor und die Größe der einzelnen Partitionen stehen in der Partitionstabelle. Bei BIOS-basierten Computern der x86-Architektur befindet sich die Partitionstabelle im Master Boot Record (MBR), welcher im ersten Sektor des Datenträgers zu finden ist und 512 Byte groß ist. (vgl. Russinovich und Solomon, 2005, S.593 ff)

1.5 Dateisysteme und Clusters

Ein Dateisystem ist durch sein Dateisystemformat definiert. Dieses bestimmt, auf welche Art und Weise Dateien auf einem Speichermedium zu speichern sind und legt alle weiteren Eigenschaften des Dateisystems fest. Wenn beispielswiese das Dateisystemformat keine Benutzerberechtigungen vorsieht, so kann das Dateisystem keine Sicherheitsfunktionen für Zugriffsberechtigungen unterstützen. Unter Windows werden folgende Dateisystemformate unterstützt: CDFS, UDF, FAT12, FAT16, FAT32 und NTFS. Oft verwendete Dateisystemformate unter Linux sind beispielsweise Ext2, Ext3, ReiserFS. In anderen unixoden Betriebssystemen finden Dateisystemformate wie ZFS, XFS, JFS ihre Verwendung. Das Lesen bzw. Schreiben der Daten der jeweiligen Datei erfolgt blockweise. Die Blockgröße ist durch das Dateiformat bestimmt. Solch ein Block wird im Windows-Sprachgebrauch Cluster bezeichnet. Im Unix-Umfeld ist hingegen der Terminus Block gebräuchlich. Jedoch ist ein Block des Dateisystems (Cluster) nicht mit einem Block des Datenträgers (Sektor) gleichzusetzen. Die Clustergröße ist immer ein Vielfaches der Sektorgröße. Ein Cluster ist die kleinste adressierbare Einheit eines Dateisystems, die gelesen oder geschrieben werden kann. Die Clustergröße des Datiesystems wird beim Formatieren anhand der Dateisystemgröße automatisch festgelegt, jedoch kann sie gegebenenfalls auch manuell bestimmt werden. Die Tabellen 1.1 und 1.2 zeigen die möglichen Clustergrößen für die Dateisystemformate FAT32 und NTFS. (vgl. Russinovich und Solomon, 2005, S.667 ff) (vgl. Stallings, 2003, S.605 ff)

Größe der Partition	Clustergröße
32 MB bis 8 GB	4 KB
8 GB bis 16 GB	8 KB
16 GB bis 32 GB	16 KB
32 GB	32 KB

Tabelle 1.1: Voreingestellte Clustergrößen für FAT32-Datenträger

Größe der Partition	Clustergröße
512 MB oder weniger	512 Byte
513 MB bis 1024 MB (1GB)	1 KB
1024 MB bis 2048 MB (2 GB)	2 KB
Größer als 2 GB	4 KB

Tabelle 1.2: Voreingestellte Clustergrößen für NTFS-Datenträger

1.6 Forschungsleitende Fragestellungen

In dieser Arbeit werden folgende Fragen behandelt:

- *Inwieweit kann ein standardisiertes Verfahren unabhängig von der verwendeten Technologie entworfen werden?*
- *Welche Möglichkeiten gibt es, einen oder mehrere Stego-Schlüssel sicher bzw. ebenfalls versteckt aufzubewahren?*
- *Gibt es eine Möglichkeit, die Stego-Software selbst zu verstecken/verschleiern um den Verdacht auf steganographische Datenspeicherung zu entgehen?*
- *Welche anderen Faktoren müssen bedacht werden, die eine steganographische Speicherung, unabhängig von deren verwendeten Technologie, verraten könnte?*

1.7 Kurzbeschreibung der Kapiteln

Zuerst werden die Möglichkeiten zum ”verteilten” Speichern einer Datei beschrieben. Hier werden mögliche Konzepte zum Speichern der verteilten Blöcke, die insgesamt die vollständige Datei bilden, erläutert und deren Stärken und Schwächen dargestellt. Dieser Punkt wird im Kapitel 2 behandelt.

Im Kapitel 3 wird ein detailiertes Konzept zur verteilten Speicherung vorgestellt. Dies beinhaltet unter anderem auch die notwendigen ASN.1-Strukturen, die für die Verwaltung der verteilten Datei benötigt werden.

Im Kapitel 4 wird ein konkretes Verfahren zum Speichern der zu versteckenden Datei erklärt. Dies beinhaltet sowohl das FOP Protokoll zum Kommunizieren zwischen Client und Server als auch das Distributed File Storage, welches die steganographisch versteckten Dateien verwaltet.

Das Kapitel 5 behandelt die Problematik der sicheren Aufbewahrung des ersten Schlüssels beziehungsweise der ersten Referenz, die den Schlüssel, den Pfad des ersten Stego-Objekts und noch weitere Informationen beinhaltet.

Im Kapitel 6 werden mögliche Schwächen aufgezeigt, die es einem Forensiker erlauben, die Existenz des steganographischen Verfahrens aufzudecken.

Kapitel 2

Speicherung der versteckten Datei in verteilte Blöcke

Damit eine Datei steganografisch gespeichert werden kann, muss zuerst die verfügbare Einbettungsgröße bestimmt werden. Wenn die Datei diese Größe überschreitet, werden mehrere Covers zum Einbetten benötigt. Die Datei muss aus diesem Grund in mehrere Blöcke unterteilt werden. Beispielsweise könnte eine Datei in drei Blöcke, deren Blockgrößen variieren, unterteilt werden und in drei unterschiedliche Covers eingebettet werden. Des Weiteren müssen Zusatzinformationen gespeichert werden, die die Reihenfolge und alle weiteren Informationen wie zum Beispiel den Stego-Schlüssel für die Extraktion des nächsten Stego-Objekts beinhalten. Ein Problem entsteht jedoch, wenn die verfügbare Einbettungsgröße zu klein für die Zusatzinformationen, die sich auf das nächste Stego-Objekt beziehen, ist. In diesem Fall kann das Cover nicht verwendet werden. Somit sollte jedes Cover, das verwendet werden soll, eine Mindestkapazität für die Einbettungsdaten besitzen.

2.1 Variable und fixe Blockgröße

Die Blöcke, in denen die Datei unterteilt wird, können in variable Längen oder in fixe Längen unterteilt werden. Beide Methoden weisen Vor- und Nachteile auf. Wenn zum Beispiel eine Datei wie in Abbildung 2.1 in drei Covers mit unterschiedlich verfügbaren Einbettungs-Kapazitäten aufgeteilt werden soll, dann bietet im ersten Augenblick die variable Blocklänge die ideale Lösung. Durch die variable Länge werden alle Bytes verwendet, die in einem Cover eingebettet werden können. Der Nachteil der variablen Länge ist, dass alle Referenzen und Eigenschaften, die sich auf die eingebetteten Daten beziehen wie beispielsweise Offset, Länge, Größe in Bytes angegeben werden müssen. Dadurch entsteht ein größerer Overhead als bei einer fixen Blockgröße. Der Nachteil bei einer fixen Blockgröße ist, dass nicht immer alle verfügbaren Kapazitäten bis auf das letzte Byte ausgenützt werden können. Wird beispielsweise die fixe Blockgröße

auf 512 Byte festgelegt, dann kann die Größe der Verschwendung von 1 bis 511 Byte pro Cover betragen. Die Referenzen und Adressierungen können bei einer fixen Größe blockorientiert angegeben werden. Dadurch vergrößert sich der adressierbare Bereich gegenüber der variablen Blocklänge um den Faktor der fixen Blockgröße. Wird zum Beispiel für die Eigenschaften von Länge und Offset ein zwei Byte großes Datenfeld verwendet, dann kann bei einer variablen Länge durch die byteorientierte Adressierung maximal ein Adressraum von 64 KB angesprochen werden. Wenn eine fixe Blockgröße verwendet wird, dann erfolgt die Adressierung blockorientiert und somit können in diesem Fall 65536 Blöcke angesprochen werden. Bei einer fixen Blockgröße von 512 Byte ergibt das einen maximalen Speicherbereich beziehungsweise eine maximale Dateigröße von 32768 KB (32 MB).

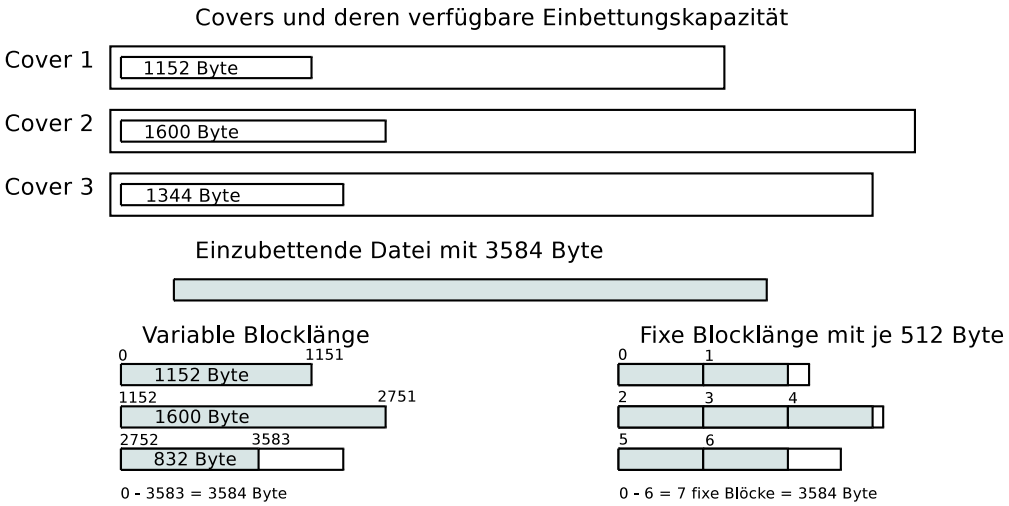

Abbildung 2.1: Variable und fixe Blockgröße

Durch das Festlegen der fixen Blockgröße auf 1 Byte entsprechen die Eigenschaften bezüglich maximalen Adressbereich und effizienter Speichernutzung den variablen Blockgrößen. In dieser Arbeit wird von einer fixen Blockgröße pro versteckter Datei ausgegangen. Dadurch, dass die fixe Blockgröße auch auf 1 gesetzt werden kann, bietet die fixe Blockgröße die meisten Vorteile.

2.2 Block-Referenzen

Für das Zusammenfügen der einzelnen Blöcke zu einer vollständigen Datei muss die Reihenfolge der Blöcke bekannt sein. Zum Sichern der Reihenfolge gibt es mehrere Möglichkeiten. Die einfachste Möglichkeit, die richtige Reihenfolge der Blöcke zu gewährleisten, bietet eine einfach verkettete Liste. Hier werden die notwendigen Informationen für das Finden und Extrahieren des nächsten Blocks bzw der nächsten Blockgruppe in einer Referenz (Verweis) gespeichert. Jeder Block bzw. jede Blockgruppe er-

hält zum Schluss eine Referenz, die auf den nächsten Block verweist. In der Referenz müssen alle notwendigen Informationen gespeichert werden, um die Daten-Blöcke aus dem Stego-Objekt extrahieren zu können. Die anschließende Liste beschreibt die notwendigen Informationen.

- **Dateiname und Pfad des nächsten Stego-Objekts**
- Checksumme des Stego-Objekts, um die Integrität der Daten zu gewährleisten
- **Einbettungsalgorithmus** bzw. Name und Version des steganographischen Programms zur Extraktion
- **Stego-Schlüssel**
- Verschlüsselung (Algorithmus, Modus)
- Verschlüsselungsschlüssel
- Anzahl der gespeicherten Blöcke
- Checksumme der gespeicherten Blöcke

Die erste Referenz muss sicher aufbewahrt werden. Sie ist indirekt der Schlüssel für alle verwendeten Stego-Objekte bzw. für die versteckte Datei. Die Abbildung 2.2 zeigt die Verwendung einer einfach verketteten Liste.

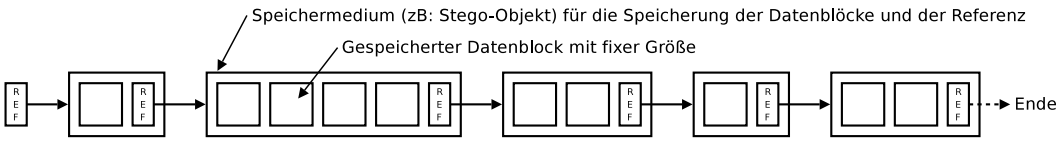

Abbildung 2.2: Einfach verkettete Liste von Blöcken und Blockgruppen

Wenn alle Stego-Objekte wie in Abbildung 2.2 einfach hintereinander verkettet sind, dann können auch die gespeicherten Datenblöcke nur sequentiell ausgelesen werden. Das bedeutet, falls der letzte Datenblock ausgelesen werden soll, müssen zuerst alle anderen Stego-Objekte extrahiert werden. Somit ist dieses Modell bzw. Konzept sehr gut für das sequentielle Auslesen einer Datei geeignet, jedoch nicht für das Auslesen von einzelnen Blöcken der Datei. Damit ein geeignetes Modell entworfen werden kann, werden zuerst die Operationen, die eine Datei unterstützen sollte, erklärt.

2.3 Dateioperationen

Abgesehen vom Öffnen und Schließen der Datei werden zum Lesen, Schreiben und Bearbeiten des Dateiinhaltes bei POSIX kompatiblen Betriebssystemen folgende Punkte unterstützt:

- Lesen, beginnend von der aktuellen Position (*read*)
- Schreiben, beginnend bei der aktuellen Position (*write*)
- Neupositionierung des aktuellen Schreib-/Lesezeigers (*lseek*)
- Dateigröße verändern (*truncate*, bzw. durch *write* am Ende der Datei)
  Dies beinhaltet sowohl Vergrößern als auch Verkleinern

Die POSIX-konformen Systemaufrufe für die jeweiligen Punkte sind *read*, *write*, *lseek* und *truncate*. Die aufgezählten Punkte gelten auch für die Dateibearbeitung unter Windows. Hierfür stehen equivalente Windows-API Funktionen zur Verfügung.

Die Möglichkeit, einzelne Bytes einer Datei am Anfang oder zwischen Dateianfang und -ende zu löschen bzw einzufügen, wird von den meisten Dateisystemen nicht unterstützt. Für diese Operation existiert auch kein POSIX-konformer Systemaufruf. Konkret bedeutet das, dass bei der steganographisch versteckten Datei die Möglichkeit, einzelne Bytes bzw. Blöcke am Dateianfang oder zwischen Anfang und Ende zu löschen bzw. einzufügen, nicht unterstützt werden muss. Es muss nur das Anfügen von Daten am Dateiende unterstützt werden. Wenn bei einer Datei eines Dateisystems von Windows oder Linux ein oder mehrere Bytes vom Dateianfang gelöscht werden sollen, muss der ganze Dateiinhalt um die jeweilige Byteanzahl nach vorne kopiert werden und anschießend die Datei um die notwendige Anzahl verkleinert werden.

Die verkettete Liste laut Abbildung 2.2 weist einige Probleme bei der Unterstützung der aufgezählten Dateioperationen auf. Das größte Problem ist die Neupositionierung des Lese/Schreibzeigers, der die aktuelle Position festhält, speziell wenn die aktuelle Dateiposition nach vorne verschoben werden soll. Wenn in diesem Fall das vorherige Stego-Objekt benötigt wird, weil die notwendigen Datenblocks sich darin befinden, dann müssen wieder, beginnend mit der ersten Referenz, alle Stego-Objekte bis zum angeforderten Stego-Objekt durchgearbeitet werden. Im anderen Fall, wenn die aktuelle Position nach hinten verschoben wird, ist die verkettete Liste auch nicht ideal. Es kann zwar nicht der Fall auftreten, dass von der ersten Referenz alle Stego-Objekte durchgearbeitet werden müssen, jedoch kann es passieren, dass der neue Offset auf einen Datenblock verweist, der sich nicht im nächsten, sondern beispielsweise im übernächsten Stego-Objekt befindet. In diesem Fall muss trotzdem das Stego-Objekt dazwischen ausgelesen und extrahiert werden, um die notwendige Referenz für das richtige Stego-Objekt zu erhalten.

2.4 Buffering der versteckten Datei

Eine Problemlösung der Datei-Neupositionierung könnte das Buffern der versteckten Datei im Arbeitsspeicher darstellen. Dies bedeutet, dass das steganographische

Programm mit Hilfe der ersten Referenz alle Datenblöcke aus den Stego-Objekten nacheinander extrahiert und im Arbeitsspeicher zu einer Datei zusammenfügt. Der notwendige Arbeitsspeicher bei dieser Methode sollte kein Problem darstellen, denn eine Speicherreservierung bis zu 100 MB ist für heutige Rechner, die bis zu einigen Gigabytes an Arbeitsspeicher verfügen, leicht zu bewerkstelligen. Trotzdem sollte der Fall der Speicherauslagerung berücksichtigt werden, wo die Daten der versteckten Datei unverschlüsselt auf der Festplatte (z.B: Swap-Partition) gespeichert werden. Eine versteckte Datei von 100 MB ist somit nicht unmöglich. Wenn beispielsweise angenommen wird, dass in einer Coverdatei Daten bis zu einem Prozent der Covergröße versteckt werden können, dann werden für eine steganographisch versteckte Datei von 100 MB insgesamt ca. 10 GB an Coverdaten benötigt. Ist die versteckte Datei im Arbeitsspeicher geladen, bereiten die Dateioperationen bis auf das Schreiben keine Schwierigkeiten mehr. Das Ändern der Daten im Arbeitsspeicher ist beim Schreibvorgang nicht die Problematik, sondern das schlussendliche Speichern der geänderten Daten in den Stego-Objekten. Denn für die geänderten Datenblöcke müssen die jeweiligen Stego-Objekte, wie in der Einleitung beschrieben (siehe 1.2.1), durch neue ersetzt werden. Jedoch kann bei einer einfach verketteten Liste der Stego-Objekte nicht ein Stego-Objekt durch ein neues ersetzt werden. Der Grund dafür liegt darin, dass bei einem Austausch eines Stego-Objekts die Referenz, die im vorderen Stego-Objekt gespeichert ist, für das neue Stego-Objekt nicht mehr gültig ist und somit auch das vordere Objekt ausgetauscht werden muss. Ist das der Fall, obwohl sich hier kein Datenblock geändert hat, sondern nur die Referenz, dann muss durch den Austausch wiederum das vordere Stego-Objekt ersetzt werden. Schlussendlich zieht sich dieses Szenario bis zur ersten Referenz hindurch. Somit ist das einfache Modell mit der Verkettung der Stego-Objekte für die Speicherung und Bearbeitung der versteckten Datei nicht geeignet.

2.5 Zugriffsproblematik der versteckten Datei

Unabhängig davon, ob die versteckte Datei im Arbeitsspeicher abgelegt wird oder ob die einzelnen Datenblöcke bei den Leseoperationen jedes Mal neu vom Stego-Objekt extrahiert werden, darf ein anderer Prozess nicht auf die extrahierten Datenblöcke zugreifen. Somit kann das steganographische Programm die Datenblöcke aus den Stego-Objekten zwar extrahieren, jedoch anderen Programmen für deren Lese- und Schreiboperationen, nicht zur Verfügung stellen. Die einfachste Möglichkeit, die versteckte Datei einem anderen Programm zur Verfügung zu stellen, wäre die Datei auf der Festplatte des Betriebssystems zu speichern. Jedoch ist dieser Vorgang aus Sicht eines Forensikers nicht vertretbar, denn selbst wenn nach dem Gebrauch der Datei, diese wieder gelöscht wird, kann deren Inhalt im Normalfall rekonstruiert werden. Dieser Umstand beruht darauf, dass beim Löschen einer Datei nur der Eintrag im Dateiverzeichnis, jedoch nicht der eigentliche Dateiinhalt gelöscht wird. Der eigentliche Dateiinhalt wird erst wieder überschrieben, wenn das Dateisystem den Speicher für eine neue Datei verwendet. Unter Linux existiert für diesen Fall eine eigene Option für das Dateisystem, das Linux dazu veranlasst, den Inhalt beim Löschen einer Datei zu überschreiben. Die

elegante Lösung wäre für die Extraktion der versteckten Datei eine eigene RAM-Disk zu verwenden. Somit können andere Programme problemlos auf die Datei zugreifen, obwohl die Daten nicht auf der Festplatte gespeichert werden. Eine weitere elegante Möglichkeit bietet die Realisierung eines steganographisch versteckten Dateisystems laut Abbildung 1.8 (Abbildung in der Einleitung). Dies ermöglicht ebenfalls den einfachen Zugriff von externen Programmen.

2.6 Indextabelle für Blockreferenzen

Als Alternative zur einfach verketteten Stego-Objekt-Liste kann eine Indextabelle für die Blockreferenzen verwendet werden. Die Indextabelle enthält für jede Blockgruppe die jeweilig passende Referenz zum Stego-Objekt und als Index für diese Referenz dient die Blocknummer. Soll beispielsweise der sechste Block ausgelesen werden, dann besitzt der Block den Index 5 und die notwendige Referenz zum benötigten Stego-Objekt kann mittels der Indexnummer in der Indextabelle gefunden werden. Die Indextabelle selbst ist in einem oder mehreren Stego-Objekten gespeichert. Die erste Referenz, die nicht steganographisch gespeichert ist, verweist auf das Stego-Objekt, welches die Indextabelle besitzt. Die Abbildung 2.3 zeigt die Verwendung einer Indextabelle, die alle Referenzen für die Stego-Objekte verwaltet.

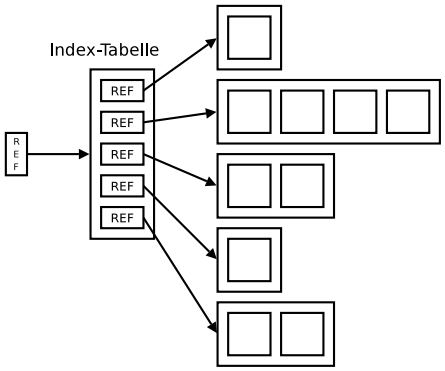

Abbildung 2.3: Indextabelle zur Verwaltung der Datenblöcke

Die Verwendung der Indextabelle zum Verwalten der Referenzen für die Datenblöcke bietet folgende Vorteile gegenüber der einfachen Verkettung der Stego-Objekte:

- Die Indextabelle ermöglicht den wahlfreien Zugriff auf die Datenblöcke ohne zusätzliche Stego-Objekte extrahieren zu müssen. Falls die Stego-Objekte auf einem Webspace gespeichert sind, brauchen die nicht benötigten Stego-Objekte nicht extra heruntergeladen werden. Durch die Indextabelle ist eine Neupositionierung des aktuellen Lese/Schreibzeigers unproblematisch.
- Die Unterstützung einer Schreiboperation ist möglich.
  Beim Austausch eines Stego-Objekts, das Datenblöcke besitzt, müssen keine weiteren Stego-Objekte, die eingebettete Datenblöcke besitzen, ersetzt werden.

Nur das Stego-Objekt mit der Indextabelle und die erste Referenz müssen eventuell ersetzt werden.

- Beim Verlust eines Stego-Objekts, das steganographisch gespeicherte Datenblöcke besitzt, sind nicht automatisch die nachfolgenden Datenblöcke der anderen Stego-Objekte zerstört.

Kapitel 3

Detailiertes Konzept zur Speicherung

In diesem Kapitel wird ein Konzept zur verteilten Speicherung einer Datei in einzelne Datenblöcke beschrieben. Die Abbildung 3.1 zeigt vereinfacht dargestellt die Grundfunktion des Konzepts.

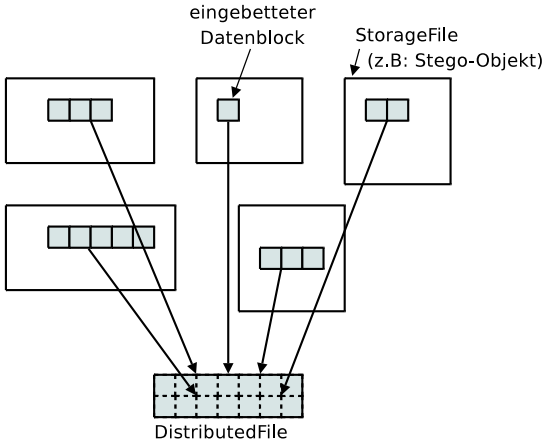

Abbildung 3.1: Verteilte Datei (DistributedFile) gespeichert in mehreren StorageFiles

Die verteilte speicherte Datei wird im weiteren Verlauf *DistributedFile*, im Sinne von ”verteilte Datei”, genannt. Die einzelnen Datenblöcke, aus denen das DistributedFile zusammengesetzt wird, sind in einzelnen *StorageFiles* gespeichert. Wenn die Datenblöcke des DistributedFile in mehreren Stego-Objekten gespeichert sind, dann entspricht das jeweilige StorageFile jeweils einem Stego-Objekt. Jedoch ist dieses Konzept zur verteilten Speicherung einer Datei (DistributedFile), nicht nur auf die Verwendung von Covers bzw. Stego-Objekte beschränkt, daher wird das Medium, in dem die einzelnen Datenblöcke gespeichert werden, StorageFile (”Speicherdatei”) genannt.

Als **StorageFile** sind folgende Dateien möglich:

- Unverschlüsselte Binärdatei
- Verschlüsselte Datei mit Hilfe der Kryptographie
- **Stego-Objekt**-Datei mit Hilfe der Steganographie

Als Basis für dieses Konzept dient das Modell mit der zentralen Indextabelle (siehe Kapitel 2, Abbildung 2.3). Damit das Programm, das die verteilte Datei verwaltet, die Indextabelle erstellen kann, werden mehrere Datensätze in einigen StorageFiles gespeichert. Die einzelnen Datensätze werden durch ASN.1-Strukturen repräsentiert und mit Hilfe der Basic Encoding Rules (BER) kodiert und in den StorageFiles gespeichert. Ein StorageFile kann zum Speichern eines oder mehrerer Datensätze oder zum Speichern der Datenblöcke, die den eigentlichen Inhalt der verteilten Datei (DistributedFile) beinhalten, verwendet werden.

Damit Änderungen an der verteilten Datei vorgenommen werden können, muss dem Programm immer ein Pool bzw. eine Menge an nicht benutzten StorageFiles zur Verfügung stehen. Jedes dieser verfügbaren StorageFiles kann vom Programm zur Speicherung von ASN.1-Strukturen oder zur Speicherung von Datenblöcken verwendet werden.

Ein StorageFile kann somit folgende Zustände besitzen:

- Verfügbar für eine Speicherung
- Verwendet - beinhaltet eine oder mehrere ASN.1 Strukturen
- Verwendet - beinhaltet Datenblöcke der verteilten Datei (DistributedFile)

Die Abbildung 3.2 zeigt den Zusammenhang der einzelnen StorageFiles.

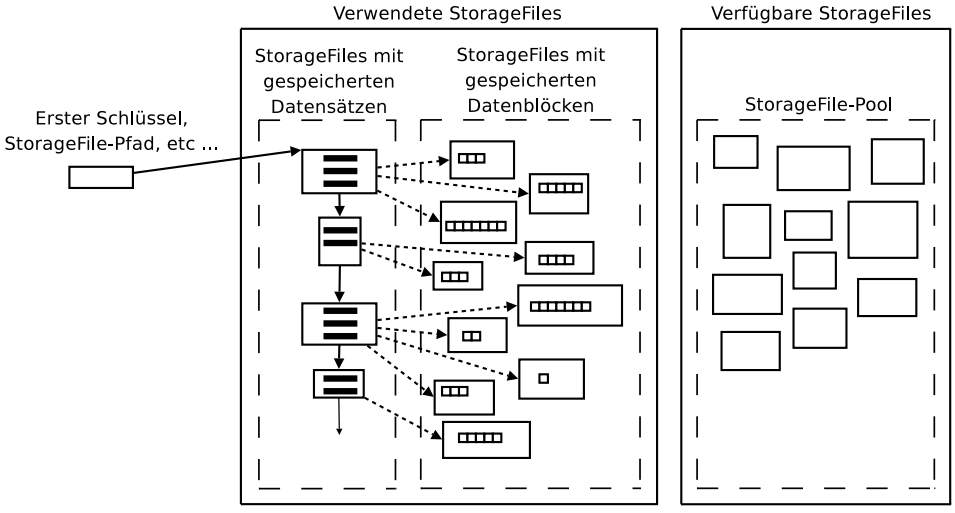

Abbildung 3.2: Möglicher Verwendungszweck der einzelnen StorageFiles

3.1 Datensätze, Relationales Datenbankmodell

Mithilfe eines relationalen Datenbankmodells können alle notwendigen Informationen zur verteilten Speicherung einer Datei verwaltet werden. Anhand der ausgelesenen ASN.1-Strukturen, die in den StorageFiles gespeichert sind, können die notwendigen Datensätze und Tabellen erstellt werden. In welcher Reihenfolge die einzelnen ASN.1-Strukturen gespeichert werden müssen und wie das Programm anhand

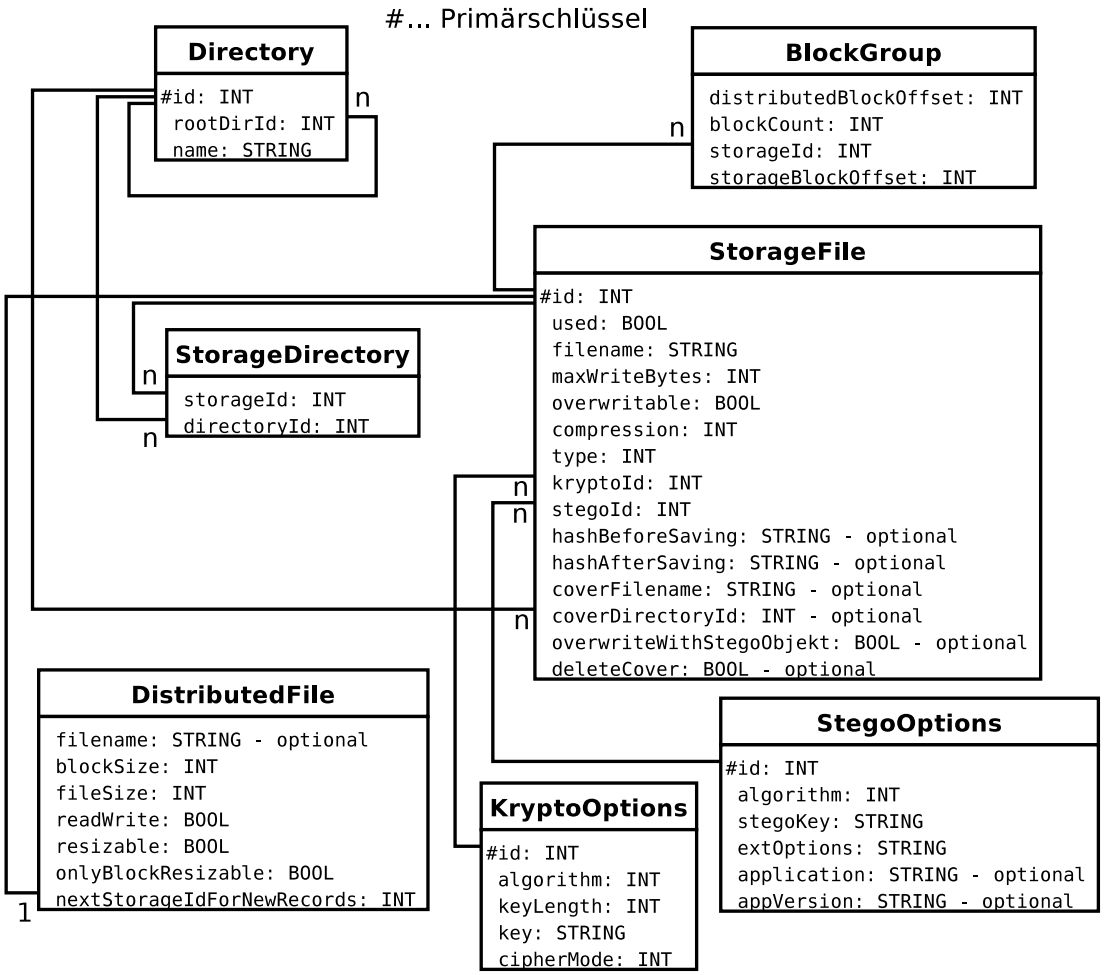

Abbildung 3.3: Relationales Datenbankmodell

der ASN.1-Strukturen die Tabellen mit den Datensätzen generieren kann, wird später in diesem Kapitel behandelt. Zuerst werden die Tabellen und deren Beziehungen des Datenbankmodells genau erläutert. Die Abbildung 3.3 zeigt das notwendige relationale Datenbankmodell für die verteilte Speicherung einer Datei. Wie schlussendlich die ASN.1-Strukturen vom Programm verarbeitet werden, hängt von der Implementierung des Programms ab. So muss beispielsweise das Programm keine richtige relationale Datenbank implementieren bzw. verwenden, sondern kann zum Beispiel die Tabellen mit Hilfe von dynamischen Datenstrukturen realisieren. Als dynamische Datenstruk-

turen könnten lineare Listen (einfach oder doppelt verkettet) und binäre Bäume deren Verwendung finden. Des Weiteren eignen sich fertige Container-Strukturen. So bietet die Standard Template Library (STL), die als Bibliothek in der Programmiersprache C++ zur Verfügung steht, unter anderem die Container *vector*, *deque*, *list*, *set*, *multiset*, *map* und *multimap* an. Einen guten Überblick zu dynamischen Daten- und Container-Strukturen bietet das Buch *Softwaretechnik in C und C++* von Rolf Isernhagen und Hartmut Helmke (siehe Isernhagen und Helmke, 2004). Auch wenn schlussendlich das Programm die Informationen der gespeicherten ASN.1-Strukturen nicht in eine richtige relationale Datenbank speichert, so eignet sich dieses Modell trotzdem zur Erklärung der einzelnen Beziehungen und Referenzen der Daten.

Im ersten Augenblick erscheint dieses Modell eventuell zu komplex, jedoch bietet dieses relationale Datenbankmodell viele Möglichkeiten und Vorteile. Durch diese Anordnung werden folgende Funktionen unterstützt:

- Wahlfreien Zugriff auf die Datenblöcke
- Bearbeiten der verteilten Datei (DistributedFile)
  Dies gilt auch, wenn es sich bei den StorageFiles um StegoObjekte handelt.
- Redundante Speicherung
  Alle gespeicherten Datenblöcke und ASN.1-Strukturen können problemlos redundant gesichert werden. Werden beispielsweise die Stego-Objekte, die als StorageFiles dienen, auf mehreren Gratis-Web-Accounts redundant gespeichert, dann führt der Verlust eines Accounts nicht zum Verlust der verteilt gespeicherten Datei.
- Konsistenter Zustand bei Änderungen
  Durch die richtige Reihenfolge beim Speichern der StorageFiles kann kein inkonsistenter Zustand auftreten.

Zum besseren Verständnis folgt nun eine genaue Beschreibung der einzelnen Tabellen und welchen Nutzen sie erfüllen.

3.1.1 Directory-Tabelle

In der Directory-Tabelle werden alle notwendigen Verzeichnisse gespeichert. Aus diesen Einträgen wird jeweils der vollständige Pfad zusammen gestoppelt. Die Alternative zur Directory-Tabelle wäre, den vollständigen Pfad direkt in den notwendigen Tabellen wie beispielsweise in der StorageFile-Tabelle zu speichern. Jedoch würde dies zu einer unerwünschten redundanten Speicherung von Informationen führen. Denn es ist davon auszugehen, dass sich mehrere StorageFiles im selben Verzeichnis befinden oder sich eventuell im vollständigen Pfad in einem Unterverzeichnis unterscheiden. Zum Beispiel könnte der vollständige Dateiname von mehreren StorageFiles, wie in der Tabelle 3.1 aufgelistet, aussehen. Wenn nun für jedes StorageFile eigens der Pfad in der StorageFile-Tabelle abgespeichert wird, dann würde in jedem Eintrag der Tabelle entweder *C:\Daten\Bilder\Urlaub\Gastein*, *C:\Daten\Bilder\Urlaub\Podersdorf* oder

Vollständiger Dateiname (inkl. Pfad)
C:\Daten\Bilder\Urlaub\Gastein\Pic034.jpg
C:\Daten\Bilder\Urlaub\Gastein\Pic035.jpg
C:\Daten\Bilder\Urlaub\Gastein\Pic038.jpg
C:\Daten\Bilder\Urlaub\Gastein\Pic041.jpg
C:\Daten\Bilder\Urlaub\Podersdorf\Bild12.jpg
C:\Daten\Bilder\Urlaub\Podersdorf\Bild14.jpg
C:\Daten\Bilder\Urlaub\Podersdorf\Bild15.jpg
C:\Daten\Video\Snowboard\Jump.mpg

Tabelle 3.1: Beispiel für StorageFile-Dateinamen

C:\Daten\Video\Snowboard gespeichert werden. Die Directory-Tabelle verhindert genau diese redundante Speicherung. In der Directory-Tabelle wird immer nur der Name eines Verzeichnisses und nicht der vollständige Pfad gespeichert. Zusätzlich beinhaltet der jeweilige Datensatz die ID (rootDirId) des übergeordneten Verzeichnisses und eine eigene eindeutige Identifikation. Zum Speichern der drei oben angeführten Pfade wären die Einträge der Tabelle 3.2 notwendig. Durch die Datensätze der Tabelle

id (Schlüssel)	rootDirId	name
1	0	windows
2	1	C:
3	2	Daten
4	3	Bilder
5	4	Urlaub
6	5	Gastein
7	5	Podersdorf
8	3	Video
9	8	Snowboard

Tabelle 3.2: Beispiel Directory-Tabelle

3.2 können wieder die drei oben angeführten Pfade hergestellt werden. Die anderen Tabellen müssen für einen bestimmten Pfad nur mehr auf die richtige ID referenzieren. Beispielsweise für den Pfad, der mit Podersdorf endet, muss die ID 7 verwendet werden. Die Abbildung 3.4 zeigt die Pfade, die durch die einzelnen Datensätze gebildet werden können. Weiters stellt die Abbildung Pfade für HTTP, FTP und unixartige Betriebssysteme dar.

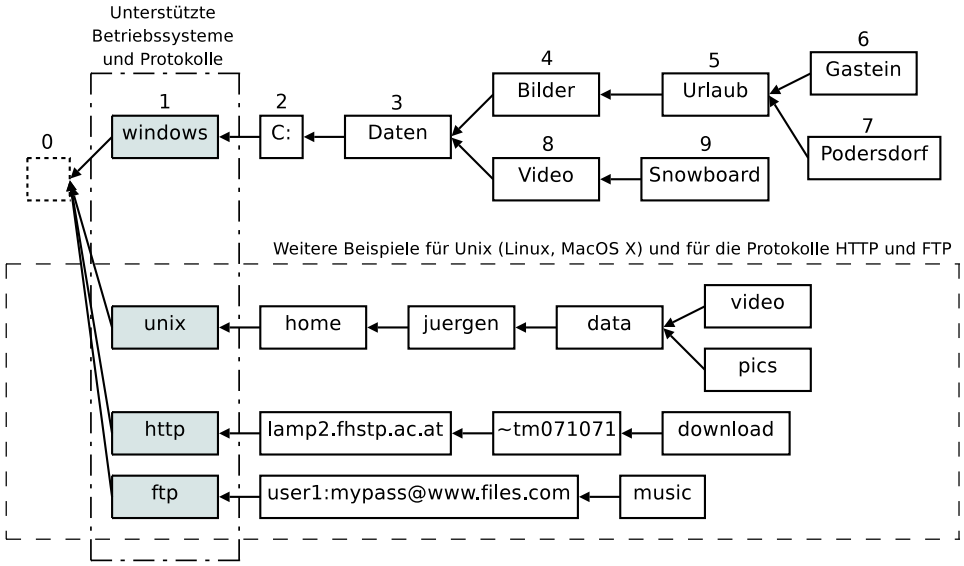

Abbildung 3.4: Pfad-Rekonstruktion aus Directory-Tabelle

Die Datensätze, die kein Verzeichnis darstellen, sondern das Betriebssystem beziehungsweise das Protokoll angeben, können durch den Wert von rootDirId unterschieden werden, denn diese Datensätze müssen für rootDirId immer den Wert 0 gespeichert haben. Somit können auch problemlos die Namen windows, linux, http und ftp für Verzeichnisse verwendet werden.

3.1.2 StorageFile- und StorageDirectory-Tabelle

In den beiden Tabellen StorageFile und StorageDirectory sind die Informationen zu jedem StorageFile gespeichert. Die Tabelle StorageFile besitzt folgende Datenfelder:

- **id** (Primärschlüssel)
  Jedes StorageFile besitzt eine eindeutige Identifikation, jedoch muss es sich bei einem StorageFile nicht genau um eine Datei handeln. Ein StorageFile kann mehrmals redundant unter verschiedenen Pfaden abgespeichert sein. Diese redundant gespeicherten Dateien repräsentieren ein StorageFile und haben somit dieselbe Identifikationsnummer.

- **used**
  zeigt, ob das StorageFile bereits für eine Speicherung verwendet worden ist. Wenn das Attribut *used* auf TRUE gesetzt ist, darf das StorageFile nicht für eine erneute Speicherung verwendet werden, außer das Attribut *overwritable* ist ebenfalls auf TRUE gesetzt.

- **filename**
  speichert den Dateinamen des StorageFiles. Dies beinhaltet jedoch nicht den Pfad, sondern nur den Dateinamen an sich.

- **maxWriteBytes**
  gibt an, wieviele Bytes für die Speicherung von Datenblöcke oder für die Spei-

cherung von ASN.1-Strukturen zu Verfügung stehen. Wenn es sich bei dem StorageFile um ein Stego-Objekt (bzw. vor der Einbettung Cover) handelt, gibt *maxWriteBytes* die maximale Größe der eingebetteten Daten an. Bei einem Binärdatei oder bei einer verschlüsselten Datei entspricht *maxWriteBytes* gleich der maximalen Dateigröße.

- **overwritable**
  bestimmt, ob das StorageFile mehrmals überschrieben werden darf. Wenn es sich bei dem StorageFile um ein Stego-Objekt handelt, ist dieses Attribut üblicherweise auf FALSE zu setzen. Bei einer Binär- oder verschlüsselten Datei, kann dieses Attribut auf TRUE gesetzt werden. Das hat zu Folge, dass bei der Verwendung von Binär- und verschlüsselte Dateien als StorageFiles kein zusätzlicher Pool von nicht verwendeten StorageFiles benötigt wird, sofern die Größe des DistributedFile nicht erweitert wird.

- **compression**
  gibt die zu verwendende Komprimierung an. Der Wert 0 bedeutet keine Komprimierung.

- **type**
  bestimmt, um welchen Typ von StorageFile es sich handelt, in dem die Datenblöcke bzw. ASN.1-Strukturen gespeichert sind. Es werden folgende Typen unterstützt:

  - Binärdatei (0x01)
  - Verschlüsselte Datei (0x02)
  - Stego-Objekt (0x03)

- **kryptoId**
  ist die Referenz zu den kryptographischen Informationen, die zum Lesen des StorageFiles benötigt werden. Besitzt das Attribut *kryptoId* den Wert 0, dann ist bei diesem StorageFile keine Verschlüsselung verwendet worden.

- **stegoId**
  ist die Referenz zu den steganographischen Informationen, die zum Lesen des StorageFiles benötigt werden. Besitzt das Attribut *stegoId* den Wert 0, dann handelt es sich bei diesem StorageFile nicht um ein Stego-Objekt.

- **hashBeforeSaving** (optional)
  Dieses Datenfeld wird nur dann benötigt, wenn das StorageFile vor der Speicherung schon Daten beinhaltet. Dies ist üblicherweise der Fall, wenn es sich bei dem StorageFile um ein Cover handelt. Anhand dieses Hashwertes kann überprüft werden, ob es sich bei diesem StorageFile noch um ein Cover handelt, oder ob schon Daten eingebettet worden sind und somit das StorageFile ein Stego-Objekt ist.

- **hashAfterSaving** (optional)
  ist der Hashwert nach dem Speichern bzw. Anlegen des StorageFiles. Hiermit kann die Integrität des StorageFiles überprüft werden.

- **coverFilename**, **coverDirectoryId**, **overwriteWithStegoObjekt**, **deleteCover** (opt.)
  Diese Datenfelder sind optional und werden eventuell nur dann benötigt, wenn es sich bei dem StorageFile um den Typ Stego-Objekt handelt. Im Normalfall wird das StorageFile, das vor der Speicherung dem Cover entspricht, durch das Stego-Objekt ersetzt. Wenn jedoch das StorageFile als Stego-Objekt neu angelegt werden soll und als Cover eine unabhängige Datei verwendet werden soll, dann müssen diese optionalen Datenfelder verwendet werden.

  - **coverFilename**
    Dateiname des Covers
  - **coverDirectoryId**
    Referenz zum verwendeten Pfad
  - **overwriteWithStegoObjekt**
    Gibt an, ob zusätzlich diese Coverdatei auch als Stego-Objekt überschrieben werden soll, damit ein Differenzbild zwischen dem Storage-File und dem Cover nicht möglich ist.
  - **deleteCover**
    Wenn dieses Datenfeld auf TRUE gesetzt ist, dann wird nach dem Anlegen und Speichern des StorageFiles das Cover gelöscht.

Eine wichtige Information kann aus der StorageFile-Tabelle nicht entnommen werden, dies ist der Speicherpfad des StorageFiles. Diese Information ist in die Tabelle StorageDirectory ausgelagert. Dadurch können einem StorageFile mehrere Speicherpfade zugeordnet werden und somit wird eine redundante Speicherung eines StorageFiles ermöglicht.

Die Tabelle StorageFile besteht aus den Datenfeldern *storageId* und *directoryId*.

- **storageId**
  referenziert auf den gewünschten Datensatz in der StorageFile-Tabelle.
- **directoryId**
  referenziert auf die Directory-Tabelle und bestimmt somit den zu verwendenden Pfad.

Ein Datensatz für ein StorageFile vom Typ Stego-Objekt könnte wie in Tabelle 3.3 aussehen. Bei diesem Eintrag werden die optionalen Datenfelder nicht verwendet. Da es sich bei diesem StorageFile um den Typ Stego-Objekt handelt und die optionalen Datenfelder nicht verwendet werden, muss die Datei, in der das Stego-Objekt gespeichert wird, schon existieren und als Cover zur Verfügung stehen.

id	used	filename	maxWriteBytes	overwriteable	type	kryptoId	stegoId
7	false	pic34.jpg	5230	false	2 (stego)	43	21

Tabelle 3.3: Beispiel StorageFile-Datensatz

Die Tabelle 3.4 gibt die zu verwendenden Pfade für die Speicherung des StorageFiles an. In diesem Beispiel sind drei Pfadangaben für das StorageFile mit der ID 7 gespeichert. Somit wird dieses StorageFile nicht nur einmal, sondern dreimal abgelegt. Dadurch ist dieses StorageFile redundant gespeichert. Die drei Pfade sollten nicht auf denselben Datenträger verweisen, da ansonsten beispielsweise bei einem Headcrash des Datenträgers alle drei Dateien auf einmal vernichtet werden.

storageId	directoryId
7	5
7	9
7	17

Tabelle 3.4: Beispiel StorageDirectory-Datensätze

Die Abbildung 3.5 bildet die Tabellen-Einträge als Objekte ab, wie sie eventuell von einem Programm im Arbeitsspeicher abgelegt werden könnten.

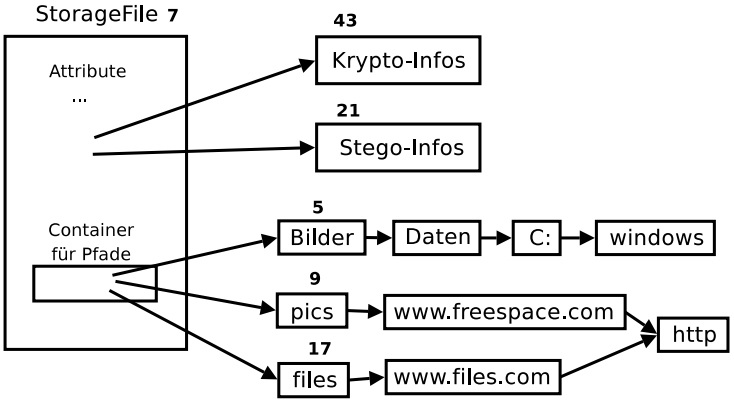

Abbildung 3.5: Objektdarstellung eines StorageFiles

Laut Abbildung 3.5 bilden sich aus den drei Pfadangaben plus dem Dateinamen *pic34.jpg* die vollständigen Dateinamen *C:\Daten\Bilder\pic34.jpg*, *http://www.freespace.com/pics/pic34.jpg* und *http://www.files.com/files/pic34.jpg*. Vor dem Speichern des StorageFiles muss eine der drei Dateien bereits existieren, die als Cover dient. Diese Cover-Datei wird nach dem Auslesen mit dem Stego-Objekt überschrieben und die anderen beiden Dateien werden, falls sie nicht existieren, angelegt und ebenfalls mit dem Dateiinhalt des Stego-Objekts beschrieben.

3.1.3 KryptoOptions und StegoOptions-Tabelle

Die kryptographischen und steganographischen Informationen, die für die StorageFiles benötigt werden, sind in den Tabellen KryptoOptions und StegoOptions gespeichert. Wenn es sich bei allen StorageFiles um den Typ Binär-Datei handelt, dann werden diese beiden Tabellen nicht benötigt. Bei verschlüsselten Dateien wird die Tabelle KryptoOptions benötigt und für Stego-Objekte werden im Regelfall beide Tabellen benötigt, außer die Datenblöcke werden unverschlüsselt eingebettet. Dadurch, dass diese

krypto- und steganographischen Informationen in eigenen Tabellen abgespeichert werden, können mehrere StorageFiles die gleichen Parameter für die Verschlüsselung und für die Einbettung benutzen.

Die KryptoOptions-Tabelle besteht aus folgenden Datenfeldern:

- **id** (Primärschlüssel)
  Die eindeutige Identifikationsnummer repräsentiert die gespeicherten kryptographischen Parameter.
- **algorithm**
  bestimmt, welcher Verschlüsselungsalgorithmus verwendet werden soll.
- **keyLength**
  ist die Länge des Schlüssels. Zum Beispiel könnte die Schlüssellänge 128 Bit betragen.
- **key**
  ist der Schlüssel für die Verschlüsselung.
- **cipherMode**
  bestimmt den zu verwendenden Cipher Mode. (z.B: ECB, CBC, OFB, CTR, ...)

Die StegoOptions-Tabelle beinhaltet folgende Datenfelder:

- **id** (Primärschlüssel)
  Die eindeutige Identifikationsnummer für den steganographsichen Parametersatz.
- **algorithm**
  gibt den Einbettungs-Algorithmus an. Mögliche Werte sind beispielsweise LSB(1), Echo(2), Phase(3) oder Frequency(4). Der Wert 0 steht für *externes Programm*. In diesem Fall müssen die Datenfelder *application* und *appVersion* verwendet werden.
- **stegoKey**
  ist der Stego-Schlüssel für die Einbettung.
- **extOptions**
  dient als Datenfeld für weitere Parameter, die eventuell bei bestimmten Einbettungsverfahren benötigt werden.
- **application** (optional)
  bestimmt den Namen des externen Programms, das für die Einbettung verwendet wird. Zum Beispiel *steghide*, *outguess*, etc.
- **appVersion** (optional)
  gibt die Versionsnummer des externen Programms an.

3.1.4 DistributedFile- und BlockGroup-Tabelle

Die zwei Tabellen DistributedFile und BlockGroup werden zur Verwaltung der eigentlichen Daten des DistributedFiles benötigt. Die Tabelle speichert allgemeine Informationen zum DistributedFile. Diese Tabelle besitzt immer nur einen Datensatz. Die Werte der einzelnen Datenfelder können sich eventuell ändern, jedoch darf diese Tabelle niemals mehr als einen Datensatz besitzen. Die zweite Tabelle BlockGroup gibt darüber Auskunft, welche DatenBlöcke in welchem StorageFile gefunden werden und welche Datenblöcke bei Schreiboperationen durch neue ersetzt wurden.

In der Tabelle DistributedFile stehen zur Speicherung der Eigenschaften folgende Datenfelder zur Verfügung:

- **filename** (optional)
  ist der Name der verteilten Datei (DistributedFile). Dieses Feld ist optional, da das Programm die Datei für weitere Clients sowieso unter einem eigenen Namen zur Verfügung stellen kann.

- **blockSize**
  bestimmt die Größe eines Datenblocks in Byte. Diese Größe ist die kleinste Einheit, in der Datenblöcke verwaltet werden. Jede Blockgruppe ist genau ein Vielfaches dieser Größe. Trotzdem muss die Größe des DistributedFile nicht ein Vielfaches der Blockgröße sein, denn gegebenenfalls wird der letzte Block mit Nullen aufgefüllt. Anhand des Datenfeldes *fileSize* kann genau die Grenze im letzten Datenblock berechnet werden. Der Wert von *blockSize* darf auf keinen Fall verändert werden, denn dieser Wert ist für die Berechnung der jeweiligen Offsets und Längenangaben für alle Lese- und Schreiboperationen, die sich auf die Datenblöcke beziehen, notwendig. *blockSize* kann zum Beispiel den Wert 512 betragen.

- **fileSize**
  speichert die Größe des DistributedFile in Byte. Bei Größenänderungen der verteilten Datei muss diese Größe angepasst werden.

- **readWrite**
  besagt, ob der Inhalt der verteilten Datei (DistributedFile) verändert werden darf. Damit die Größe der Datei verändert werden darf, muss *readWrite* und zusätzlich noch *resizable* oder *onlyBlockResizable* auf TRUE gesetzt werden.

- **resizable**
  Wenn dieses Feld gesetzt ist, darf die Größe der Datei verändert werden. In diesem Fall darf das Feld *onlyBlockResizable* nicht gesetzt werden.

- **onlyBlockResizable**
  Besagt, dass die Datei nur um ein Vielfaches der Blockgröße verändert werden darf. Wenn dieses Feld gesetzt ist muss die Dateigröße (*filesize*) des DistributedFile genau ein Vielfaches der Blockgröße betragen und *resizable* darf nicht gesetzt sein.

- **nextStorageIdForNewRecords**
  Reserviert ein verfügbares StorageFile für die nächsten zu speichernden ASN.1-Strukturen. Der genaue Nutzen bzw. Sinn hinter diesem Datenfeld wird bei den ASN.1-Strukturen genauer erklärt.

Bis auf das Datenfeld *blockSize* dürfen sich alle Werte der Datenfelder während der Benutzung des DistributedFiles verändern.

Die zweite Tabelle BlockGroup speichert die Informationen, in welchen StorageFiles welche Datenblöcke zu finden sind. Pro Datensatz wird eine Blockgruppe des DistributedFiles repräsentiert. Ein Datensatz besteht aus folgenden Datenfeldern:

- **distributedBlockOffset**
  legt den Startpunkt im DistributedFile für die Blockgruppe festgelegt.
- **blockCount**
  bestimmt die Länge der Blockgruppe bzw. die Anzahl der hintereinander folgenden Blöcke.
- **storageId**
  gibt an, auf welchem StorageFile die Blockgruppe gespeichert ist.
- **storageBlockOffset**
  gibt an, wo die Blockgruppe im StorageFile beginnt. Wenn in einem StorageFile mehrere Blockgruppen hintereinander gespeichert werden, dann wird dieses Datenfeld benötigt und würde ab der zweiten Blockgruppe einen anderen Wert als 0 besitzen.

Die Abbildung 3.6 zeigt eine mögliche BlockGroup-Tabelle für ein DistributedFile, das aus insgesamt zehn Datenblöcken besteht. Der Inhalt des DistributedFiles ist laut dieser Abbildung in vier StorageFiles verteilt gespeichert. Das erste StorageFile speichert die Datenblöcke null, eins und zwei. Das zweite StorageFile speichert die Blöcke drei, vier, fünf und sechs. Und das dritte StorageFile speichert die Blöcke sieben, acht und neun. Somit sind in den ersten drei StorageFiles alle zehn Blöcke (von Index null bis neun) des DistributedFiles gespeichert. Wenn es sich bei den StorageFiles um Stego-Objekte handelt, dann ist dieses nicht wiederbeschreibbar und somit darf der Inhalt der einzelnen Datenblöcke nicht einfach überschrieben werden. In diesem Fall wird ein weiteres StorageFile benötigt. Das vierte StorageFile ersetzt die Datenblöcke zwei, drei, fünf und null durch neue Datenblöcke. Dies legen die letzten drei Datensätze in der BlockGroup-Tabelle fest. Wenn ein Datenblock in mehreren unterschiedlichen Versionen in verschiedenen StorageFiles vorliegt, dann bestimmt die Reihenfolge der Datensätze in der BlockGroup-Tabelle die gültige Version. Die letzten eingetragenen Datensätze haben immer Vorgang gegenüber älteren Datensätzen. Somit sind die Datenblöcke zwei, drei, fünf und null aus den StorageFiles eins, zwei und drei nicht mehr gültig und werden durch die Datenblöcke aus dem StorageFile vier ersetzt.

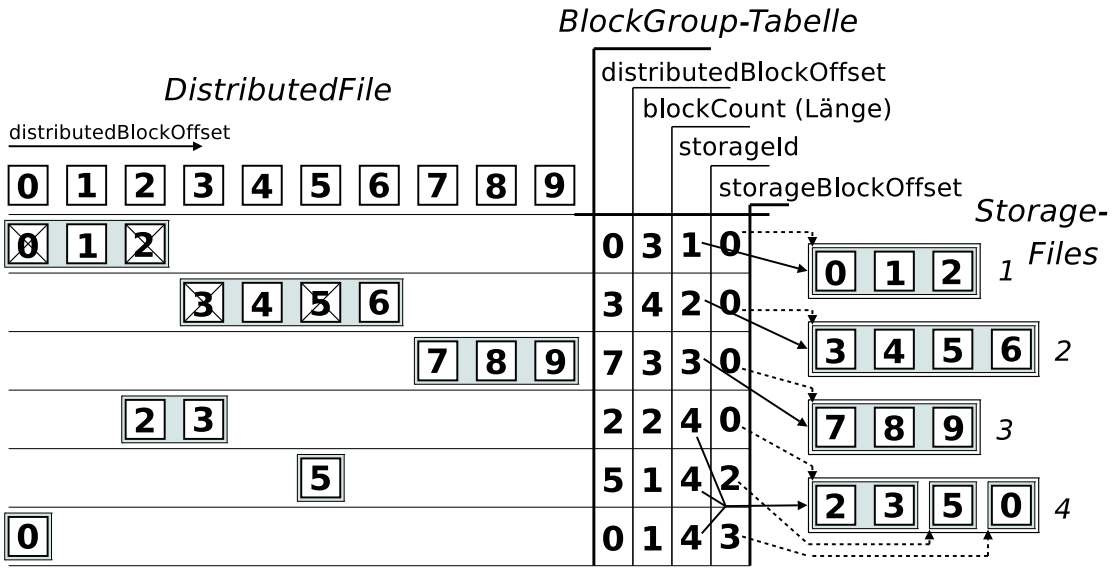

Abbildung 3.6: Beispiel BlockGroup-Tabelle

Dieser Schreibvorgang, der durch das Ersetzen der Datenblöcke ermöglicht wird, hat den Nachteil, dass bei ständigen Schreiboperationen die Anzahl der verwendeten StorageFiles ansteigt und somit der benötigte Speicherplatz ebenfalls ansteigt. Eine Verminderung dieses Problems kann dadurch erzielt werden, indem alle Schreiboperationen des DistributedFiles zuerst gepuffert werden und erst beim Schließen der Datei anschließend gesichert werden. Diese Maßnahme bewirkt, dass bei einem mehrmaligen Überschreiben desselben Blocks zwischen dem Öffnen und Schließen der Datei nicht alle Versionen gespeichert werden, sondern nur die aktuelle letzte Version gespeichert wird. Dieses Problem der steigenden StorageFiles gilt speziell für Stego-Objekte, da diese im Regelfall nicht mehrmals verwendet werden sollten. Bei den anderen beiden StorageFile-Typen (Binärdatei und verschlüsselte Datei) tritt dieses Problem nicht auf, weil die Datenblöcke direkt im jeweiligen StorageFile geändert werden können und dadurch keine weiteren Einträge in der BlockGroup-Tabelle notwendig sind und keine zusätzlichen StorageFiles benötigt werden. Wenn der Speicherbedarf für die StorageFiles zu groß wird und beispielsweise schon mehr als 50 Prozent der Datenblöcke nicht mehr gültig sind, dann könnte zum Beispiel eine Kopie als neues DistributedFile angelegt werden und somit das alte DistributedFile mit all seinen StorageFiles gelöscht werden.

3.2 ASN.1-Stukturen

Alle Verwaltungsinformationen für das Auslesen und Bearbeiten des DistributedFiles werden ebenfalls wie die Datenblöcke in StorageFiles gespeichert. Hierfür wird die Abstract Syntax Notation One (ASN.1) als Beschreibungssprache zur Definition der notwendigen Datenstrukturen verwendet. Zur Codierung der ASN.1-Stukturen, die für die Speicherung in den StorageFiles erforderlich ist, werden die Basic Encoding Rules (BER) angewendet. Die Basic Encoding Rules benutzen die TLV-Form, um die ASN.1

Strukturen zu codieren. Im Regelfall wird BER zur Übertragung der Datenstrukturen in Protokollen verwendet, jedoch bietet sie die selben Vorteile auch bei der Speicherung der ASN.1 Strukturen in Dateien. Die Vorteile von BER und der TLV-Form sind:

- Parameter können optional erklärt werden
- Parameter können unterschiedliche Längen besitzen
- Spätere Versionen des Protokolls beziehungsweise der einzelnen ASN.1 Strukturen können leicht abwärtskompatibel gehalten werden

Die Flexibilität von BER ist der größte Vorteil für die Speicherung. Sie ermöglicht ein einfaches Speichern und Auslesen der notwendigen Informationen. Des Weiteren existieren bereits fertige ASN.1 Compiler für diverse Programmiersprachen, die die Implementierung der Basic Encoding Rules für die selbst definierten ASN.1 Strukturen übernehmen. So existieren beispielsweise die freien ASN.1 Compiler *asn1c* und *snacc* für die Programmiersprache C.

Die aktuelle Version von ASN.1 ist in der Serie X.680[1] standardisiert und die Basic Encoding Rules (BER) sind im Standard X.690[2] festgelegt. Zum besseren Verständnis des ASN.1 Standards existieren die zwei frei verfügbaren Bücher *ASN.1 Communication between Heterogeneous Systems*[3] von Olivier Dubuisson und *ASN.1 Complete*[4] von John Larmouth. Die spezifischen Strukturen, die für dieses Konzept benötigt werden, sind ASN.1 konform definiert. (vgl. Dubuisson, 2000) (vgl. Larmouth, 1999)

Zum Speichern der notwendigen Informationen sind folgende ASN.1-Strukturen definiert:

- DistributedFile
- FileSize
- FileChangeable
- Directory
- StorageFile
- Cover
- HashData

[1] *Standard-Serie X.680:*
ITU-T Rec. X.680 | ISO/IEC 8824-1 - http://www.itu.int/rec/T-REC-X.680/en
ITU-T Rec. X.681 | ISO/IEC 8824-2 - http://www.itu.int/rec/T-REC-X.681/en
ITU-T Rec. X.682 | ISO/IEC 8824-3 - http://www.itu.int/rec/T-REC-X.682/en
ITU-T Rec. X.683 | ISO/IEC 8824-4 - http://www.itu.int/rec/T-REC-X.683/en
[2] *ITU-T Rec. X.690 | ISO/IEC 8825-1* - http://www.itu.int/rec/T-REC-X.690/en
[3] *ASN.1 Communication between Heterogeneous Systems* - http://www.oss.com/asn1/dubuisson.html
[4] *ASN.1 Complete* - http://www.oss.com/asn1/larmouth.html

- StoredBlocks
- BlockGroup
- NextEntries
- StegoOptions
- StegoApplication
- KryptoOptions

Jede gespeicherte Struktur-Instanz kann in den meisten Fällen eins zu eins als Datensatz für die jeweilige Tabelle gespeichert werden. Benutzt eine Struktur wiederholt dieselbe Identifikationsnummer, so ist der entsprechende Datensatz in der jeweiligen Tabelle zu ersetzen. In einem StorageFile können mehrere beliebige Struktur-

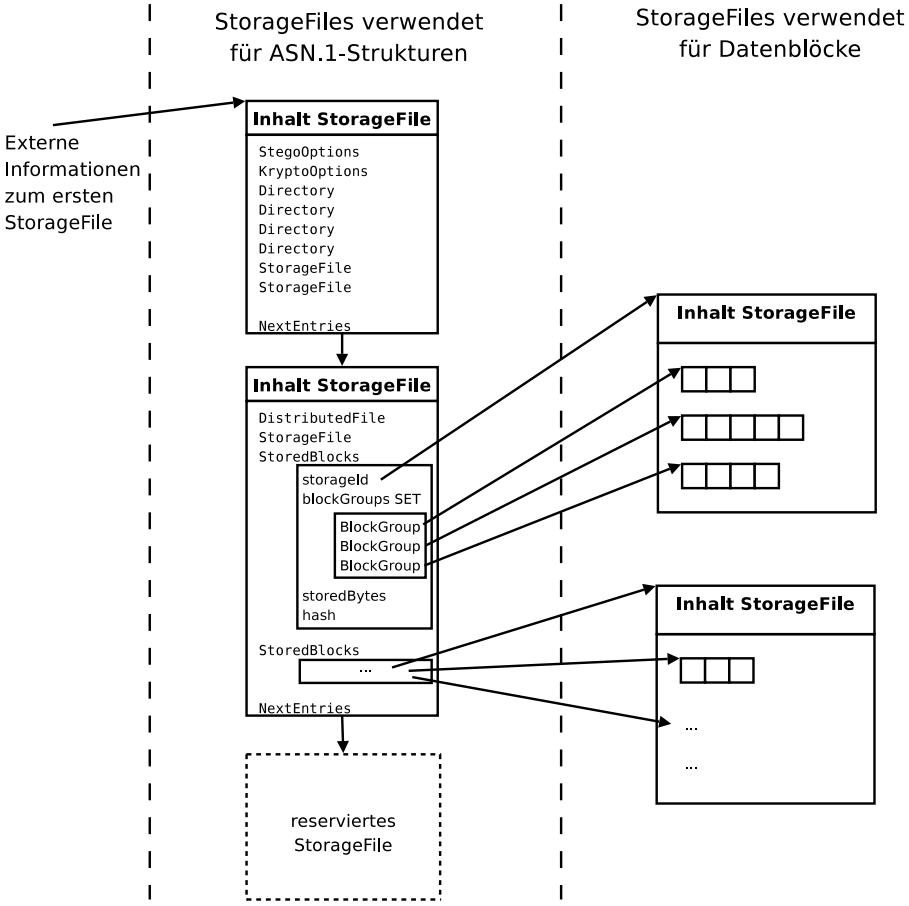

Abbildung 3.7: Gespeicherte Struktur-Instanzen und Datenblöcke

Instanzen gespeichert werden. Abschließend wird immer eine Instanz der Struktur NextEntries gespeichert. Die Struktur NextEntries besitzt ein Attribut *storageId*, welches das nächste StorageFile für weitere gespeicherte ASN.1-Strukturen festlegt. Dieses nächste StorageFile muss noch nicht existieren. Falls dieses StorageFile noch nicht

existiert bzw. verwendet worden ist, dann ist die Liste der gespeicherten ASN.1-Strukturen zu Ende. Obwohl das jeweilige StorageFile von NextEntries eventuell noch nicht existiert, muss es bei NextEntries schon festgelegt und damit für weitere Einträge reserviert werden. Der Grund dafür liegt darin, dass der gespeicherte Inhalt des StorageFiles (z.B: eines Stego-Objekts) je nach Typ nicht mehr geändert werden kann. In der Tabelle DistributedFile wird im Datenfeld *nextStorageIdForNewRecords* die letzte NextEntries-Referenz gespeichert. Die Abbildung 3.7 zeigt die gespeicherten Struktur-Instanzen und die jeweils zuletzt angeführte Struktur NextEntries, die auf das nächste StorageFile referenziert. Zum Schluss verweist NextEntries auf ein StorageFile, das noch nicht verwendet worden ist und somit ist die Liste der ASN.1-Strukturen hier beendet. Das nicht verwendete StorageFile, auf welches die letzte NextEntries-Instanz verweist, ist somit automatisch für weitere ASN.1-Strukturen reserviert und darf daher nicht zur Speicherung von Datenblöcken verwendet werden.

Abgesehen von der Struktur-Instanz NextEntries dürfen keine anderen Struktur-Instanzen auf Einträge verweisen, die noch nicht gespeichert sind. Daher müssen die Struktur-Instanzen in der richtigen Reihenfolge gespeichert werden. Damit beispielsweise eine Struktur StorageFile gespeichert werden kann, müssen die notwendigen Einträge für Verzeichnisse, kryptographische und steganographische Parameter, etc. zuvor als Struktur-Instanzen gespeichert werden. Die Abbildung 3.7 zeigt eine mögliche Reihenfolge der einzelnen Strukturen.

3.2.1 DistributedFile Struktur

Die Struktur DistributedFile beinhaltet allgemeine Informationen zur verteilt gespeicherten Datei. Zusätzlich verwendet die Struktur zwei weitere Strukturen bzw. Sequenzen. Durch die separaten Strukturen *FileSize* und *FileChangeable* können die Eigenschaften *size*, *readWrite*, *resizable* und *onlyBlockResizable* jederzeit durch das Speichern neuer Instanzen dieser Strukturen geändert werden.

```
DistributedFile ::= [PRIVATE 1] SEQUENCE
{
    filename              UTF8String OPTIONAL,
    blockSize             INTEGER,
    size                  FileSize,
    changeable            FileChangeable
}

FileSize ::= [PRIVATE 2] SEQUENCE
{
    size                  INTEGER
}

FileChangeable ::= [PRIVATE 3] SEQUENCE
{
    readWrite             BOOLEAN,
    resizable             BOOLEAN,
    onlyBlockResizable    BOOLEAN
}
```

Listing 3.1: DistributedFile Struktur

3.2.2 Directory Struktur

Die Directory Struktur ist das Äquivalent zur Directory Tabelle.

```
Directory ::= [PRIVATE 4] SEQUENCE
{
    id          INTEGER,
    rootDirId   INTEGER,
    name        UTF8String
}
```

Listing 3.2: Directory Struktur

3.2.3 StegoOptions Struktur

Diese Struktur beinhaltet alle notwendigen Informationen für eine steganographische Einbettung. Mithilfe des Attributs *stegoId* kann die Struktur StorageFile auf die gewünschten steganographischen Parameter verweisen. Wenn der Inhalt vor der Einbettung noch verschlüsselt werden soll, muss zusätzlich die Struktur KryptoOptions verwendet werden, auf die ebenfalls von der StorageFile-Struktur referenziert wird. Falls für die Einbettung ein externes Programm wie beispielsweise Steghide verwendet werden soll, dann kann das Attribut algorithm auf 0 gesetzt werden und mit der optionalen Struktur StegoApplication das gewünschte Programm angegeben werden.

```
StegoOptions ::= [PRIVATE 11] SEQUENCE
{
    algorithm     ENUMERATED
    {
        application (0),
        lsb         (1),
        echo        (2),
        phase       (3),
        frequency   (4)
    },
    stegoKey      OCTET STRING,
    extOptions    UTF8String OPTIONAL,
    stegoApp      StegoApplication OPTIONAL
}

StegoApplication ::= [PRIVATE 12] SEQUENCE
{
    name          UTF8String,
    version       UTF8String
}
```

Listing 3.3: StegoOptions Struktur

3.2.4 KryptoOptions Struktur

KryptoOptions stellt alle notwendigen Attribute für die kryptographischen Informationen zur Verfügung. Die jeweiligen Werte können eins zu eins in die gleichnamige Tabelle übernommen werden.

```
KryptoOptions ::= [PRIVATE 13] SEQUENCE
{
    algorithm    ENUMERATED
    {
        rsa          (1),
        elGamalDL    (2),
        elGamalEC    (3),
        aes          (4)
    },
    keyLength    INTEGER OPTIONAL,
    key          OCTET STRING OPTIONAL,
    cipherMode   ENUMERATED
    {
        ecb          (1),
        cbc          (2),
        ofb          (3),
        ctr          (4)
        -- koennen noch weitere folgen --
    }
}
```

Listing 3.4: KryptoOptions Struktur

3.2.5 HashData Struktur

HashData besteht aus zwei Attributen, dies sind *algorithm* und *hashValue*. Das erste bestimmt den zu verwendenden Hash-Algorithmus und das zweite speichert den Hashwert. Die Struktur HashData wird von den Strukturen StorageFile und StoredBlocks verwendet.

```
HashData ::= [PRIVATE 7] SEQUENCE
{
    algorithm       INTEGER {
        md5           (1),
        sha1          (2),
        sha256        (3),
        sha512        (4)
        -- evt. noch weitere --
    },
    hashValue       OCTET STRING
}
```

Listing 3.5: BlockGroup Struktur

3.2.6 StorageFile Struktur

Die Attribute der Sturktur StorageFile entsprechen in den meisten Fällen den Datenfeldern aus der Tabelle StorageFile. Durch das Attribut *redundantDirId* können noch weitere Speicherpfade (abgesehen vom Pfad durch *directoryId*) für eine redundante Speicherung des StorageFiles angegeben werden. Wenn das StorageFile vom Typ *stego* ist und der vollständige Dateiname durch *filename* plus *directoryId* vor der Speicherung nicht automatisch dem Cover entspricht, kann durch das optionale Attribut *stegoCover* eine alternative Cover-Datei angegeben werden. Mit dem Attribut *hash* kann die Integrität des StorageFiles vor dessen Benutzung festgehalten werden. Nach

der Speicherung wird der entsprechende Hashwert in der Struktur StoredBlocks gespeichert.

```
StorageFile ::= [PRIVATE 5] SEQUENCE
{
    id              INTEGER,
    filename        UTF8String,
    directoryId     INTEGER,
    redundantDirId  SET OF INTEGER OPTIONAL,
    maxBytes        INTEGER,
    overwritable    BOOLEAN,
    compression     INTEGER {
        none          (0),
        krypto        (1),
        stego         (2)
        -- noch weitere --
    },
    type            ENUMERATED {
        binary        (0),
        krypto        (1),
        stego         (2)
    },
    stegoId         INTEGER,
    kryptoId        INTEGER,
    hash            HashData OPTIONAL,
    stegoCover      Cover OPTIONAL
}

Cover ::= [PRIVATE 6] SEQUENCE
{
    filename                    UTF8String OPTIONAL,
    directoryId                 INTEGER OPTIONAL,
    overwriteWithStegoObject    BOOLEAN,
    deleteCover                 BOOLEAN
}
```

Listing 3.6: Storage Struktur

3.2.7 StoredBlocks Struktur

Die Struktur StoredBlocks beinhaltet alle Informationen über die gespeicherten Blöcke bzw. Blockgruppen in einem StorageFile. Das Attribut *storageId* bestimmt das StorageFile, auf welches sich die Informationen beziehen. Das Attribut *blockGroups* ist vom Typ SET und beinhaltet mehrere Instanzen der Struktur *BlockGroup*. Falls nötig, kann mit dem Attribut *storedBytes* die Anzahl der gespeicherten (bzw. eingebetteten) Bytes im StorageFile angegeben werden. Der Wert von *storedBytes* muss nicht gleich der Anzahl der gespeicherten Blöcke mal der Blöckgröße sein. Dies ist der Fall, wenn die Daten vor dem Speichern komprimiert werden. Mit dem Attribut *hash* kann zusätzlich ein Hashwert vom referenzierten StorageFile gespeichert werden. Hiermit kann die Integrität des StorageFiles und somit der gesicherten Datenblöcke gewährleistet werden.

```
StoredBlocks ::= [PRIVATE 8] SEQUENCE
{
    storageId       INTEGER,
    blockGroups     SET OF BlockGroup,
    storedBytes     INTEGER OPTIONAL,
    hash            HashData OPTIONAL
}
```

```
BlockGroup ::= [PRIVATE 9] SEQUENCE
{
    distributedBlockOffset    INTEGER,
    blockCount                INTEGER
}
```

Listing 3.7: StoredBlocks Struktur

Die Struktur BlockGroup speichert die Informationen, welche Blöcke im StorageFile welchen Blöcken im DistributedFile entsprechen. Die Abbildung 3.8 zeigt eine mögliche StoredBlocks-Instanz mit ihren Werten. Die Instanz besitzt im Set blockGroups mehrere Instanzen der Struktur BlockGroup. Diese BlockGroup-Instanzen beschreiben das abgebildete StorageFile mit der ID 4. Des Weiteren können durch die StoredBlocks-Instanz die notwendigen Einträge für die BlockGroup-Tabelle berechnet werden.

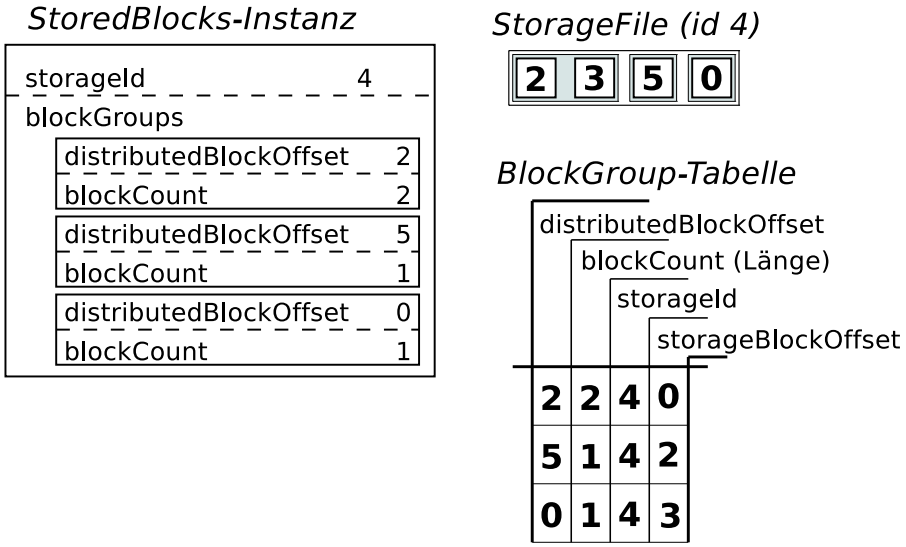

Abbildung 3.8: Beispiel StoredBlocks Struktur

3.2.8 NextEntries Struktur

Diese Struktur besitzt das Attribut *storageId*, welches die Identifikationsnummer des nächsten StorageFile speichert. Dieses referenzierte StorageFile beinhaltet die nächsten gespeicherten Instanzen der ASN.1-Strukturen. Wenn das angegebene StorageFile noch nicht verwendet wird, dann ist die Liste der gespeicherten Struktur-Instanzen abgeschlossen und das angegebene StorageFile wird für weitere Strukturen reserviert.

```
NextEntries ::= [PRIVATE 10] SEQUENCE
{
    storageId    INTEGER
}
```

Listing 3.8: NextEntries Struktur

3.3 Möglichkeiten durch dieses Konzept

Durch dieses Konzept ergeben sich einige Möglichkeiten, die generell in zwei Kategorien unterteilt werden können.

Verteilte Speicherung einer Datei ...

- lokal am Rechner
- auf extern verfügbaren Webspace-Accounts

3.3.1 Versteckte Speicherung mit Hilfe der Steganography

Bei der verteilten Speicherung einer Datei lokal am Rechner ist im Regelfall die verteilte Speicherung nur ein Mittel zum Zweck, da in den meisten Fällen ein Cover zur Speicherung einer größeren zu versteckenden Datei zuwenig Einbettungskapazität bietet. Lokal am Rechner macht nur der StorageFile-Typ Stego-Objekt Sinn. Denn wenn eine Datei kryptographisch oder unverschlüsselt gespeichert werden soll, verursacht die Aufteilung in einzelne Blöcke keinen zusätzlichen Nutzen. Somit bietet sich dieses Konzept lokal am Rechner für das Verstecken von Dateien mit Hilfe der Steganographie an. Diese Möglichkeit erfüllt somit genau die Grundbedingung dieser Arbeit. Doch dieses Konzept bietet noch einen weiteren Nutzen.

3.3.2 Externe Speicherung auf Webaccounts als Backuplösung

Die verteilte Speicherung einer Datei auf extern verfügbaren Webspace-Accounts kann als ”einfache Backuplösung für Jedermann” verwendet werden. Beispielsweise können lokale Partitionen von einem Rechner auf verschiedenen Webaccounts redundant gesichert werden. Hier macht die Unterstützung des StorageFile-Typs Kryptographie durchaus Sinn. Denn wenn eine Datei oder ein ganzes Partitions-Image kryptographisch in einer Datei beispielsweise auf einem FTP-Server gespeichert werden soll, dann muss jedes Mal die ganze Datei heruntergeladen werden, um ein einzelnes Byte ändern zu können. Durch die verteilte Speicherung muss nur jeweils das verschlüsselte StorageFile mit dem jeweiligen Datenblock heruntergeladen, geändert und anschließend wieder hochgeladen werden. Falls der Anbieter des Webspaces keine Speicherung von verschlüsselten Dateien erlaubt, kann immer noch auf den StorageFile-Typ Steganographie gewechselt werden. Jedoch wird natürlich in diesem Fall ein Vielfaches der Kapazität der jeweiligen zu speichernden Datei benötigt.

3.3.3 Externe Speicherung auf Webaccounts zur versteckten Übertragung

Dadurch, dass die Stego-Objekte auch auf Webaccounts abgelegt werden können, kann ein User die geheime Datei auf einem oder mehreren Webaccounts versteckt speichern.

Ein anderer User kann mithilfe der notwendigen Informationen diese versteckte Datei wieder finden und extrahieren.

3.4 Sicherheit dieses Konzepts

Die Sicherheit dieses Konzepts ist hauptsächlich von zwei Faktoren abhängig:

- Es fordert die sichere Aufbewahrung des ersten Schlüssels plus Zusatzinformationen für das Lesen des ersten StorageFiles.
- Je nach StorageFile-Typ, Kryptographie oder Steganographie, müssen entsprechend sichere Verschlüsselungs- bzw. Einbettungsverfahren verwendet werden.

Wenn die Informationen des ersten Schlüssels plus die zusätzlichen Informationen wie Dateiname, Pfad, verwendeter Algorithmus, etc. in falsche Hände geraten, kann diese Person die geschützte oder versteckte Datei vollständig extrahieren und lesen. Alle notwendigen Informationen für das Lesen der weiteren StorageFiles liefern die schon zuvor extrahierten StorageFiles.

Bei den Verschlüsselungs- und Einbettungsverfahren sollten immer bewährte sichere und öffentlich bekannte Algorithmen, deren Sicherheit nur vom Schlüssel abhängig sind, verwendet werden. Speziell für die StorageFiles, in denen die ASN.1-Strukturen gespeichert werden, müssen sichere Algorithmen benutzt werden. Falls die StorageFiles mit den Datenblöcken einen unsicheren Algorithmus verwenden und dadurch der Inhalt einzelner StorageFiles extrahiert werden kann, kann deshalb nicht gleich die ganze Datei extrahiert werden. Denn durch das Knacken eines StorageFiles, das Datenblöcke beinhaltet, werden keine weiteren Informationen für das nächste StorageFile gewonnen. Es können nur die StorageFiles mit demselben Schlüssel extrahiert werden. Jedoch kann jedes StorageFile seine eigenen Parameter für die Verschlüsselung und Einbettung verwenden. Prinzipiell soll trotz dieses Umstandes eine sichere Verschlüsselung und Einbettung für die StorageFiles, die die Datenblöcke des DistributedFiles beinhalten, gewählt werden.

Die sichere Aufbewahrung des ersten Schlüssels und der Zusatzinformationen für das erste StorageFile werden im übernächsten Kapitel behandelt.

Kapitel 4

Distributed File Storage (DFS)

In diesem Kapitel wird der grundsätzliche Aufbau einer möglichen Realisierung der verteilten Speicherung, wie sie in Kapitel 3 dargestellt ist, beschrieben. Das Programm für die Verwaltung der verteilten Dateien wird im weiteren Verlauf *Distributed File Storage* genannt.

Distributed File Storage verwaltet virtuelle Dateien, die von einer Client-Software mit Hilfe des File Operation Protokolls (FOP) geöffnet, gelesen, beschrieben und geschlossen werden können. Die virtuellen Dateien können in mehrere Blöcke unterteilt und auf unterschiedlichste Weise gesichert am DFS vorliegen. DFS verwendet das Konzept zur verteilten Speicherung aus dem Kapitel 3. DFS kann intern in mehrere Komponenten unterteilt werden, die jeweils eine andere Aufgabe erfüllen und aufeinander aufbauen.

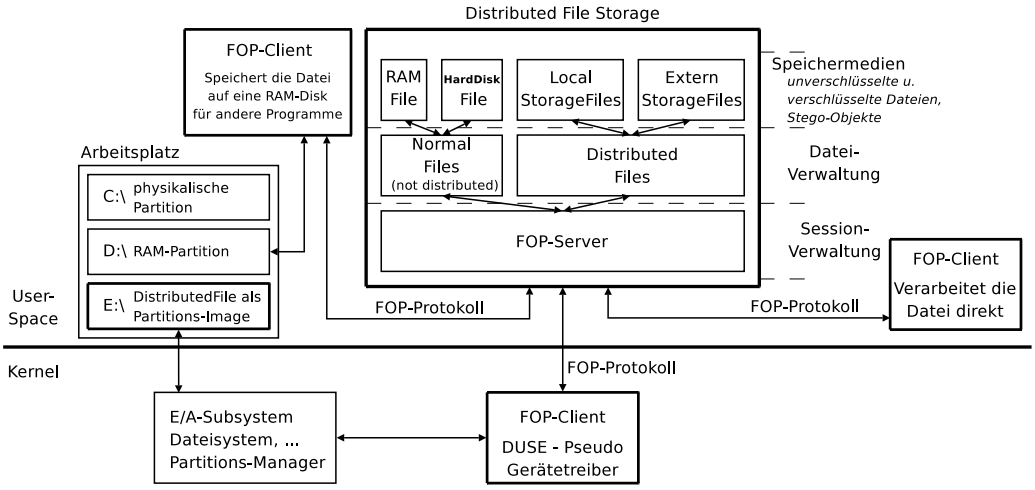

Abbildung 4.1: Überblick: Distributed File Storage

Die Abbildung 4.1 zeigt die einzelnen Komponenten aus denen sich das DFS zusam-

mensetzt. Die unterste Schicht bildet der FOP-Server. Er nimmt die Anfragen des FOP-Clients zum Öffnen, Lesen, Schreiben und Schließen einer virtuellen Datei entgegen. Der Client verwendet für die Kommunikation mit dem Server das File Operation Protokoll (FOP). Wenn die zu öffnende Datei nur lesbar ist, kann sie von mehreren Clients gleichzeitig geöffnet werden. Der rechts abgebildete FOP-Client verarbeitet direkt die Daten der geöffneten Datei. Jedoch ist es auch möglich, dass ein Client die geöffnete Datei des DFS auf eine RAM-Disk kopiert und somit anderen Programmen, die das FOP-Protokoll nicht beherrschen, zur Verfügung stellt. Diesen Fall stellt der links abgebildete Client dar. Die Änderungen der Ram-Disk-Datei können anschließend wieder vom Client zum Server übermittelt werden. Der Pseudo-Gerätetreiber DUSE ermöglicht das Einbinden der virtuellen Datei als Dateisystem-Image ins Betriebssystem. Natürlich muss in der Datei auch wirklich ein Image von einem Dateisystem (z.B: NTFS) gespeichert sein, ansonsten fordert Windows den User auf, die Partition zu formatieren. Der Gerätetreiber selbst kommuniziert ebenfalls mit dem Server mithilfe des FOP-Protokolls. Der Treiber DUSE ermöglicht somit, dass ein ganzes Dateisystem beispielsweise steganographisch versteckt oder extern auf einem Webaccount gespeichert wird.

Im Zusammenhang mit dieser Arbeit wurden folgende Punkte bereits getestet und implementiert:

- FOP-Server
- Anbieten von einfachen RAM-Files für die FOP-Clients
- FOP-Client im User-Mode
- Pseudo Gerätetreiber DUSE zum Einbinden eines gespeicherten Dateisystem-Images

Diese implementierten Komponenten bieten die Grundvoraussetzung für die Implementierung des Konzepts der verteilten Speicherung, das im Kapitel 3 abgehandelt wurde. Diese Arbeit beinhaltet jedoch nicht die Implementierung der verteilten Speicherung, da dies den Zeitrahmen für diese Arbeit überschritten hätte.

4.1 Verwaltung und Konfiguration eines DistributedFiles

Wenn ein Client-Programm ein DistributedFile öffnen soll, dann erfolgt die Eingabe des ersten Schlüssels, des Pfades vom ersten StorageFile und alle weiteren Parameter, die zum Extrahieren des DistributedFiles notwendig sind, ausschließlich am Server. Alle Informationen, die zum Öffnen und Lesen des DistributedFiles notwendig sind, müssen dem Server bekannt sein, damit er das DistributedFile einem Client anbieten kann. Die Konfiguration von neuen StorageFiles oder sonstige Konfigurationsänderungen am DistributedFile erfolgen ausschließlich am Server. Nach außen hin zum Client-Programm kann ein DistributedFile beispielsweise nicht von einem RAM-File unterschieden werden. Für das Client-Programm handelt es sich immer nur um eine

zur Verfügung stehende Datei, die vom Server angeboten wird und mittels des FOP-Protokolls geöffnet und benutzt werden kann.

4.2 DUSE

DUSE steht für Device in Userspace und soll den eigentlichen Zugriff auf das Gerät vom Kernel in den Userspace verlagern. Der Name DUSE ist an das Kernelmodul FUSE angelehnt, das eine ähnliche Funktion für Dateisysteme zur Verfügung stellt. DUSE ist speziell für virtuelle Geräte konzipiert, die durch eine Datei oder Netzwerkverbindung realisiert werden, da in diesen Fällen der Zugriff auf das Gerät vom Userspace aus erfolgen kann. Die Implementierung von DUSE ist als Gerätetreiber realisiert. Damit DUSE verwendet werden kann, muss der Treiber duse.sys in den Kernel geladen werden. Der Quellcode, Treiber und die genaue Beschreibung zur Installation finden sich auf der CD, die der Arbeit beigelegt ist.

4.3 File Operation Protocol (FOP)

Um die Interoperabilität zwischen dem FOP-Client und -Server gewährleisten zu können, wird im folgenden Abschnitt das File Operation Protocol spezifiziert.

Das File Operation Protocol dient zum Öffnen, Schließen, Lesen, Schreiben, Verkleinern bzw. Vergrößern von Dateien. Das FOP Protokoll wurde in Zusammenhang mit dieser Arbeit entwickelt um den Zugriff auf die Dateien (Client-Programm) und das Speichern der Dateien (z.B: steganographisch in Bilddateien, Server-Programm) unabhängig von einander implementieren und behandeln zu können. Zum Beispiel kann es sich beim Client Programm um einen virtuellen Festplattentreiber handeln, der im Kernel-Mode läuft und über das FOP Protokoll mit dem Server kommuniziert, der hingegen im User-Mode läuft. Ein weiterer Vorteil besteht darin, dass das bei Anpassungen des Client-Programms, der Server nicht geändert werden muss.

Damit in der geöffneten Datei Operationen durchgeführt werden können, muss ein Client-Programm anhand einer Anfrage die gewünschte Operation an den Server senden. Der Server antwortet auf die Anfrage und gibt Auskunft darüber, ob die Operation erfolgreich durchgeführt werden konnte. Damit ein Client-Programm das FOP Protokoll sinnvoll verwenden kann, muss ein bidirektionaler Datenstrom zwischen Client und Server möglich sein. Bei der bidirektionalen Datenverbindung kann es sich zum Beispiel um eine TCP/IP Verbindung handeln. Jedoch auch andere Verbindungen außer Netzwerkverbindungen sind möglich. Beispielsweise kann das FOP Protokoll auch bei einer bidirektionalen Datenverbindung zwischen Client- und Server-Prozess, die mit Hilfe von zwei Named Pipes realisiert ist, verwendet werden.

4.3.1 FOP Sessions

Das FOP Protokoll arbeitet mit Sessions. Jeder Client kann pro Datenverbindung bis zu 256 Sessions (inkl. Session 0) aufbauen. Für jede geöffnete Datei erhält der Client eine neue Session Identifikationsnummer, die einen Wert zwischen 1 und 255 besitzt. Muss ein Client mehr als 255 Dateien gleichzeitig offen halten, so muss er eine zweite Datenverbindung zum Server aufbauen. Die Session mit der Identifikationsnummer 0 existiert, sobald eine Datenverbindung aufgebaut wird. Die Session 0 muss über die ganze Verbindungsdauer bestehen bleiben. Die Session 0 dient für spezielle Aufgaben, wie zum Beispiel zum Öffnen einer Datei oder zum Trennen der Datenverbindung. Das Trennen der Datenverbindung wird durch das Schließen der Session 0 ausgelöst. Beim Schließen der Datenverbindung werden zwangsweise alle anderen offenen Sessions von 1 bis 255 geschlossen. Es wird jedoch empfohlen, dass alle Sessions extra vor der Session 0 geschlossen werden. Das FOP Protokoll unterstützt zwei Arten von Zugriffen. Den block-basierten und byte-basierten Zugriff. Je nach Zugriffsart muss der Datei-Offset und die Länge der zu lesenden bzw. schreibenden Daten in Bytes oder in Blöcken angegeben werden. Des Weiteren kann der Offset und die Länge in jeweils vier Byte großen Datenfeldern oder in acht Byte großen Datenfeldern gesendet werden. Durch die acht Byte großen Datenfeldern ist auch ein bytebasierter Zugriff auf Dateien mit einer Größe über 4 GB möglich.

Eine Datenverbindung zwischen Client und Server arbeitet in zwei Phasen. In Phase 1 wird die Version des FOP Protokoll ausgehandelt. Diese Phase 1 ist derzeit noch sehr primitiv realisiert. Jedoch kann sie jederzeit durch eine bessere Version, die nicht nur die Version des FOP Protokolls aushandelt, sondern eventuell noch Verschlüsselung, Datenkomprimierung etc festlegt, ersetzt werden. In der Phase 2 der Datenverbindung wird das eigentliche FOP Protokoll in der verhandelten Version verwendet. Befindet sich einmal die Datenverbindung in Phase 2 und verwendet das FOP Protokoll, dann kann nicht mehr in die Phase 1 gewechselt werden. Die Phase 2 wird durch die Trennung der Verbindung geschlossen.

In diesem Abschnitt folgt nun die genaue Erklärung zum FOP Protokoll Version 1.

4.3.2 Aushandlung der FOP Version (Übergang Phase 1 zu 2)

Die Aushandlung und Festlegung der FOP Version beginnt nach dem Verbindungsaufbau. Der Client wartet auf das erste gesendete Byte vom Server. Der Server sendet die gewünschte FOP Version. Der Client kann die Version mit einem Byte langen Acknowledgement Code (Ack Code) annehmen oder ablehnen. Zum Akzeptieren der FOP Version sendet der Client ein ACK (Wert 0, positive Bestätigung) an den Server. Zum Ablehnen der FOP Version muss der Client ein NAK (Wert 1, negative Bestätigung) an den Server senden. Ein Akzeptieren der Version vom Client hat zur Auswirkung, dass sich ab diesem Zeitpunkt der Client und Server in Phase 2 befinden und nur noch das FOP Protokoll verwenden dürfen. Beim Ablehnen der vorgeschlagenen Version kann der Server eine neue FOP Version, die sich von der alten unterscheiden muss,

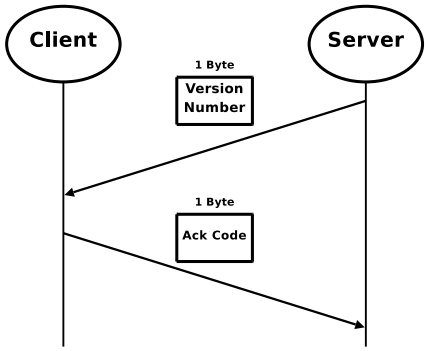

Abbildung 4.2: FOP Versionsaushandlung

vorschlagen oder die Verbindung zum Client trennen. Die Abbildung 4.2 zeigt den zeitlichen Ablauf der FOP Versionsaushandlung.

4.3.3 Open Request (Anfrage vom Client zum Server)

Zum Öffnen einer Datei muss der Client einen Open Request an den Server senden. Der Server antwortet auf die Anfrage mit einem Open Response. Der genaue Aufbau des Open Requests kann der Abbildung 4.3 entnommen werden.

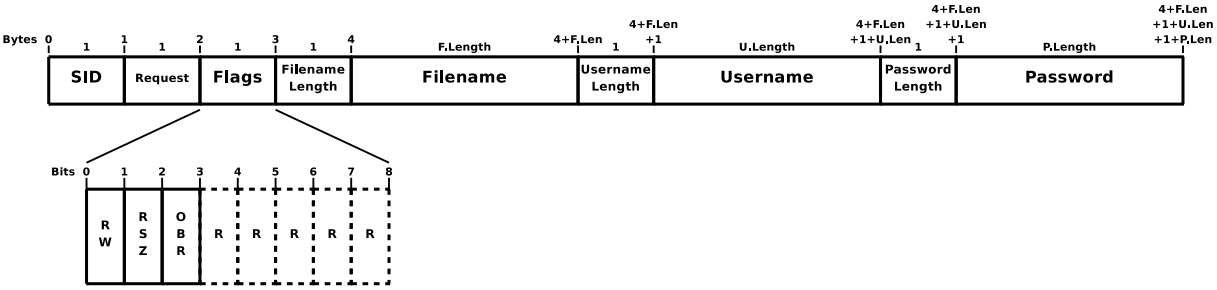

Abbildung 4.3: Anfrage zum Öffnen einer Datei

- **SID** Session Identification
  Beim Open Request muss immer die SID 0 verwendet werden.
- **Request**
  Das Feld Request muss auf REQUEST_OPEN_FILE (Wert 1, 0x01) gesetzt werden.
- **Flags**
  Die Flags können dazu verwendet werden, um die Datei mit bestimmten Eigenschaften zu öffnen. Kann der Server diesen Anforderungen nicht gerecht werden, weil beispielsweise die Größe der Datei nicht geändert werden kann,

jedoch das RSZ Flag gesetzt ist, so schlägt das Öffnen der Datei fehl. Als Kontrolle, dass die Flags vom Server auch wirklich umgesetzt wurden, können die Flags der Server Antwort kontrolliert werden.

  - **RW** Read Write
    öffnet die Datei zum Lesen und Schreiben. Ist die Datei am Server nur lesbar und das RW Flag ist gesetzt, so schlägt das Öffnen der Datei fehl. Ist das RW Flag **nicht** gesetzt, dann werden die Flags RSZ und OBR ignoriert.
  - **RSZ** Resizable
    Die Größe der Datei kann byteweise verändert werden. Das beinhaltet sowohl das Vergrößern als auch das Verkleinern der Datei. Die Größenänderung kann eine beliebige Anzahl von Bytes betragen und ist unabhängig von der Blockgröße der Datei. Ist das RSZ Flag gesetzt, dann wird das Flag OBR ignoriert.
  - **OBR** Only Blocksize Resizable
    Die Datei kann nur um die Blockgröße der Datei vergrößert bzw. verkleinert werden. Dieses Flag wird nur berücksichtigt, wenn das RW Flag gesetzt und das RSZ Flag nicht gesetzt ist.

- **Filename Length**
  beinhaltet die Anzahl der Bytes des UTF-8 kodierten Dateinamens. Die Anzahl der Bytes muss nicht gleich der Anzahl der Zeichen entsprechen. Nur im Spezialfall, wenn der UTF-8 String nur ASCII-Zeichen beinhaltet, würde die Anzahl der Bytes gleich der Anzahl der Zeichen entsprechen.

- **Filename**
  ist der Dateiname als UTF-8 kodierte Zeichenkette. Das Ende der Zeichenkette kann nur anhand der Filename Length bestimmt werden. Die Zeichenkette beinhaltet keinen Null-Terminator, welcher das Ende kennzeichnen würde.

- **Username Length**
  enthält die Anzahl der Bytes des UTF-8 kodierten Usernamens.

- **Username**
  ist ein UTF-8 kodierter Username.

- **Password Length**
  inkludiert die Anzahl der Bytes des UTF-8 kodierten Passwortes.

- **Password**
  ist ein UTF-8 kodiertes Passwort.

4.3.4 Open Response (Antwort vom Server zum Client)

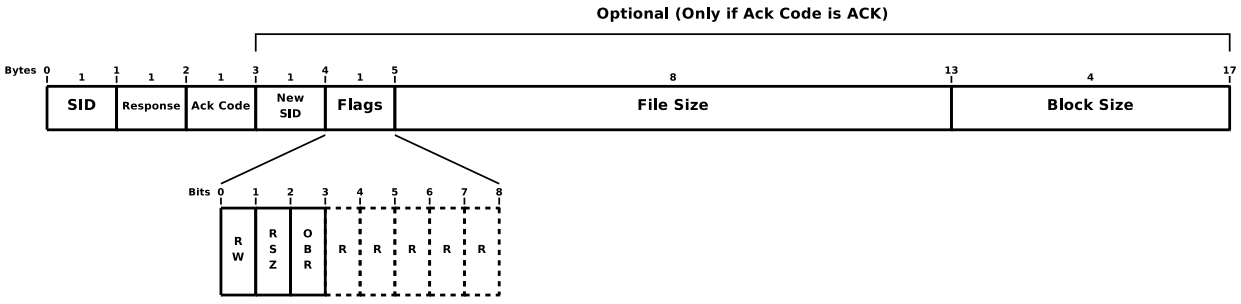

Abbildung 4.4: Antwort vom Öffnen einer Datei

- **SID** Session Identification
  Die SID einer Server Antwort muss immer dieselbe der Anfrage sein. Daher muss die SID des Open Response 0 betragen.

- **Response**
  Das Feld Response muss auf RESPONSE_OPEN_FILE (Wert 129, 0x81) gesetzt werden.

- **Ack Code**
  Der Acknowledgement Code kann den Wert ACK (Wert 0) für eine positive Bestätigung oder NAK (Wert 1) für eine negative Bestätigung besitzen. Wenn der Ack Code gleich NAK ist, dann ist die Antwort zu Ende und es werden keine weiteren Felder übertragen. Beträgt der Ack Code ACK, dann folgen die Felder New SID, Flags, File Size und Block Size.

- **New SID**
  ist die SID der neuen Session, die für alle weiteren Dateizugriffe auf die geöffnete Datei verwendet werden muss.

- **Flags**

  - **RW** Read Write
    Die Datei darf gelesen und beschrieben werden. Nur wenn das RW Flag gesetzt ist, kann gegebenenfalls das Flag RSZ oder OBR gesetzt sein. Ist dieses Flag nicht gesetzt, so kann die Datei nur gelesen werden und somit auch nicht in ihrer Größe verändert werden. Somit dürfen bei einem nicht gesetzten RW Flag, die Flags RSZ und OBR ebenfalls nicht gesetzt sein.
  - **RSZ** Resizable
    Die Datei kann vergrößert und verkleinert werden. Die kleinstmögliche Größenänderung ist ein Byte. Ist dieses Flag gesetzt, dann darf das Flag OBR nicht gesetzt sein.
  - **OBR** Only Blocksize Resizable
    Die Dateigröße ist nur um ein Vielfaches der Blockgröße veränderbar. Dieses Flag kann nur gesetzt sein, wenn RSZ nicht gesetzt ist.

- **File Size**
  gibt die Größe der Datei in Byte an.

- **Block Size**
  ist die Größe eines Blocks in Byte. Die Anzahl der Blocks lässt sich durch File Size / Block Size errechnen. Die File Size muss nicht unbedingt ein genaues Vielfaches der Block Size betragen. Die Block Size muss größer gleich 1 Byte sein. Wenn die File Size unbedingt ein Vielfaches der Block Size sein muss, und die Datei in ihrer Größe erweiterbar ist, dann ist üblicherweise das OBR und nicht das RSZ Flag gesetzt.

4.3.5 Close Request (Anfrage vom Client zum Server)

Close Request wird zum Schließen einer Session und somit indirekt zum Schließen einer offenen Datei bzw. zum Trennen der Verbindung benötigt.

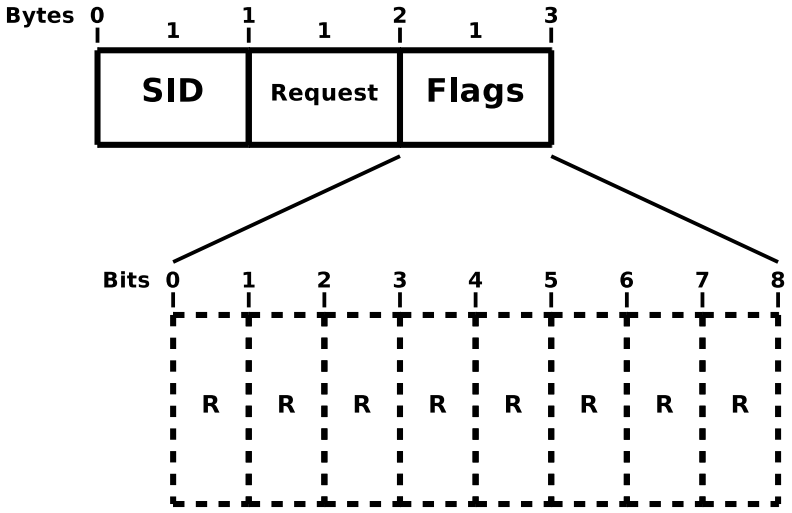

Abbildung 4.5: Anfrage zum Schließen einer Datei

- **SID** Session Identification
  ist die SID der zu schließenden Session. Beträgt der Wert der SID zwischen 1 und 255, dann wird die jeweilige Session und die dazugehörige Datei am Server geschlossen. Wird die SID 0 verwendet, dann werden alle offenen Sessions der geöffneten Dateien geschlossen und anschließend die Session 0 und somit die Verbindung zum Server getrennt.

- **Request**
  Das Feld Request muss auf REQUEST_CLOSE_FILE (Wert 2, 0x02) gesetzt werden.

- **Flags**

  Derzeit keine Verwendung.

4.3.6 Close Response (Antwort vom Server zum Client)

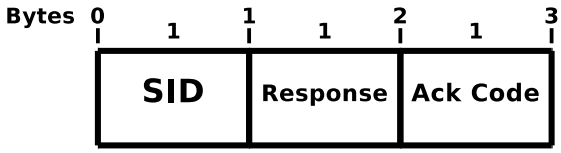

Abbildung 4.6: Antwort vom Schließen einer Datei

- **SID** Session Identification
  beinhaltet die SID der zu schließenden Session
- **Response**
  Das Feld Response muss auf RESPONSE_CLOSE_FILE (Wert 130, 0x82) gesetzt werden.
- **Ack Code**

  ACK Session erfolgreich geschlossen

  NAK Session konnte nicht geschlossen werden

4.3.7 Read Request (Anfrage vom Client zum Server)

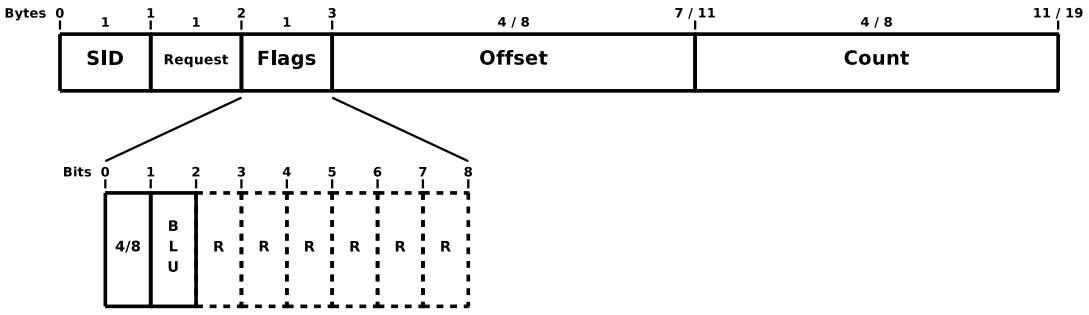

Abbildung 4.7: Anfrage zum Lesen einer Datei

- **SID** Session Identification
  ist die SID der zu lesenden Datei
- **Request**
  Das Feld Request muss auf REQUEST_READ_FILE (Wert 3, 0x03) gesetzt werden.
- **Flags**
  - **4/8** Use 4 or 8 Byte Fields
    Dieses Flag bestimmt, ob für die Felder Count und Offset jeweils 4 Byte pro Feld verwendet werden oder ob 8 Byte pro Feld verwendet werden

sollen. Durch die 8 Byte Felder ist auch ein bytebasierter Zugriff auf Datenträger, die größer als 4GB sind, möglich.

– **BLU** Block Size Unit
Das Flag bestimmt, ob der Zugriff byte- oder block-basierend stattfindet. Wenn das Byte nicht gesetzt ist, dann erfolgt der Zugriff bytebasierend und die Einheit für die Felder Count und Offset ist ein Byte. Wenn das BLU Flag gesetzt ist, dann erfolgt der Zugriff block-basierend und die Einheit für die Felder Count und Offset ist ein Block.

- **Offset**
ist der Offset der Leseposition, bezogen auf den Dateianfang. Je nach Flag ist die Einheit des Offsets ein Byte oder ein Block.

- **Count**
enthält die Anzahl der zu lesenden Bytes bzw. zu lesenden Blöcke

4.3.8 Read Response (Antwort vom Server zum Client)

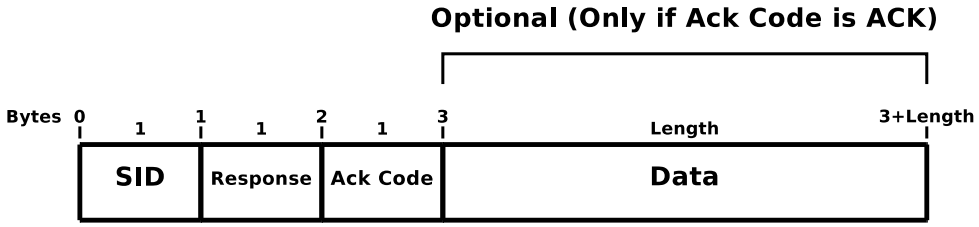

Abbildung 4.8: Antwort vom Lesen einer Datei

- **SID** Session Identification
SID der gelesenen Datei

- **Response**
Das Feld Response muss auf RESPONSE_READ_FILE (Wert 131, 0x83) gesetzt werden.

- **Ack Code**

  ACK Daten erfolgreich gelesen. Optionales Feld Data folgt.

  NAK Lesen fehlgeschlagen. Es folgen keine Daten (Data).

- **Data**
beinhaltet die gelesenen Daten, die vom Read Request angefordert wurden. Die Länge der Daten (Length) lässt sich anhand der verwendeten Werte des Read Requests berechnen. Wurde das Flag BLU gesetzt, dann berechnet sich die Länge folgenderweise:
Length = Count * Block Size
Im anderen Fall, wenn das BLU Flag nicht gesetzt war, handelt es sich um einen bytebasierten Lesezugriff und die Länge ist gleich der Anzahl der zu lesenden Bytes (Length = Count).

4.3.9 Write Request (Anfrage vom Client zum Server)

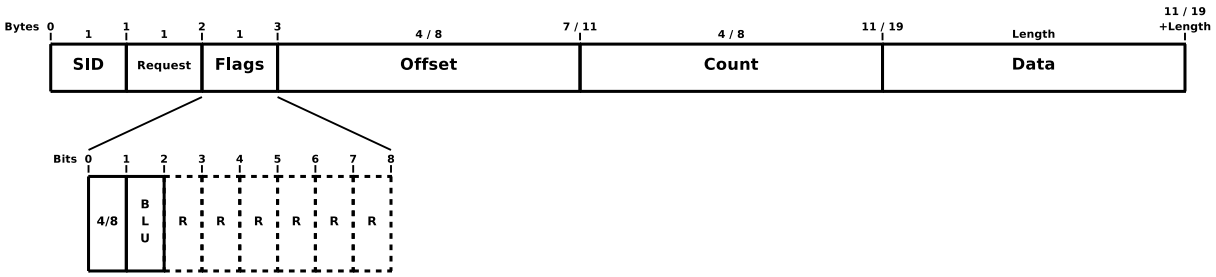

Abbildung 4.9: Anfrage zum Schreiben in eine Datei

- **SID** Session Identification
  ist die SID der zu schreibenden Datei
- **Request**
  Das Feld Request muss auf REQUEST_WRITE_FILE (Wert 4, 0x04) gesetzt werden.
- **Flags**
  - **4/8** Use 4 or 8 Byte Fields
    Dieses Flag bestimmt, ob für die Felder Count und Offset jeweils 4 Byte pro Feld verwendet werden oder ob 8 Byte pro Feld verwendet werden sollen. Durch die 8 Byte Felder ist auch ein bytebasiertes Schreiben auf Datenträgern, deren Größe 4GB überschreiten, möglich.
  - **BLU** Block Size Unit
    Das Flag bestimmt, ob der Zugriff byte- oder block-basierend stattfindet. Wenn das Byte nicht gesetzt ist, dann erfolgt der Zugriff bytebasierend und die Einheit für die Felder Count und Offset ist ein Byte. Wenn das BLU Flag gesetzt ist, dann erfolgt der Zugriff block-basierend und die Einheit für die Felder Count und Offset ist ein Block.
- **Data**
  beinhaltet die zu schreibenden Daten. Die Länge der Daten (Length) ist vom Wert Count und vom Flag BLU abhängig.
  - BLU Flag nicht gesetzt:
    Length = Count
  - BLU Flag gesetzt:
    Length = Count * Block Size

4.3.10 Write Response (Antwort vom Server zum Client)

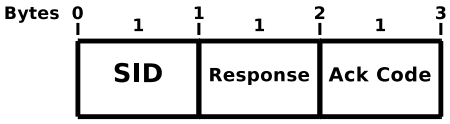

Abbildung 4.10: Antwort vom Schreiben in eine Datei

- **SID** Session Identification
  beinhaltet die SID der geschriebenen Datei
- **Response**
  Das Feld Response muss auf RESPONSE_WRITE_FILE (Wert 132, 0x84) gesetzt werden.
- **Ack Code**

  ACK Daten erfolgreich geschrieben.

  NAK Schreiben der Daten fehlgeschlagen.

4.3.11 Resize Request (Anfrage vom Client zum Server)

Mit dem Resize Request kann die Größe einer Datei geändert werden. Es ist ein Vergrößern und ein Verkleinern der Datei möglich. Damit der Resize Request erfolgreich ausgeführt werden kann, muss die Datei mit dem Flag RSZ oder OBR geöffnet worden sein. Das Vergrößern einer Datei ist auch mithilfe des Write Requests möglich.

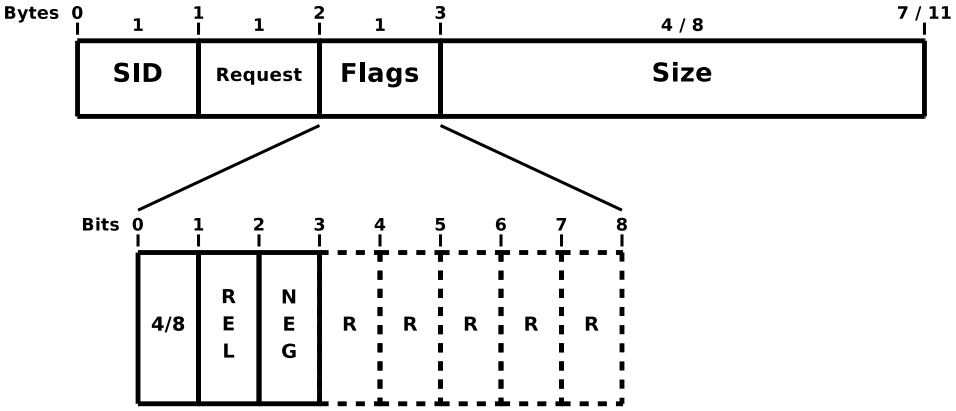

Abbildung 4.11: Anfrage zum Ändern der Größe einer Datei

- **SID** Session Identification
  die SID der Datei, deren Größe geändert werden soll
- **Request**
  Das Feld Request muss auf REQUEST_RESIZE_FILE (Wert 5, 0x05) gesetzt werden.

- **Flags**
  - **4/8** Use 4 or 8 Byte Fields
    Dieses Flag bestimmt, ob das Feld Size 4 Byte oder 8 Byte Länge besitzt.
  - **REL** Relative
    Die Größenänderung kann relative oder absolut angegeben werden. Ist dieses Flag nicht gesetzt, dann erfolgt die Größenangabe absolut. Das bedeutet, dass die neue Dateigröße gleich dem Wert von Size entspricht. Ist das Flag REL gesetzt, dann erfolgt die Größenänderung relative. Das bedeutet, dass der Wert von Size zur aktuellen Dateigröße addiert oder subtrahiert wird. Ob eine Addition oder Subtraktion durchgeführt wird, ist vom Flag NEG abhängig. Das Flag NEG wird nur berücksichtigt, wenn dieses Flag (REL) gesetzt ist.
  - **NEG** Negative
    Dieses Flag bestimmt, ob bei einer relativen Größenänderung der Wert von Size negative oder positive ist und somit zur aktuellen Größe addiert oder subtrahiert werden muss.

4.3.12 Resize Response (Antwort vom Server zum Client)

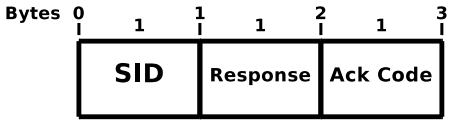

Abbildung 4.12: Antwort der Größenänderung einer Datei

- **SID** Session Identification
  ist die SID der Datei, deren Größe geändert wurde.
- **Response**
  Das Feld Response muss auf RESPONSE_RESIZE_FILE (Wert 133, 0x85) gesetzt werden.
- **Ack Code**

  ACK Größe erfolgreich geändert.

  NAK Größenänderung fehlgeschlagen.

Kapitel 5

Problem des ersten Schlüssels

Damit die steganographisch versteckte Datei aus den Stego-Objekten (StorageFiles) extrahiert werden kann, muss der Schlüssel für das erste StorageFile bekannt sein. Jedoch, wo soll dieser erste Schlüssel aufbewahrt werden?

Ein weiteres Problem ist, dass nicht nur ein Schlüssel für das Extrahieren des ersten StorageFiles ausreicht, sondern eine Vielzahl von weiteren Parametern notwendig sind. Beispielsweise müssen auch der Dateiname und der Pfad des ersten StorageFiles bekannt sein, damit dieses auch gefunden werden kann. Weiters muss zum Extrahieren der verwendete Einbettungs-Algorithmus bekannt sein und wenn vor der Einbettung die Daten noch verschlüsselt wurden, sind auch noch die Informationen für die Entschlüsselung notwendig. All diese Informationen bilden den sogenannten Parametersatz, der für das Extrahieren der versteckten Daten des ersten StorageFiles benötigt wird.

5.1 Parametersatz für das erste StorageFile

Der Parametersatz setzt sich in seinen Grundzügen aus den ASN.1 Strukturen StorageFile, StegoOptions und KryptoOptions zusammen. So beinhaltet er folgende Parameter:

- filename
- paths
  - path 1
  - path 2
  - ...
- compression
- type

- kryptoOptions
  - algorithm
  - keyLength
  - key
  - cipherMode
- stegoOptions
  - algorithm
  - stegoKey
  - extOptions
  - application (optional)
  - appVersion (optional)

Das Listing 5.1 zeigt einen typischen Parametersatz für das erste StorageFile. Die Informationen für die Extraktion der restlichen StorageFiles, die für die Wiederherstellung der versteckten Datei (DistributedFile) benötigt werden, werden aus den zuvor extrahierten StorageFiles gewonnen. In diesem Beispiel sind zwei Pfade für das erste StorageFile angegeben. Das bedeutet, dass es redundant gespeichert ist.

```
filename: Pic0077.jpg
paths:
   path: F:\Bilder\Urlaub\Flachau
   path: http://lamp2.fhstp.ac.at/~tm071071/Bilder/Winterurlaub
compression: gzip
type: stego-object
kryptoOptions:
   algorithm: AES
   keyLength: 192 Bit
   key: 0dfb b150 3c8d 237f b92c 31a4 8cb7 c334 875b 0e9f 038e 9d77
   cipherMode: CBC
stegoOptions:
   algorithm: LSB
   stegoKey: "This is @ simple p@ssw0rd phr@se."
   extOptions: use as passphrase
```

Listing 5.1: Beispiel für einen Parametersatz

Das Problem am Parametersatz ist, dass die Werte der einzelnen Parameter für einen Menschen nur sehr schwer zu merken sind. Daher muss der Parametersatz für das erste StorageFile auf sichere Art und Weise gespeichert werden, sodass kein unbefugter Dritter die Möglichkeit hat, ihn in die Hände zu bekommen.

5.2 Chipkarte, RFID

Eine sehr sichere Möglichkeit bietet die Chipkarte. Der Parametersatz wird auf der Chipkarte gespeichert und durch eine sichere Authentifizierung geschützt. Eine Chipkarte erlaubt eine ein- oder mehrstufige Authentifizierung. Als Authentifizierung kann

beispielsweise ein PIN-Code oder Biometrie-Daten des Benutzers zum Einsatz kommen. Im Falle eines PIN-Codes würde schon ein vierstelliger Code den Sicherheitsanforderungen nachkommen, denn bei einer Chipkarte kann die Brute-Force-Methode nicht angewendet werden. Üblicherweise wird beim dritten Fehlversuch die Chipkarte gesperrt. Somit beträgt die Wahrscheinlichkeit, den richtigen PIN-Code zu erraten, 0,03 Prozent. Der Vorteil des PIN-Codes ist, dass er für einen Menschen sehr leicht zu merken ist. Natürlich können auch andere Authentifizierungsmethoden verwendet werden. Generell bietet die Chipkarte bei richtiger Verwendung eine sichere Möglichkeit den Parametersatz aufzubewahren, jedoch ergeben sich durch dadurch auch einige Nachteile, die aufgezeigt werden müssen. Der größte Nachteil ist die zusätzliche Hardware bzw. Peripherie und die dadurch notwendigen Zusatzinstallationen. (vgl. Rankl und Effing, 2008)

Folgende Punkte werden zusätzlich beim Einsatz einer Chipkarte benötigt:

- Chipkarte
- Peripheriegerät zum Programmieren der Chipkarte (Parametersatz, Authentifizierung) und Auslesen des gespeicherten Parametersatzes
- Installation des Treibers und der Software für das Peripheriegerät

Aus diesem Grund werden andere Möglichkeiten zum sicheren Aufbewahren des Parametersatzes gesucht, die keine zusätzliche Peripherie benötigen.

5.3 Steganographische Speicherung des Parametersatzes

Eine weitere Alternative wäre, den Parametersatz wieder in ein Cover einzubetten. Dazu könnte ein steganographisches Programm verwendet werden, das für die Einbettung und Extraktion nur ein Passwort benötigt. Somit reduziert sich der zu merkende Parametersatz auf ein einziges Passwort. Das Programm *steghide* würde sich für diesen Zweck eignen. Es benutzt standardmäßig einen graph-theoretischen Ansatz für die Einbettung. Dabei werden die einzubettenden Daten zuerst komprimiert und anschließend mit dem AES-Algorithmus verschlüsselt. Der Schlüssel für die Einbettung und Verschlüsselung wird jeweils vom Passwort abgeleitet. Somit liegt die sichere Aufbewahrung des Parametersatzes einzig und alleine im verwendeten Passwort. Und genau hier liegt das Problem, denn ein leicht zu merkendes Passwort kann nicht als sicher betrachtet werden. Für ein sicheres Passwort sollte eine längere Passwort-Phrase verwendet werden, die im Idealfall viele Sonderzeichen besitzt und keine vollständigen Wörter enthält. Zum Generieren einer Passwort-Phrase kann ein Buch, das dem Benutzer immer zu Verfügung steht, benutzt werden. Beispielsweise könnte aus dem Buch *Windows Internals* (4. Auflage, deutsch) von der Seite 234 jeweils der erste Buchstabe jeder Zeile und anschließend jeder letzte Buchstabe als Passwort-Phrase gewählt werden.

Damit würde sich folgende Phrase ergeben:
DdDdWddKZpusdsMCg[TTTc{[[}ATOCsuetIdKKdEdsbl-m-.ee.n-e-t-e_t.:::]r{;;;ee_dem-.rret-f—.

Jedesmal wenn der Parametersatz benötigt wird, muss das entsprechende Buch verfügbar sein und aus der Seite 243 die Passwort-Phase generiert werden. Jedoch selbst diese Passwort-Phrase ist nicht wirklich ausreichend sicher, da die einzelnen Zeichen nicht zufällig, sondern stark von der verwendeten Sprache im Buch abhängig sind. Generell ist bei dieser Aufbewahrung des Parametersatzes das Passwort das Problem. Wenn es zu leicht gewählt wird, kann es durchprobiert werden und wenn es den Anforderungen entspricht, benötigt der Mensch üblicherweise ein weiteres Hilfsmittel (wie hier beispielsweise das Buch) um das Passwort erneut abrufen zu können. Somit bietet diese Möglichkeit auch keine optimale Lösung für die Aufbewahrung des Parametersatzes.

5.4 Aufbewahrung des Parametersatzes im Internet

Das Internet bietet neue Formen für das Aufbewahren des Parametersatzes, denn das Internet besitzt zwei Eigenschaften die der sicheren Aufbewahrung sehr zugute kommen.

- Das Internet verfügt über eine unüberschaubare Anzahl von Adressen bzw. URLs. Somit ist es unmöglich, alle Adressen auf gespeicherte Parametersätze zu überprüfen.
- Eine Authentifizierung ist im Internet nur sehr schwer zu knacken bzw. zu umgehen. (Vorausgesetzt, der Server besitzt keine kritischen Sicherheitslücken) Dieser Umstand beruht darauf, dass eine Brute-Force- und Dictionary-Attacke nur sehr begrenzt möglich sind, denn einerseits steigt die Durchführungszeit durch die Latenzzeit des Internets und des Servers in nicht vertretbare Dimensionen und andererseits kann der Server zusätzliche Regeln, wie beispielsweise zehn Authentifizierungsversuche pro IP pro Stunde, festlegen.

5.4.1 HTTPS und Authentifizierung

Eine einfache Variante zur Aufbewahrung des Parametersatzes bietet ein Webserver. In einer Web-Seite kann im Klartext der Parametersatz gespeichert werden. Diese Seite sollte nur über das HTTPS-Protokoll und nicht über das HTTP-Protokoll erreichbar sein. Dadurch ist eine abhörsichere Datenübertragung gewährleistet. Weiters muss zusätzlich zur SSL/TLS-Authentifizierung eine Authentifizierung des Clients erfolgen, die den unbefugten Zugriff eines Dritten auf den Parametersatz verhindern soll. Beispielsweise lässt sich der Zugriffsschutz durch eine einfache *.htpasswd*-Datei auf einem NCSA-kompatiblen Webserver realisieren.

Leider können nicht auf jedem Webaccount, speziell bei den Gratis-Webaccounts, die notwendigen Einstellungen für diese Methode vorgenommen werden. Diese Variante bietet sich bei einem eigenen Webserver an, bei dem der Besitzer problemlos alle notwendigen Konfigurationseinstellungen vornehmen kann. Jedoch sollte man bedenken, dass bei einer Beschlagnahmung des Laptops mit einer hohen Wahrscheinlichkeit auch der eigene Webserver sichergestellt wird und somit der Parametersatz den Forensikern offen in die Hände gelegt wird.

5.4.2 Steganographische Speicherung in Foren, Free-Webspaces, etc.

Eine weitere Methode bieten die öffentlichen Foren und Free-Webspaces. Üblicherweise erlauben die meisten Foren, ein Bild zum Eintrag hinzuzufügen oder eine beliebige Datei anzuhängen. Somit kann problemlos ein Stego-Objekt in einem Forum gespeichert werden. Ebenfalls dafür geeignet sind frei verfügbare Webspaces. So können beispielsweise einfache HTML-Seiten generiert werden, die Bilder mit den eingebetteten Parametersatz beinhalten. Denkbar wäre auch eine Einbettung in Videos, die anschließend zum Beispiel auf Youtube hochgeladen werden. Jedoch sollte hier bedacht werden, dass die hochgeladenen Videos ins Flash Videoformat konvertiert werden und daher ein sehr robustes Einbettungsverfahren gewählt werden muss. Für die Einbettung des Parametersatzes könnte ein öffentlich bekannter Basis-Parametersatz mit einem einfach gewählten Stego-Schlüssel verwendet werden, denn selbst wenn das Stego-Objekt erkannt wird und der eingebettete Parametersatz extrahiert werden kann, ist es für den jeweiligen Betreiber des Forums bzw. der öffentlichen Plattform nicht einfach möglich, auf den Besitzer zu schließen. Problematisch wird diese Methode, wenn die eigentlichen StorageFiles ebenfalls im Internet frei zugänglich sind, denn in diesem Fall kennt der unbefugte Dritte zwar den Eigentümer der versteckten Datei (DistributedFile) nicht, kann jedoch die versteckte Datei extrahieren und lesen. Um dieses Problem zu lösen, kann der Pfad oder nur die Domain des ersten StorageFiles aus dem Parametersatz entfernt werden. Diese Variante bietet den Vorteil, dass ein Forensiker, der Stego-Objekte auf dem beschlagnahmten Laptop vermutet und den notwendigen Parametersatz sucht, unmöglich alle URLs von Foren, Websites, freien Portalen auf Stego-Objekte, die den richtigen Parametersatz gespeichert haben, untersuchen kann.

Steht aber ein Benutzer unter Verdacht, gesetzeswidrige Handlungen zu tätigen (und sei es nur der kritische Bericht in einem zensurierten Land), ist davon auszugehen, dass sein Internetanschluss mitaufgezeichnet wird. In diesem Fall wäre das Stego-Objekt, das den Parametersatz beinhaltet, erheblich leichter zu finden. Wenn für das Stego-Objekt der öffentliche Basis-Parametersatz mit einem leicht gewählten Stego-Schlüssel benutzt wurde, ist es eventuell sogar extrahierbar.

Kapitel 6

Möglicher Existenzbeweis durch Forensik

Die IT-Forensik, Computer-Forensik bzw. Digitale Forensik beschäftigt sich mit der Feststellung des Tatbestandes von IT-Systemen und der Erfassung, Analyse und Auswertung von digitalen Spuren in Computersystemen. Vor der Analyse werden forensische Duplikate der beschlagnahmten Datenträger erstellt. Das *Distributed File Storage* Programm muss auch den Fall der Beschlagnahmung des Laptops berücksichtigen. Deshalb sollten wenn möglich keine Spuren, die auf die Existenz einer steganographisch versteckten Datei schließen lassen, existieren. Als Grundbedingung muss das Programm ein sicheres Einbettungsverfahren für die Covers benutzen, damit keine erfolgreiche Steganalyse an den Stego-Objekten möglich ist. Dadurch erscheinen die Stego-Objekte, die beispielsweise Bilder, Videos und Musiklieder sein können, als unauffällige Dateien. Doch für die Verschleierung der versteckten Datei müssen noch weitere Punkte berücksichtigt werden. Erstens sollten sich die Zeitstempel einer Datei durch die Einbettung nicht ändern und zweites sollte bei der Beschlagnahmung kein steganographisches Programm auf dem Computer abgespeichert sein. (vgl. Geschonneck, 2008)

6.1 Zeitstempel einer Datei

Durch die geänderten Zeitstempel der Stego-Objekte können eventuell Muster entstehen, die auf den Einsatz eines steganographischen Programms schließen lassen. Natürlich ändert sich auch der Zeitstempel einer Bilddatei durch die Bearbeitung mit einem Bildbearbeitungsprogramm, jedoch wenn anhand des Zeitstempels festgestellt werden kann, dass sich innerhalb einer Minute alle 100 Bilddateien auf einem USB-Stick geändert haben, weist dieser Umstand nicht unbedingt auf die Benutzung eines Bildbearbeitungsprogramms hin. Somit sollte der Zeitstempel bei einer Einbettung unverändert bleiben.

Jede Datei unter Windows besitzt folgende drei Zeitstempel:

- Creation Time (Erstelldatum)
- Last Access Time (letzter Zugriff)
- Last Write Time (letzte Änderung)

Unter unixoiden Betriebssystemen werden folgende Zeitstempel pro Datei gespeichert:

- Last Access Time (letzter Zugriff)
- Last Modification Time (letzte Änderung des Dateiinhaltes)
- Last Change Time (letzte Änderung der Besitzer, Gruppe, Rechte, etc.)

Damit das steganographische Programm keine Spuren hinterlässt, sollten die Zeitstempel *Last Access Time* und *Last Write Time* unter Windows und *Last Access Time* und *Last Modification Time* unter Linux entsprechend auf den ursprünglichen Wert zurückgesetzt werden. Hierfür muss das Programm die Zeitstempel vor der Benutzung der Datei auslesen und nach deren Verwendung wieder zurücksetzen. Unter Windows stehen dafür die API-Funktionen *GetFileTime* und *SetFileTime* zur Verfügung. POSIX-kompatible Betriebssysteme bieten für diesen Zweck die Systemaufrufe *stat* und *utime* an. Durch die richtige Verwendung dieser Funktionen sollten die forensischen Spuren, die durch die Zeitstempel-Änderungen entstehen, verhindert werden. (vgl. Microsoft Corp., 2009) (vgl. Linux, 1998) (vgl. Linux, 1995)

6.2 Existenz des steganographischen Programms

Die Existenz eines steganographischen Programms lässt vermuten, dass Daten am Laptop, USB-Stick oder im Internet versteckt wurden. Es kann zwar anhand des Programms die Nutzung dessen nicht bewiesen werden, jedoch wird es einen Forensiker misstrauisch stimmen und ihn dazu veranlassen, sich mehr auf die Steganalyse von potentiellen Stego-Objekten zu konzentrieren. Aus diesem Grund sollte das Programm immer nur temporär gespeichert werden. Beispielsweise könnte am System eine RAM-Disk eingerichtet werden, die für die Speicherung des steganographischen Programms verwendet wird. Weiters sollte das Programm keine zusätzliche Installationsroutine benötigen, damit keine Einträge im System (z.B: in der Registry unter Windows) vorgenommen werden. Das Programm selbst könnte jedesmal vor deren Verwendung vom Internet heruntergeladen werden.

Eine weitere Herausforderung ist der spurlose Zugriff auf die versteckte Datei. Diese Problematik ist in Kapitel 2 (siehe Abschnitt 2.5) beschrieben. In diesem Zusammenhang muss auch darauf geachtet werden, dass das Programm (Editor, Word, ...), mit dem die versteckte Datei gelesen oder bearbeitet wird, keine Sicherheitskopien auf der Festplatte für eventuelle Datei-Wiederherstellungen ablegt.

Kapitel 7

Fazit und Ausblick

Ziel dieser Arbeit war es, die steganographische Speicherung von Daten unabhängig von der verwendeten Einbettungstechnologie zu standardisieren. Dabei wird die versteckte Speicherung einer beliebigen Datei in mehreren Covers, die unterschiedliche Einbettungsalgorithmen verwenden können, unterstützt. Durch das beschriebene Verfahren im Kapitel 3 soll zukünftig die Interoperabilität zwischen steganographischen Programmen ermöglicht werden. Im Weiteren wird nun auf die Punkte der forschungsleitenden Fragestellungen (siehe Abschnitt 1.6) eingegangen, die wie folgt zusammengefasst werden können:

- Standardisiertes Verfahren zur versteckten Speicherung
- Sichere Aufbewahrung des Parametersatzes
- Beseitigung von forensischen Spuren

7.1 Standardisiertes Verfahren

Im Zusammenhang dieser Arbeit wurde ein standardisiertes Verfahren, das im Kapitel 3 beschrieben ist, entwickelt. Alle weiteren notwendigen Komponenten für das standardisierte Verfahren wurden im Kapitel 4 beschrieben und teilweise für Testzwecke implementiert. Für den problemlosen Zugriff auf die versteckten Dateien wurde das File Operation Protokoll entworfen. Das Verfahren zur versteckten Speicherung von Dateien benutzt ASN.1-Strukturen, die alle notwendigen Informationen zum Auslesen und Bearbeiten der Datei beinhalten. Die ASN.1-Strukturen sind neben den eigentlichen Daten der versteckten Datei ebenfalls in den Stego-Objekten gespeichert. Im Anhang A sind alle notwendigen ASN.1-Strukturen aufgelistet. Damit sich dieses Verfahren als Standard durchsetzen kann, muss es erst einmal vollständig als Software-Anwendung implementiert werden. Schließend wird die Zukunft zeigen, ob es sich als Standard durchsetzen kann und ob es als standardisiertes Verfahren geeignet ist.

7.2 Sichere Aufbewahrung des Parametersatzes

Die Problematik zur sicheren Aufbewahrung des Parametersatzes für das erste StorageFile bzw. Stego-Objekt wird im Kapitel 5 behandelt. Grundsätzlich kann der Schluss gezogen werden, dass nach heutigem Standard die Chipkarte eine wirklich sichere Methode zum Aufbewahren des Parametersatzes bietet. Die anderen beschriebenen Alternativen werden in den meisten Fällen ausreichen, jedoch können sie nicht als wirklich sicher betrachtet werden. Sie bieten dafür den Vorteil, dass keine zusätzliche Peripherie benötigt wird. Als Grundregel gilt: Je sicherer die Aufbewahrung des Parametersatzes sein soll, desto mehr Aufwand muss dafür betrieben werden.

7.3 Beseitigung von forensischen Spuren

Die Kernprobleme, die sich durch den Einsatz eines steganographischen Programms ergeben, wurden im Kapitel 6 abgehandelt. Speziell wurde hier auf die möglichen Hinweise für eine steganographische Speicherung eingegangen. Dies beinhaltet die Problematik der Zeitstempel-Änderungen von Stego-Objekten und die Existenz des Programms an sich. Wenn diese beiden Hauptprobleme gelöst sind, müssen noch die üblichen Fundstellen behandelt werden, wie beispielsweise Logdateien des Betriebssystems, eventuell gespeicherte Fragmente der versteckten Datei, die durch die Speicherauslagerung entstehen können, und sonstige Eigenheiten des jeweiligen Betriebs- und Dateisystems. Schlussendlich bewirkt dies in den meisten Fällen ein Kopf-an-Kopf-Rennen zwischen den Forensikern und Programmierern, die die steganographische Software erstellen.

7.4 Ausblick

Diese Arbeit soll einen Grundstein für ein standardisiertes Verfahren zur versteckten Speicherung legen. Als nächster Schritt muss dieses Konzept in die Realität umgesetzt werden, um eventuelle Schwächen, die derzeit noch nicht bekannt sind, aufzuzeigen und zu beheben. Schlussendlich kann die Implementierung dieses Verfahren als Programmbibliothek für die Verwendung in steganographischen Programmen angeboten werden und somit dessen Verbreitung und Akzeptanz bewirken. Des Weiteren könnte dadurch das Ziel, die Interoperabilität zwischen steganographischen Programmen zu gewährleisten, erreicht werden.

Anhang A

ASN.1 Definitionen

```
DistributedFileStorage DEFINITIONS ::=
BEGIN

DistributedFile ::= [PRIVATE 1] SEQUENCE
{
    filename              UTF8String OPTIONAL,
    blockSize             INTEGER,
    size                  FileSize,
    changeable            FileChangeable
}

FileSize ::= [PRIVATE 2] SEQUENCE
{
    size                  INTEGER
}

FileChangeable ::= [PRIVATE 3] SEQUENCE
{
    readWrite             BOOLEAN,
    resizable             BOOLEAN,
    onlyBlockResizable    BOOLEAN
}

Directory ::= [PRIVATE 4] SEQUENCE
{
    id          INTEGER,
    rootDirId   INTEGER,
    name        UTF8String
}

StorageFile ::= [PRIVATE 5] SEQUENCE
{
    id              INTEGER,
    filename        UTF8String,
    directoryId     INTEGER,
    redundantDirId  SET OF INTEGER OPTIONAL,
    maxBytes        INTEGER,
    overwritable    BOOLEAN,
    compression     INTEGER {
        none          (0),
        krypto        (1),
        stego         (2)
        -- noch weitere --
    },
    type            ENUMERATED {
```

```
        binary      (0),
        krypto      (1),
        stego       (2)
    },
    stegoId         INTEGER,
    kryptoId        INTEGER,
    hash            HashData OPTIONAL,
    stegoCover      Cover OPTIONAL
}

Cover ::= [PRIVATE 6] SEQUENCE
{
    filename                    UTF8String OPTIONAL,
    directoryId                 INTEGER OPTIONAL,
    overwriteWithStegoObject    BOOLEAN,
    deleteCover                 BOOLEAN
}

HashData ::= [PRIVATE 7] SEQUENCE
{
    algorithm       INTEGER {
        md5         (1),
        sha1        (2),
        sha256      (3),
        sha512      (4)
        -- evt. noch weitere --
    },
    hashValue       OCTET STRING
}

StoredBlocks ::= [PRIVATE 8] SEQUENCE
{
    storageId       INTEGER,
    blockGroups     SET OF BlockGroup,
    storedBytes     INTEGER OPTIONAL,
    hash            HashData OPTIONAL
}

BlockGroup ::= [PRIVATE 9] SEQUENCE
{
    distributedBlockOffset    INTEGER,
    blockCount                INTEGER
}

NextEntries ::= [PRIVATE 10] SEQUENCE
{
    storageId    INTEGER
}

StegoOptions ::= [PRIVATE 11] SEQUENCE
{
    algorithm     ENUMERATED
    {
        application (0),
        lsb         (1),
        echo        (2),
        phase       (3),
        frequency   (4)
    },
    stegoKey      OCTET STRING,
    extOptions    UTF8String OPTIONAL,
    stegoApp      StegoApplication OPTIONAL
}

StegoApplication ::= [PRIVATE 12] SEQUENCE
{
```

```
    name            UTF8String,
    version         UTF8String
}

KryptoOptions ::= [PRIVATE 13] SEQUENCE
{
    algorithm     ENUMERATED
    {
        rsa             (1),
        elGamalDL       (2),
        elGamalEC       (3),
        aes             (4)
    },
    keyLength     INTEGER OPTIONAL,
    key           OCTET STRING OPTIONAL,
    cipherMode    ENUMERATED
    {
        ecb             (1),
        cbc             (2),
        ofb             (3),
        ctr             (4)
        -- koennen noch weitere folgen --
    }
}

END
```

Listing A.1: ASN.1 Definitionen

Literaturverzeichnis

[Church 2009] CHURCH, Alfred J.: *The Story of the Persian War*. 2009. – http://heritage-literature.org/www/heritage.php?Dir=books&MenuItem=display&author=church&book=persian&story=revolt

[Dubuisson 2000] DUBUISSON, Olivier: *ASN.1 Communication between Heterogeneous Systems*. 5. Juni 2000. – http://www.oss.com/asn1/dubuisson.html, ISBN: 0-12-633361-0

[Geschonneck 2008] GESCHONNECK, Alexander: *Computer-Forensik: Computerstraftaten erkennen, ermitteln, aufklären*. 3., aktualisierte und erweiterte Auflage. 2008. – ISBN: 3898645347

[International Telecommunication Union] INTERNATIONAL TELECOMMUNICATION UNION: *X.690 : Information technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*. – http://www.itu.int/rec/T-REC-X.690-200811-P/en

[Isernhagen und Helmke 2004] ISERNHAGEN, Rolf ; HELMKE, Hartmut: *Softwaretechnik in C und C++ - Das Kompendium: Modulare, objektorientierte und generische Programmierung*. 4. Auflage. März 2004. – ISBN: 3446227156

[Johnson und Jajodia 1998] JOHNSON, Neil F. ; JAJODIA, Sushil: *Steganography: Seeing the Unseen*. 1998. – IEEE Computer, February 1998: S26-34

[Katzenbeisser und Petitcolas 2000] KATZENBEISSER, Stefan ; PETITCOLAS, Fabien A. P.: *INFORMATION HIDING techniques for steganography and digital watermarking*. 2000. – ISBN: 1-58053-035-4

[Larmouth 1999] LARMOUTH, John: *ASN.1 Complete*. 31. Mai 1999. – http://www.oss.com/asn1/larmouth.html

[Linux 1995] LINUX: *UNIX Manual Page: man 2 utime*. 10. Juni 1995. – http://www.cl.cam.ac.uk/cgi-bin/manpage?2+utime

[Linux 1998] LINUX: *UNIX Manual Page: man 2 stat*. 13. Mai 1998. – http://www.cl.cam.ac.uk/cgi-bin/manpage?2+stat

[McDonald und Kuhn 2000] MCDONALD, Andrew D. ; KUHN, Markus G.: *StegFS: A Steganographic File System for Linux*. 2000. – http://www.cl.cam.ac.uk/~mgk25/ih99-stegfs.pdf

[Microsoft Corp. 2009] MICROSOFT CORP.: *Microsoft Developer Network: File Times*. 12. Februar 2009. – http://msdn.microsoft.com/en-us/library/ms724290(VS.85).aspx

[Rankl und Effing 2008] RANKL, Wolfgang ; EFFING, Wolfgang: *Handbuch der Chipkarten: Aufbau - Funktionsweise - Einsatz von Smart Cards*. 5. Auflage. 2008. – ISBN: 3446404023

[Russinovich und Solomon 2005] RUSSINOVICH, Mark E. ; SOLOMON, David A.: *Windows Internals*. 4. Auflage. 2005. – ISBN: 3-86063-977-3

[Stallings 2003] STALLINGS, William: *Betriebssysteme*. 4., überarbeitete Auflage. 2003. – ISBN: 3-8273-7030-2

[Tanenbaum 2001] TANENBAUM, Andrew S.: *Modern Operating Systems*. 2. Auflage. 2001. – ISBN: 8120320638

[Vitaliev 2007] VITALIEV, Dmitri: *Digital Security and Privacy for Human Rights Defenders*. Februar 2007. – http://www.frontlinedefenders.org/manual/en/esecman/esecman.en.pdf, Kapitel Steganography: http://www.frontlinedefenders.org/manual/en/esecman/chapter2_8.html