Reverse Engineering proprietärer Dateisysteme

Reverse engineering of proprietary file systems

Master-Thesis von Jonas Plum

Tag der Einreichung: 5. Januar 2016

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Martin Steinebach

TECHNISCHE UNIVERSITÄT DARMSTADT

Fraunhofer SIT

Reverse Engineering proprietärer Dateisysteme
Reverse engineering of proprietary file systems

Vorgelegte Master-Thesis von Jonas Plum

1. Gutachten: Prof. Dr. Michael Waidner
2. Gutachten: Dr. Martin Steinebach

Tag der Einreichung: 5. Januar 2016

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 5. Januar 2016

(J. Plum)

Zusammenfassung

Für IT-forensische Analysen werden häufig Speichermedien verschiedener Arten untersucht. Zur Unterstützung bei dieser Aufgabe existieren verschiedene Programme, welche Teile dieser Analyse übernehmen. Diese Programme müssen die jeweils verwendeten Dateisysteme verstehen, um alle darauf vorhandenen Daten interpretieren zu können. Für neue, proprietäre Dateisysteme kann dieses Verständnis durch das Reverse Engineering des Dateisystems erlangt werden. Diese Aufgabe wurde bisher meist in einem langwierigen Prozess manuell erledigt. Die vorliegende Arbeit stellt ein mehrphasiges, weitgehend automatisiertes Konzept vor, durch das ein Dateisystem analysiert werden kann. Hierbei wird aus einem Festplattenabbild oder einem eingebundenen Volume ein Modell des Dateisystems abgeleitet.

Abstract

In digital forensics storage media of various kinds are investigated. To assist this task, there are several programs that take over some parts of this analysis. These programs need to understand the file systems used in each case in order to interpret all data. For new, proprietary file systems, this understanding can be attained through the reverse engineering of the file system. In the past this task has been done manually in a tedious process. This paper presents a multi-phase, largely automated approach, to analyse a file system. A model of the file system is derived from a hard-disk image or a mounted volume.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

1 Einleitung

Die Zahl der Computer hat in den letzten Jahren rasant zugenommen und wird auch in Zukunft mit Trends wie dem Internet of Things weiter zunehmen. Viele dieser Computersysteme besitzen eine Möglichkeit zur permanenten Speicherung von Daten, wie Festplatten oder Speicherkarten. Um Daten auf diesen Medien zu sichern, werden Dateisysteme eingesetzt. Dateisysteme beschreiben, wie Daten auf einem elektronischen Speichermedium abgelegt werden. Sie bilden somit eine Abstraktionsebene zwischen den konsekutiv gespeicherten Bytes und den Dateien und Verzeichnisse, die ein Betriebssystem den darauf laufenden Anwendungen zur Verfügung stellt. Mit neuen Computersystemen entstehen immer wieder neue Anforderungen an die Dateisysteme und daher auch neue Dateisysteme.

1.1 Problemstellung

Dateisysteme lassen sich, wie Software auch, in zwei Kategorien einordnen: Quelloffene Dateisysteme und proprietäre Dateisysteme. Beispiele für quelloffene Dateisysteme sind ext4 oder btrfs. Für beide sind viele Dokumentationen und eine Referenzimplementierung frei verfügbar. Andere Dateisysteme sind proprietäre Dateisysteme, zu denen nur wenig oder keine Dokumentation existiert. Hierdurch entstehen zwei Probleme:

1. Die Nutzung eines Speichermediums mit einem anderen Computer, welcher das proprietäre Dateisystem nicht unterstützt, ist nicht möglich. Um eine Nutzung möglich zu machen, muss für das neue Dateisystem ein Treiber geschrieben werden. Dateisystemtreiber sind oft Teil eines Betriebssystems und so nur unter einem bestimmten Betriebssystem nutzbar. Um einen Dateisystemtreiber für ein anderes Betriebssystem zu schreiben, muss man den Aufbau und die Funktionsweise des Dateisystems genau kennen.
2. Die forensische Auswertung eines Speichermediums ist kompliziert. Überwachungskameras, Drucker und andere Geräte nutzen proprietäre Dateisysteme. Sind diese Teil einer forensischen Untersuchung, sind hierzu Programme, die diese Dateisysteme kennen, hilfreich für eine schnellere Auswertung. Auch um forensische Programme mit Unterstützung für neue Dateisysteme zu erweitern, ist daher die genaue Kenntnis des Aufbaus der Dateisysteme nötig.

Aus beiden Problemen ergibt sich die zentrale Fragestellung:

> *Wie ist das zu untersuchende Dateisystem aufgebaut?*

1.2 Ziel der Arbeit

Ziel dieser Masterarbeit soll das Erstellen eines Konzeptes zur Analyse von Dateisystemen sein, welches genutzt werden kann, um unbekannte Dateisysteme zu untersuchen und den Aufbau dieser zu erkennen.

Auf Grundlage dieser Untersuchung soll dann anschließend die Erstellung und Erweiterung von Treibern für Betriebssysteme und forensischen Werkzeugen möglich sein.

Hierbei kann es zwei verschiedene Ausgangssituationen geben. Bei einem neuen Dateisystem eines Betriebssystems kann auf dieses lesend und schreibend über das Betriebssystem zugegriffen werden und es können auch Abbilder des Datenträgers erstellt werden. Abbilder von Speichermedien oder Dateisystemen sind Dateien, welche eine vollständige Kopie der Daten des Speichermediums enthalten. Eine andere Situation als bei Dateisystemen mit vollem Schreib- und Lesezugriff liegt bei unbekannten Dateisystemen vor, welche nicht in ein vorliegendes Betriebssystem eingebunden werden können. Hierbei existiert als Ausgangslage nur ein Abbild einer Partition, die mit dem Dateisystem formatiert wurde, und kein Zugriff auf einzelne Dateien darauf.

Um die Struktur eines proprietären Dateisystems zu verstehen, bietet sich die Möglichkeit dieses mittels Reverse Engineering zu analysieren. Reverse Engineering beschreibt im Allgemeinen die Untersuchung eines Produktes, um hieraus dessen Aufbauanleitung zu erzeugen. In der Informatik gibt es verschiedene Bereiche, in denen Reverse Engineering eingesetzt wird. So werden Programme, wie z. B. Malware, untersucht, um ihre genaue Funktion kennenzulernen.

Um einen eigenen Ansatz zum automatisierten Reverse Engineering von Dateisystemen zu erstellen, sollen in dieser Arbeit Ansätze im Reverse Engineering von Dateisystemen und anderen Datenstrukturen dargelegt und gegenübergestellt werden. Auf Grundlage dieser Ansätze soll dann ein Konzept erstellt werden, welches zum weitestgehend optimierten und automatisierten Reverse Engineering von Dateisystemen genutzt werden kann.

Auf Basis des Konzepts soll ein Programm erstellt werden, welches das Konzept praktisch umsetzt. Dieses Programm soll dabei möglichst automatisiert zum Reverse Engineering von Dateisystemen eingesetzt werden können. Evaluiert werden soll das Konzept, indem das erstellte Programm auf das proprietäre ReFS (Resilient File System) angewandt wird.

Nicht behandelt wird in dieser Arbeit ein Reverse Engineering der Dateisystemtreiber. Diese liegen nicht immer für Untersuchungen vor oder können nicht einfach extrahiert und analysiert werden.

1.3 Aufbau der Arbeit

Die Masterarbeit gliedert sich in folgende Kapitel: Zunächst werden in Kapitel 2 verschiedene gängige Dateisysteme vorgestellt und anschließend verglichen. Hierbei wird vor allem auf die Strukturen, welche in diesen Dateisystemen existieren, eingegangen und diese werden verschiedenen Kategorien zugeordnet. Das anschließende Kapitel 3 beschreibt die Dateisystemanalyse und grenzt diese vom Dateisystem-Reverse-Engineering ab. Kapitel 4 gliedert diese Arbeit in den aktuellen Stand der Forschung ein. Hierzu werden sowohl Arbeiten zum Reverse Engineering verschiedener Dateisysteme betrachtet als auch Konzepte zu automatischem Reverse Engineering. In Kapitel 5 werden die Bedingungen an ein Konzept, welches zum automatisierten Reverse Engineering von Dateisystemen genutzt werden kann, beschrieben. Kapitel 6 beschreibt den Kern dieser Arbeit. Auf Basis der vorher beschriebenen Arbeiten wird das Reverse-Engineering-Konzept aufgestellt. Hierzu wird ein mehrstufiger Prozess entworfen und im Detail

erläutert. Die Implementierung dieses Konzeptes wird in Kapitel 7 beschrieben. Anschließend wird in Kapitel 8 das Konzept sowie die Implementierung bewertet und diskutiert. Zuletzt folgt Kapitel 9, in dem die Arbeit sowie deren Ergebnisse zusammengefasst werden und ein Ausblick auf offene Fragen gegeben wird.

2 Dateisysteme

Dateisysteme bilden die Basis zu persistenten Speicherung von Daten in Betriebssystemen. Ein Dateisystem organisiert, in welcher Art und Weise Dateien auf einem Volume abgelegt werden. Der Dateisystemtreiber ist dabei eng mit dem Betriebssystem verbunden und Anwendungen des Betriebssystems steht eine einfachere Zugriffsmöglichkeit auf Dateien und Verzeichnisse zur Verfügung.

Die Grundlage zum Speichern bieten elektronische Speichermedien wie Festplatten, CDs oder Speicherkarten. Diese physikalischen Speichermedien bestehen aus vielen adressierbaren Speichereinheiten, den Sektoren. Die Größe dieser Sektoren liegt bei älteren Festplatten in der Regel bei 512 Byte, bei neueren Festplatten inklusive Solid-State-Drives (SSDs) bei 4 KiB. Dateisysteme fassen diese Sektoren in logische Blöcke zusammen, welche je nach Dateisystem unterschiedlich bezeichnet werden. NTFS und FAT nutzen hierfür den Begriff „Cluster“, ext und HFS+ „Block“ und Brian Carrier „data unit“ [6]. In dieser Arbeit wird im folgenden der Begriff „Block“ verwendet. Unter anderem um die Abnutzung bei SSD-Festplatten gering zu halten, bietet sich auch auf Dateisystemebene an die Blockgröße auf 4096 Bytes oder ein Vielfaches davon zu setzen.

Die Größe moderner Festplatten macht eine möglichst effiziente Ausnutzung des Speicherplatzes weniger wichtig. Je nach Einsatzzweck können Speichermedien als ein einzelnes Volume oder in mehrere Volumes eingeteilt werden. Es können auch mehrere Speichermedien als ein Volume zusammengefasst werden, so wie es z. B. in RAID-Konfigurationen der Fall ist. Diese Volumes bieten Betriebssystemen und Anwendungen eine Möglichkeit Sektoren zu adressieren, als ob diese direkt fortlaufend vorliegen würden. Strukturen wie der Master Boot Record oder die GUID Partition Table bieten die Möglichkeit ein Volume in mehrere Partitionen einzuteilen. Diese Partitionen können dabei auch wieder als Volumes bezeichnet werden, da auch hier wieder einzelne Sektoren adressiert werden können. Ein Dateisystem kann in einem existierendem Volume erstellt werden.

Inzwischen gibt es eine große Anzahl an verschiedenen Dateisystemen, die jeweils verschiedene Anforderungen erfüllen. Die größten Marktanteile bei Desktopbetriebssystemen haben Windows-Systeme. Diese nutzen vor allem das proprietäre NTFS. Im mobilen Bereich sind Android-Geräte am weitesten verbreitet. Während bei diesen zunächst YAFFS als Dateisystem genutzt wurde, wird seit Android-Version 2.3 vorrangig ext4 verwendet. Einige Hersteller verwenden jedoch auch eigene Dateisysteme, so nutzt Samsung teilweise RFS, ein an FAT angelegtes Dateisystem, welches um ein Journal ergänzt wurde. Durch ein Journal können im Falle eines Absturzes die zuletzt durchgeführten Aktionen erkannt werden und mit dieser Information die Integrität des Dateisystems wiederhergestellt werden. Apple-iOS-Geräte wie das iPhone oder iPad implementieren HFSX als Dateisystem, welches nahe verwandt mit HFS+ ist, das unter OS X eingesetzt wird. Windows Phone nutzt FAT oder exFAT. Linux-Systeme verwenden verschiedene Dateisysteme. Gängige Dateisysteme sind hier: btrfs, ext2, ext3, ext4, jfs, ReiserFS, Reiser4, XFS und ZFS. Für CDs, DVDs und Blu-Ray-Discs werden Versionen von UDF verwendet. Davor war vor allem ISO 9660

im Einsatz. USB-Sticks und kleinere Speichermedien nutzen oft noch FAT, ein früheres Dateisystem von Microsoft mit breiter Unterstützung. Andere Geräte wie Smart-TVs, Drucker und NAS-Systeme arbeiten teilweise mit hier genannten Dateisystemen oder eigenen Implementierungen. Daneben kann dasselbe Dateisystem in verschiedenen Treibern unterschiedlich implementiert werden und so von anderen Versionen des gleichen Dateisystems abweichen.

Alle Dateisysteme nutzen zur Speicherung Datenstrukturen, welche die Daten logisch verknüpfen und schnell auffindbar machen. Brian Carrier teilt in seinem Buch „File System Forensic Analysis“ [6] diese Strukturen von Dateisystemen in fünf Datenkategorien ein:

Dateisystem
: Strukturen mit Informationen über das komplette Dateisystem und dessen speziellen Aufbau.

Inhalt
: Strukturen für den eigentlichen Inhalt von Dateien in Dateisystemen.

Metadaten
: Strukturen für Metadaten (z. B. Größe, Zeiten und Rechte) von Dateien.

Dateinamen
: Strukturen für menschenlesbare Bezeichnung von Dateien.

Anwendung
: Weitere Strukturen, die oft aus Performanzgründen direkt auf Dateisystemebene implementiert sind. Ein Beispiel hierfür ist ein Dateisystemjournal.

Diese Strukturen stehen untereinander in Verbindung. So verweist die Dateisystemstruktur oft auf die Speicherorte der anderen Struktur und die Metadatastruktur auf die jeweiligen Inhalte der Dateien.

Daten und Strukturen innerhalb von Dateisystemen gehören entweder zu essentiellen oder zu nicht-essentiellen Daten. Essentielle Daten werden für das Nutzen des Dateisystems unbedingt benötigt, während nicht-essentielle Daten auch veraltet oder nicht vorhanden sein können [6]. Strukturen können dabei sowohl essentielle als auch nicht-essentielle Daten enthalten. Der im Bootsektor von NTFS vermerkte Dateisystemname ist so z. B. nicht-essentiell und kann beliebig belegt werden, während die Anzahl an Sektoren pro Cluster essentiell ist und zum Verständnis des Dateisystems benötigt wird.

Die folgenden Abschnitte stellen vier der genannten Dateisysteme genauer vor.

2.1 FAT

FAT (File Allocation Table) ist eine Gruppe von Dateisystemen, dessen erste Version bereits Ende der 70er Jahre von Microsoft entwickelt wurde. FAT gliedert sich dabei in die Dateisysteme FAT12, FAT16 und FAT32. Diese haben grundsätzlich denselben Aufbau, unterscheiden sich aber in einigen Strukturen und demzufolge auch in den Größenlimits, welche z. B. für Dateigrößen oder die Dateisystemgröße bestehen. FAT16 hat z. B. eine maximale Größe von 4 GiB, FAT32 von 8 TiB. Neben den Limitierungen für Dateisystem- und Dateigrößen gibt es noch weitere Einschränkungen bei FAT-Dateisystemen, so können Dateinamen in der Basisversion nur 8 Zeichen für den Dateinamen und 3 Zeichen für die Dateiendung

haben. Diese Einschränkung wird durch die Erweiterung VFAT aufgehoben. Auch die gespeicherten Zeitstempel sind im Gegensatz zu moderneren Dateisystemen gröber aufgelöst. Während die Erstellungszeit auf 10 Millisekunden genau ist, ist die Modifizierungszeit auf 2 Sekunden bestimmt und die Zugriffszeit nur tagesgenau [14].

FAT ist das Standarddateisystem unter DOS sowie Windows 95 und 98. Durch die Limitierung auf 8 TiB Dateisystemgröße und 4 GiB große Dateien wird inzwischen von Microsoft vor allem NTFS eingesetzt. FAT wird jedoch weiterhin für kleinere Speicher wie USB-Sticks genutzt. Da heutzutage auch Mac OS X sowie Linux-Systeme FAT unterstützen, ist hier in nächster Zeit noch mit einer weiten Verbreitung zu rechnen.

FAT besitzt nur eine kleine Anzahl an Dateisystemstrukturen. Tabelle 2.1 veranschaulicht, wie diese in einem FAT-Dateisystem liegen. Zu Beginn der Partition steht der Bootsektor. Bei FAT32 kann es außerdem einen Backup-Bootsektor geben, welcher hier nicht verzeichnet ist. Bei FAT32 folgt auf den Bootsektor der Dateisystem-Informationsbereich (FSINFO), bei FAT12 oder FAT16 existiert dieser nicht. Bis hier werden teilweise alle bisher genannten Strukturen als reservierte Bereiche zusammengefasst. Nach den reservierten Bereichen folgen die File Allocation Tables (FATs). Hiervon sind in der Regel zwei vorhanden, die direkt hintereinander liegen. Nach den FATs kommt das Wurzelverzeichnis des Dateisystems und hierauf der Datenbereich für weitere Verzeichnisse und Dateien. Eine Besonderheit von FAT ist hier, dass FAT erst ab dem Wurzelverzeichnis damit beginnt das Dateisystem in gleich große Blöcke einzuteilen.

Tabelle 2.1: FAT-Struktur (nicht maßstabsgetreu)

Boot-sektor	FSINFO	FAT-Bereich	Wurzel-verzeichnis	Datenbereich

Im Folgenden werden die einzelnen Bestandteile aus Tabelle 2.1 kurz näher erklärt:

Bootsektor
: Der Bootsektor eines FAT-Dateisystems liegt immer im ersten Sektor der Partition. Die ersten 36 Bytes des Bootsektors haben für FAT12, FAT16 und FAT32 die gleiche Bedeutung. Die darauffolgenden Bytes sind identisch für FAT12 und FAT16 nicht aber für FAT32. Der Bootsektor enthält Informationen, die das Dateisystem allgemein betreffen. Beispiele für hier vorhandene Informationen sind: Bytes pro Sektor, Anzahl an FATs oder die maximale Anzahl an Dateien für das Wurzelverzeichnis.

FSINFO
: Der Dateisystem-Informationsbereich (FSINFO) existiert nur bei FAT32. Er enthält Informationen über freie Cluster und die Position des nächsten freien Clusters.

FAT-Bereich
: Der FAT-Bereich besteht aus einer oder mehreren FATs, in der Regel jedoch genau aus 2 FATs. Eine FAT besteht aus einer Liste an Einträgen, die je nach Dateisystem aus 12 Bit (FAT12), 16 Bit (FAT16) oder 32 Bit (FAT32) bestehen. Diese Einträge bestimmen, ob ein Block im Dateisystem belegt ist und welcher der jeweils darauf folgende Block im Dateisystem ist. Auch beschädigte Blöcke werden hier vermerkt.

Wurzelverzeichnis

Das Wurzelverzeichnis unterscheidet sich strukturell nicht von anderen Verzeichnissen bei FAT. Wie auch bei anderen Verzeichnissen sind hier Dateinamen und Metadaten zu Dateien wie die Erstellungszeit oder Dateigröße gespeichert.

Datenbereich

Der Dateibereich enthält Verzeichnisse und Dateiinhalte, die in den restlichen verfügbaren Blöcken aufgeteilt werden.

2.2 NTFS

NTFS (New Technologie File System) ist ein von Microsoft entwickeltes Dateisystem und Nachfolger des FAT-Dateisystems. NTFS löst viele der Limitierungen von FAT, so werden lange Dateinamen direkt unterstützt und Dateigrößen über 4 GiB sind möglich. Außerdem verfügt NTFS über die Möglichkeit zum Journaling, welches die Ausfallsicherheit des Dateisystems erhöht. NTFS gibt es in verschiedenen Versionen, die jeweils mit neuen Betriebssystemversionen erschienen.

NTFS wird als Standarddateisystem unter Windows NT und allen danach erschienenen Windows-Betriebssystemen verwendet [6]. Durch den hohen Marktanteil von über 90% dieser Windows-Versionen ist demnach NTFS das momentan meistgenutzte Dateisystem für Desktop-Computer [21]. OS X unterstützt NTFS nativ nur lesend, es gibt jedoch auch Treiberpakete, welche schreibenden Zugriff ermöglichen. Unter Linux ist mittels des NTFS-3G-Treibers sowohl schreibender als auch lesender Zugriff auf NTFS möglich. Der NTFS-3G-Treiber wurde durch Reverse Engineering von NTFS erstellt, was in Abschnitt 4.2 genauer beschrieben wird.

NTFS hat eine sehr viel einheitlichere Struktur als FAT. Das komplette Dateisystem ist hier in gleichgroße Blöcke eingeteilt und die Dateisystemstrukturen werden wie Dateien behandelt. Alle Einträge zu Dateien werden in einer Master File Table (MFT) gespeichert. Diese bilden zusammen mit dem Bootsektor und einer reduzierten Kopie der MFT die Grundstruktur von NTFS, welche auch in Tabelle 2.2 zu sehen ist.

Tabelle 2.2: NTFS-Struktur (nicht maßstabsgetreu)

Boot-sektor	MFT	Datenbereich	MFT-Kopie	Datenbereich

MFT-Einträge besitzen einen Header sowie eine beliebige Anzahl an Attributen. Diese Attribute bestehen wiederum aus einem Attribut Header sowie einem Attribut Content. Der Attribut Content kann aber auch außerhalb (non-resident) des MFT-Eintrags liegen. In diesem Fall enthält das Attribut dann einen Verweis auf den Attribute Content.

Die ersten 16 Einträge in der MFT sind für Dateisystemstrukturen reserviert, die jeweils bestimmte Funktionen erfüllen. All diese Strukturen beginnen mit „$“ und sind für den Nutzer des Betriebssystems nicht sichtbar. Bisher werden nur 12 dieser Einträge genutzt, daneben gibt es jedoch auch noch Erweiterungen, welche weitere Dateisystemstrukturen enthalten können.

Master File Table (MFT-Eintrag: *0*, MFT-Dateiname: *$MFT*)

Die Master File Table (MFT) ist eine Liste an Einträgen für jede Datei und jeden Ordner des Dateisystems.

Master File Table Mirror (MFT-Eintrag: *1*, MFT-Dateiname: *$MFTMirr*)

Der Master File Table Mirror enthält lediglich vier Einträge. Diese entsprechen den ersten vier Einträgen der MFT.

Log File (MFT-Eintrag: *2*, MFT-Dateiname: *$LogFile*)

Die sogenannte Log File ist das Journal des Dateisystems. Dieses enthält Informationen, um Dateisystem-Metadaten wiederherzustellen.

Volume (MFT-Eintrag: *3*, MFT-Dateiname: *$Volume*)

Die Volume-Datei enthält Volumename sowie NTFS-Version und Dateisystemstatus.

Attribut-Definitionen (MFT-Eintrag: *4*, MFT-Dateiname: *$AttrDef*)

Die Attribut-Definitionen listen alle möglichen Attribute für MFT-Einträge auf.

Wurzelverzeichnis (MFT-Eintrag: 5, MFT-Dateiname: .)

Das Wurzelverzeichnis ist als fixer MFT-Eintrag realisiert und gibt so einen Einstieg in den Verzeichnisbaum.

Cluster Bitmap (MFT-Eintrag: *6*, MFT-Dateiname: *$Bitmap*)

Die Bitmap gibt an, welche Blöcke im Dateisystem belegt sind und welche nicht.

Bootsektor (MFT-Eintrag: *7*, MFT-Dateiname: *$Boot*)

Auch der Bootsektor ist als MFT-Eintrag gehalten und so als Datei erreichbar. Der Bootsektor liegt jedoch immer in Block 0 und enthält Verweise zur MFT sowie MFT Mirror, um einen Einstieg in das Dateisystem zu ermöglichen. Ein Backup-Bootsektor liegt bei Windows NT 3.5 in der Mitte des Volumes, bei Windows NT4.0 und Windows 2000 am Ende des Volumes [30].

Datei für fehlerhafte Cluster (MFT-Eintrag: *8*, MFT-Dateiname: *$BadClus*)

Die Datei für fehlerhafte Cluster listet beschädigte Cluster, wenn diese vom Betriebssystem entdeckt wurden.

Sicherheitsdatei (MFT-Eintrag: *9*, MFT-Dateiname: *$Secure*)

Die Sicherheitsdatei speichert Attribute, die den Zugriff auf Dateien und Verzeichnisse regeln. Diese werden als Dateisystemstruktur gespeichert, da sie in vielen Dateien gleich sind und so Speicherplatz gespart werden kann, wenn diese zentral gespeichert werden. In früheren Versionen von NTFS wurden die Sicherheitsattribute noch einzeln bei den Dateien gespeichert.

Großbuchstabentabelle (MFT-Eintrag: *10*, MFT-Dateiname: *$UpCase*)

In der Großbuchstabentabelle wird die Zuordnung von Großbuchstaben zu Kleinbuchstaben gespeichert, um diese für Dateinamen gleichwertig behandelt zu können.

Erweiterungen (MFT-Eintrag: *11*, MFT-Dateiname: *$Extend*)

In der MFT sind Erweiterungen für weitere Dateisystemstrukturen möglich. Der elfte MFT-Eintrag ist hierfür als Verzeichnis realisiert, während alle vorherigen Einträge Dateien darstellen. Typische Erweiterungen sind *$Quota* (MFT-Eintrag: *24*), um Speicherplatzkontingente festzulegen sowie *$ObjId* (MFT-Eintrag: *25*) und *$Reparse* (MFT-Eintrag: *26*), welche Informationen zu Verweisen im Dateisystem enthalten.

2.3 HFS+

HFS+ wurde 1998 als Nachfolger für HFS (Hierarchical File System) von Apple Inc. vorgestellt. Auch hier wurden, ebenso wie bei NTFS, Limits des Vorgängers für Dateigröße und Dateinamen erhöht. Als Erweiterung zu HFS+ gibt es HFSX, welches im Gegensatz zu HFS+ Groß- und Kleinschreibung beachtet. Neue Versionen des Dateisystems sind jeweils mit neuen Versionen des Mac-OS-Betriebssystems erschienen und führten einige Änderungen (z. B. Journaling, Kompression) ein.

Apples Betriebssysteme OS X, für Desktop-Computer, und iOS, welches in den mobilen Geräten von Apple (iPhone, iPad) eingesetzt wird, haben gemeinsam einen Marktanteil von etwa 11% [17]. Die Desktopsysteme von Apple nutzen HFS+ als Dateisystem, die mobilen Betriebssysteme HFSX. Die Limits des Dateisystems mit 8 EiB (Exbibyte = 2^{60} Bytes) werden noch einige Jahre ausreichen. Ein Limit für das Dateisystem könnte jedoch der Zeitstempel sein, welcher 2040 ausläuft.

HFS+ beginnt mit dem sogenannten Volume Header. Dieser ist 512 Byte groß und liegt genau 1024 Byte nach dem Start des Volume. Ein Backup dieses Volume Headers liegt 1024 Byte vor dem Ende des Volumes. Daneben gibt es fünf als „special files“ bezeichnete Strukturen (Catalog File, Extents Overflow File, Allocation File, Attibutes File, Startup File). Zusätzlich gibt es ein optionales Journal, welches in der ersten Version von HFS+ noch nicht vorhanden war. Tabelle 2.3 zeigt den Aufbau eines untersuchten HFS+. Die freien Bereiche enthalten Dateien, eine Startup File ist hier nicht vorhanden.

Tabelle 2.3: HFS+-Struktur (nicht maßstabsgetreu)

Volume Header	Allocation File	Journal	Extents Overflow File	Attributes File		Catalog File		Alternate Volume Header

Volume Header und Alternate Volume Header

Der Volume Header enthält Informationen über das Dateisystem wie die Version, das Erstellungsdatum oder die Blockgröße. Außerdem sind hier Verweise auf die „special files“ sowie das Journal zu finden.

Catalog File

Die Catalog File enthält Namen sowie Metadaten (z. B. die letzte Bearbeitung oder Zugriffberechtigungen) zu Dateien und Ordnern. Im Gegensatz zu anderen Dateisystemen werden hier auch für jede Datei und jedes Verzeichnis Informationen für Finder, den Dateimanager unter OS X, abgelegt.

Extents Overflow File

Jeder Dateieintrag in der Catalog File kann auf bis zu acht Sequenzen von Blöcken verweisen, in denen die Daten dieser Datei liegen. Werden mehr Sequenzen benötigt werden diese in der Extents Overflow File angelegt und in der Catalog File referenziert. Außerdem werden Verweise auf beschädigte Sektoren hier verwaltet.

Attributes File

Die Attributes File ist eine optionale Struktur, welche zusätzliche Attribute zu Dateien speichern kann.

Allocation File

Die Allocation File funktioniert analog zur Cluster Bitmap in NTFS und gibt an, welche Blöcke des Dateisystems belegt sind.

Startup File

Die Startup File kann genutzt werden, um einfach Zugriff auf Informationen zum Booten eines Betriebssystems zu bekommen.

Journal

HFS+ kann ein Journal besitzen, welches die Wiederherstellung des Dateisystems erleichtert. Hierzu gibt es im Volume Header einen Verweis auf den Journal Info Block. Dieser verweist auf den Journal Header, auf den der eigentliche Inhalt des Journals im Journal Buffer folgt.

2.4 ext

ext (extended filesystem) ist eine Familie von Dateisystemen, die für Linux entwickelt wurden. Bis heute gibt es die Dateisysteme ext, ext2, ext3 und ext4. Das erste Dateisystem der Reihe, ext, wurde im April 1992 von Rémy Card *et al.* implementiert [4].

Die größte Verwendung findet ext4 in Android-Geräten. Diese verwenden seit Android 2.3 ext4 als Dateisystem, welches das vorher verwendete YAFFS ablöst [11]. Android hat knapp 50% Marktanteil am gesamten Betriebssystemmarkt [17], und auch wenn nicht alle Gerätehersteller ext4 nutzen, wird hierdurch eine hohe Verbreitung von ext4 erreicht.

ext hat auf oberster Ebene eine sehr einfache Struktur. Das Dateisystem beginnt mit einem Superblock und ist dann in weitgehend gleichgroße Blockgruppen eingeteilt. Hierbei kann die letzte Blockgruppe kleiner sein als die vorherigen. Tabelle 2.4 veranschaulicht diese Struktur. Die Struktur einer einzelnen Blockgruppe ist in Tabelle 2.5 dargestellt.

Tabelle 2.4: EXT-Struktur (nicht maßstabsgetreu)

Super-block	Blockgruppe 0	Blockgruppe 1	...	Blockgruppe n-1

Boot Code

Die ersten 1024 Bytes des Dateisystems können Boot Code enthalten, alternativ kann der MBR einen Bootloader enthalten, welcher dann den Boot-Prozess weiterleitet.

Superblock

Der Superblock entspricht in etwa dem Bootsektor bei FAT und NTFS oder dem Volume Header bei HFS+. Hier liegen Informationen zu der Anzahl an Blöcken im Dateisystem und pro Blockgruppe. Der erste Superblock liegt immer direkt hinter dem Boot Code, kann aber auch zur ersten Blockgruppe gehören.

Blockgruppe

Bei ext-Dateisystemen wird der zur Verfügung stehende Speicher nach dem Boot Code und dem Superblock in gleich große Gruppen von Blocks eingeteilt. Die letzte dieser Blockgruppen kann kleiner sein als die übrigen Blockgruppen, um möglichst viel Speicherplatz zu nutzen.

Tabelle 2.5: Block-Group-Struktur (nicht maßstabsgetreu)

Backup Superblock	Group Descriptor Table	Block Bitmap	Inode Bitmap	Inode Table	File Content

Backup Superblock

Die einzelnen Blockgruppen können eine Kopie des Superblocks enthalten.

Group Descriptor Table

Die Group Descriptor Table beschreibt den Aufbau aller Blockgruppen des Dateisystems. Die Group Descriptor Table kann dabei in jedem Block auftauchen, ist aber oft nur in jedem zweiten Block zu finden.

Block Bitmap

Die Block Bitmap arbeitet wie die Cluster Bitmap bei NTFS oder die Allocation File bei HFS+, jedoch begrenzt für die Blöcke der jeweiligen Blockgruppe.

Inode Bitmap

Die Inode Bitmap arbeitet wie die Block Bitmap, jedoch für die darauffolgende Inode Table.

Inode Table

Die Inode Table enthält einzelne Inodes. Inodes enthalten Dateimetadaten, aber keine Dateinamen.

File Content

Der File-Content-Bereich enthält den eigentlichen Dateiinhalt.

2.5 Vergleich

Die in den vorhergehenden Abschnitten beschriebenen Dateisysteme zeigen einige Konzepte, die in vielen Dateisystemen eingesetzt werden. Diese können auch in unbekannten Dateisystemen gesucht werden und so das Reverse Engineering von diesen erleichtern. Alle modernen Dateisysteme nutzen z. B. das Konzept der Zusammenfassung von mehreren konsekutiven Sektoren des Speichermediums in einen Block.

Dateisysteme sind nur durch wenige Vorgaben gebunden. Diese Vorgaben entstehen durch die Hardware, welche wie in Abschnitt 2 beschrieben bei Festplatten in der Regel bei 512 Byte oder 4 KiB Blöcke zur Verfügung stellt. Hierdurch ist es sinnvoll auch die Dateisysteme in Blöcke mit einem Vielfachen dieser Größe einzuteilen. Gespeicherte Daten werden dann ebenfalls in Einheiten dieser Größe geteilt und gespeichert. Hierdurch kann man in der Regel den darin gespeicherten Inhalt einfach identifizieren. Manche Dateisysteme nutzen jedoch Kompression oder Verschlüsselung, wodurch die auf dem Speichermedium vorhandenen Blöcke nicht direkt dem Dateiinhalt entsprechen. Auch möglich ist das Speichern

von kurzen Dateiinhalten direkt in einer anderen Struktur, wie es in NTFS unterstützt wird, um viel ungenutzten Speicher (so genannten Slack Space) zu vermeiden.

Alle betrachteten Dateisysteme nutzen einen Dateisystem-Header in den ersten Blöcken des Dateisystems, welche grundlegende Informationen über das Dateisystem enthält. Dies ist sinnvoll, da von außen der Start und die Größe der Partition nicht aber ein Einstiegspunkt bekannt ist. Der Dateisystem-Header beschreibt dann den weiteren Aufbau des Dateisystems, vor allem die Blockgröße und Verweise auf andere essentielle Strukturen. Von dem Dateisystem-Header gibt es in den betrachteten Dateisystemen mindestens eine weitere Kopie an einer anderen Position im Dateisystem, die bei Beschädigung des ersten Headers zur Wiederherstellung genutzt werden kann.

Eine weitere Struktur, welche in allen beschriebenen Dateisystemen existiert, ist eine Allokationsdatei. Hierbei weicht FAT von den anderen Dateisystemen ab, welche eine einfach Bitmap nutzen, statt jeweils mehrere Bits für einen Eintrag. Die Allokationsdatei ist nötig, um schnell freie Blöcke zu erkennen und belegen zu können. Einfache Strategien zum Belegen von Blöcken sind *first available* oder *next available*, wobei jeweils von Beginn oder letzter Belegung an freier Platz für Daten gesucht wird [6]. Hierdurch entstehen in Allokationsdateien Ketten von belegten Blöcken.

Im Speichern von Dateinamen und Dateimetadaten unterscheiden sich die verschiedenen Dateisysteme. FAT, HSF+ und NTFS speichern Dateinamen und Metadaten jeweils in einer Datenstruktur, während ext Namen und Metadaten getrennt speichert. Auch die gespeicherten Metadaten unterscheiden sich je nach Dateisystem. Für gespeicherte Zeiten wird der Begriff „MACtime“ genutzt, welcher die Modifikations-, Zugriffs-(A für Access) und Change-(bei Unix-Systemen) bzw. Creation-(bei Windows-Systemen) Zeitstempel zusammenfasst.

NTFS, HFS+ und ext4 können ein Journal nutzen, um im Falle eines Absturzes schnell die Integrität des Dateisystems wiederherstellen zu können. Diese Journals enthalten Informationen zu Transaktionen im Dateisystem, welche so Daten aus allen anderen Strukturen enthalten können.

Carrier versucht diese Gemeinsamkeiten durch die erstellten Datenkategorien zu strukturieren. Die Strukturen der verschiedenen Dateisysteme lassen sich wie Tabelle 2.6 dargestellt einordnen.

Neben diesen Gemeinsamkeiten besitzen die einzelnen Dateisysteme auch Eigenheiten und spezielle Strukturen. Beispielsweise die $UpCase-Datei existiert in dieser Art und Weise nur in NTFS, die Extents-Overflow-Datei nur in HFS+-Systemen und das Konzept der Inodes nur bei ext.

Tabelle 2.6: Datenstrukturen eingeteilt in die Datenkategorien (angelehnt an [6])

Daten-kategorie	FAT	NTFS	HFS+	ext
Dateisystem	Bootsektor FSINFO	$Boot $Volume $AttrDef	Volume Header Alternate Volume Header Startup File	Superblock Group Descriptor
Inhalt	Datenbereich FAT	Datenbereich $Bitmap $BadClus	Datenbereich Allocation File Extents Overflow file	File Content Block Bitmap
Metadata	Verzeichnisse FAT	$MFT $MFTMirr $Secure	Catalog File Attributes File	Inode Table Inode Bitmap Extended Attibutes
Dateiname	Verzeichnisse	$MFT $MFTMirr $UpCase	Catalog File	Directory Entries
Anwendung	-	$LogFile	Journal	Journal

3 Dateisystemanalyse

Dateisystem-Reverse-Engineering bezeichnet das Ableiten eines Dateisystemmodells aus einem existierenden Dateisystem. Dateisystemanalyse dagegen bezeichnet die Übersetzung von Bytes und Sektoren einer Partition in Verzeichnisse und Dateien [5]. Dateisystem-Reverse-Engineering bietet somit die Grundlage zur Dateisystemanalyse. Einige Methoden der Dateisystemanalyse können aber auch zum Dateisystem-Reverse-Engineering eingesetzt werden. Insgesamt kann das Reverse Engineering als Verallgemeinerung der Dateisystemanalyse gesehen werden. So wäre ein Teil der Dateisystemanalyse das Ermitteln der Blockgröße für ein konkretes Dateisystem und Teil des Reverse Engineering das Ermitteln der Position im Dateisystem-Header, an welcher diese Blockgröße festgelegt ist.

Bei der Dateisystemanalyse gibt es diverse Tools, die diese unterstützen können. Beispiele für verwendete Programme sind EnCase Forensic von Guidance Software[1], das Forensic Toolkit von AccessData[2] sowie The Sleuth Kit[3] [6]. Diese Programme listen Dateien und Ordner von den jeweils zu untersuchenden Dateisystemen auf. Daneben bieten diese Tools die Möglichkeit viele weitere Analysen des Dateisystems durchzuführen. Um diese Funktionen anbieten zu können, wird jedoch ein Modell der zu untersuchenden Dateisysteme benötigt.

Eine Methode, die unabhängig von der Kenntnis des Dateisystems ist, ist das File Carving. Hierbei werden anhand von Signaturen Dateien identifiziert und wiederhergestellt. Signaturen sind feste Bytesequenzen, welche an bestimmten Positionen in Dateien auftauchen. Bei PDF-Dateien sind die ersten vier Bytes immer „%PDF“. Durch File Carving können so auch bei fehlenden Metadatenstrukturen oder gelöschten, aber nicht überschriebenen, Dateien diese gefunden und extrahiert werden.

Neben der Möglichkeit ganze Dateien wiederherzustellen, gibt es auch Programme die bestimmte Bytefolgen suchen. `strings` ist ein Kommandozeilenprogramm, welches Zeichenketten bestimmter Länge sucht. Daneben gibt es Programme, die z. B. IPv4-Adressen, URLs oder Datumsangaben in unstrukturierten Daten finden [7,8].

Als Hilfsmittel zur manuellen Dateisystemanalyse werden häufig Hex-Editoren verwendet. Diese bieten teilweise Zusatzfunktionen, die bei der Analyse von Dateisystemen nützlich sind. Der Editor iBored[4] bieten so z. B. die Möglichkeit Daten in Blöcken bestimmter Größe anzuzeigen. Die Editoren Synalyze It![5] oder Hexinator[6] können bekannte Strukturen hervorheben.

1 http://www.encase.com/
2 http://accessdata.com/
3 http://www.sleuthkit.org/
4 http://apps.tempel.org/iBored/
5 https://www.synalysis.net/
6 https://hexinator.com/

Schuster nutzt in seiner Präsentation Entropie und Autokorrelation, um Strukturen in Dateisystemen zu erkennen [32]. Der Vorteil dieser mathematischen Ansätze ist, dass sie auch bei völlig unbekannten Strukturen funktioniert. Beide Ansätze werden in den folgenden Abschnitten genauer beschrieben.

3.1 Entropie

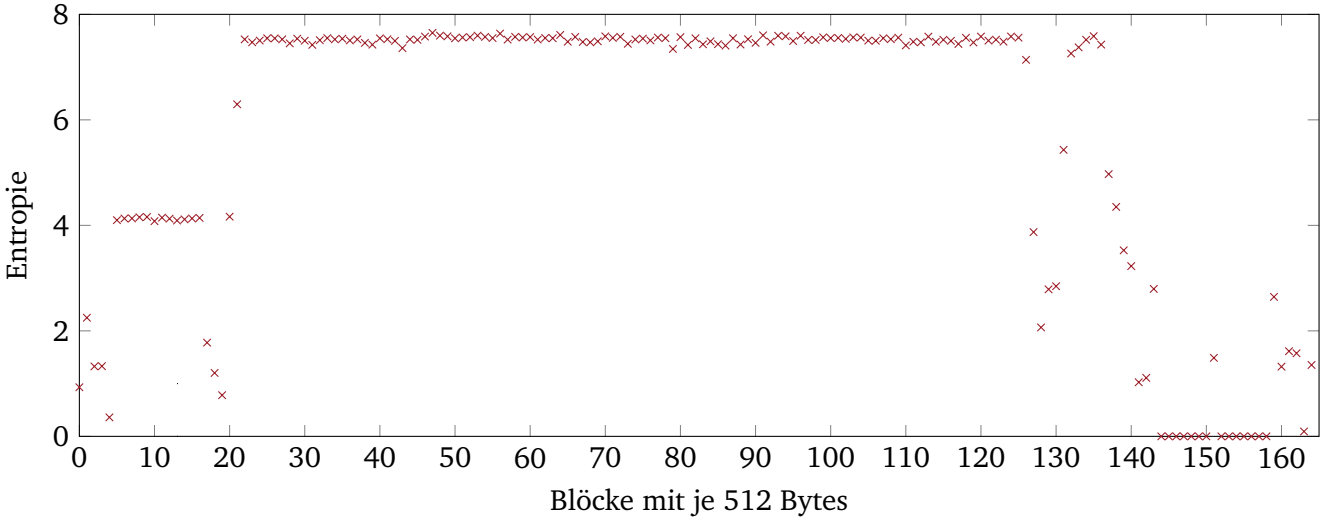
Entropie ist ein Maß, welches die Informationsdichte von Daten beschreibt [33]. Berechnen lässt sich die Entropie mit Binärdaten mittels der Formel:

$$H(X) = -\sum_{i=1}^{256} p(x_i) \log_2 p(x_i) \tag{3.1}$$

Für eine beliebige Anzahl an Bytes ergibt diese Formel einen Wert zwischen 0 und 8. Dieser gibt die Informationsdichte in Bits pro Byte an. Eine Reihe von leeren Bytes ergibt hierbei einen Wert von 0; stark komprimierte Daten kommen nahe an 8 Bits pro Byte an Informationsdichte heran. Dies ist häufig bei Bildern, Musik oder ZIP-Archiven der Fall. Text in Goethes Faust [12] hat eine Entropie von etwa 4,76.

Folgendes Beispiel zeigt die Entropie von 512-Byte-Blöcken in einer DOC-Datei. Die Datei besteht dabei aus einem Beispieltext sowie einem Bild. In Abbildung 3.1 erkennt man eine Entropie von etwa 4,2 für den enthaltenen Text in den Blöcken 5 - 16. Die höhere Entropie von etwa 7,6 in den Blöcken 20 - 127 entspricht den Daten des Bildes. Die anderen Bereiche sind Strukturen, welche z. B. das Layout des Dokuments bestimmen.

Abbildung 3.1: Entropie einer DOC-Datei mit Text und Bild



Ähnlich wie in diesem Bild haben unterschiedliche Strukturen in Dateisystemen auch oft unterschiedliche Entropie. Hierdurch können Strukturen unterteilt und eingeordnet werden. Daneben kann Entropie genutzt werden, um komprimierte Daten zu erkennen.

3.2 Chi-Quadrat-Verteilung

Mit der Chi-Quadrat-Verteilung lässt sich bestimmen, wie gleich oder ungleich Werte verteilt sind. Berechnen lässt sich die Verteilung über die Formel:

$$\chi^2 = \sum_{b=0}^{255} \frac{(C_b - C_t)^2}{C_t} \tag{3.2}$$

Hierbei ist C_b die Häufigkeit des Bytes in den Eingabedaten und C_t die Größe der Eingabedaten. Ist die Verteilung zu ungleich, führt die Chi-Quadrat-Berechnung zu einem zu hohen Wert; ist die Verteilung zu gleichmäßig, führt die Berechnung zu einem zu niedrigen Wert. Bei der Berechnung über Bytes gibt es hierbei 256 Kategorien und dementsprechend 255 Freiheitsgrade. Je näher der Wert der Chi-Quadrat-Verteilung an 255 liegt desto zufälliger sind die Bytes verteilt. Werte nahe 255 werden von guten (Pseudo-)Zufallszahlengeneratoren oder Verschlüsselungsalgorithmen erzeugt. Dazu lässt sich jeweils eine Prozentzahl ausrechnen, welche angibt, wie viele echt zufällige Ergebnisse näher an 255 liegen würden. Nimmt man an, dass echt zufällige Zahlen in nicht weniger als 5% der Fälle gleichverteilter sein sollen als die Eingabezahl, ergibt sich hier eine obere Grenze von 293,2478. Nimmt man weiter an, dass die Eingabezahl nicht gleichverteilter als 95% der echt zufälligen Zahlen sein soll, ergibt dies eine untere Grenze von 219,0252. Werte zwischen 219,0252 und 293,2478 zählen demnach als zufällig verteilt.

Tabelle 3.1 zeigt einige Werte für verschiedene Arten von Dateien. Der Gegensatz zur Entropie wird bei komprimierten Daten wie Bildern oder Archiven deutlich. Diese haben in der Regel eine hohe Entropie, jedoch eine nicht zufällige Chi-Quadrat-Verteilung. In dem Beispiel entspricht nur die verschlüsselte Datei dem oben genannten Wertebereich für zufällige Daten. In der Dateisystemanalyse kann die Chi-Quadrat-Berechnung genutzt werden, um für Datenblöcke herauszufinden, ob diese zufällig belegt sind oder nicht. Hierdurch können zum Beispiel verschlüsselte Strukturen oder Daten erkannt werden oder geprüft werden, ob das vollständige Dateisystem verschlüsselt ist.

Tabelle 3.1: Beispielhafte Chi-Quadrat-Werte

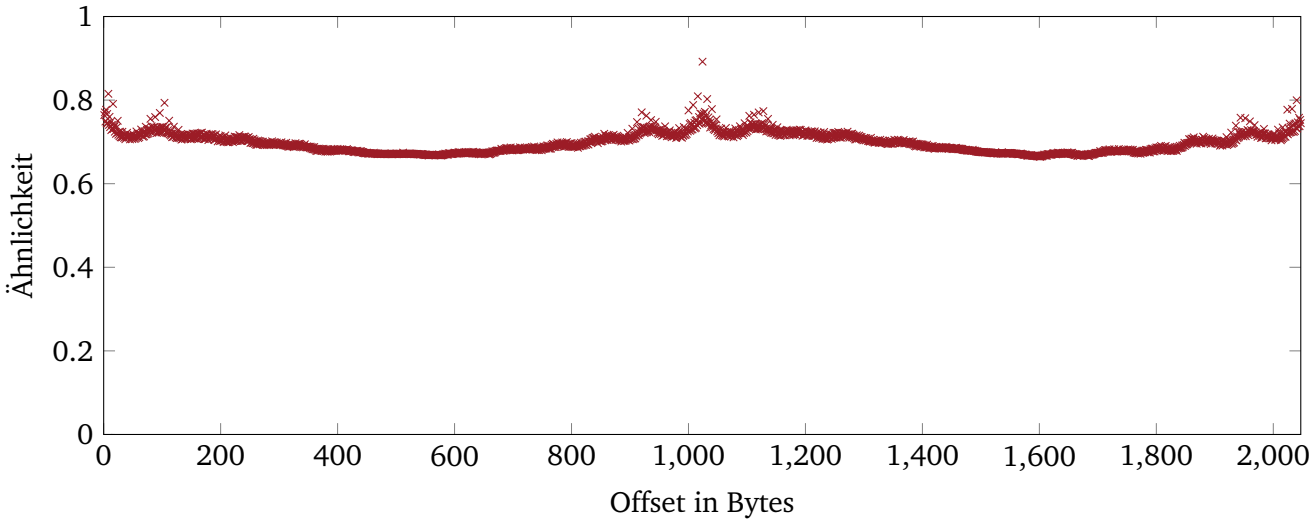
Daten	χ^2-Wert
Goethes Faust	3194012,8
JPG-Datei	34287,9
ZIP-Archiv	423,0
GnuPG verschlüsselte Datei	228,7

3.3 Autokorrelation

Autokorrelation beschreibt in der Signalverarbeitung die Korrelation von einem Signal mit sich selbst an verschiedenen Zeitpunkten. In dieser Arbeit wird, nach Schuster [32], ein daran angelegtes Verfahren als Autokorrelation bezeichnet, bei dem Sequenzen von Bytes mit einer verschobenen Kopie ihrer selbst verglichen werden. Zunächst wird dabei die Kopie der Daten um einen bestimmten Offset verschoben und dann mit den Urspungsdaten verglichen. Die Anzahl der übereinstimmenden Bytes in beiden Datenreihen wird dann durch die Länge der verglichenen Bytes geteilt und so ein Ähnlichkeitswert ermittelt. Kommt man hier auf den Wert 1.0, bedeutet dies, dass alle Bytes exakt übereinstimmen. Durch Verschieben mit unterschiedlichen Abständen entsteht so eine Kurve von Ähnlichkeitswerten.

Genutzt werden kann die Autokorrelation für die automatische Erkennung wiederkehrender Datenstrukturen. Bei Listen von Einträgen fester Größe und ähnlicher Struktur kann die Größe dieser Einträge durch das Maximum der Ähnlichkeitswerte ermittelt werden. Abbildung 3.2 zeigt die Autokorrelationswerte der Dateieinträge in der MFT bei NTFS. Hier ist gut das Maximum bei der Verschiebung um 1024 Bytes zu erkennen, welches der Größe von Einträgen in der MFT entspricht.

Abbildung 3.2: Autokorrelation eines Blocks mit Dateieinträgen bei NTFS



4 Forschungsstand

Automatisierung von Reverse-Engineering-Prozessen wurde, bei Netzwerkverkehr sowie verschiedenen Dateiformaten betrieben, jedoch nicht für Dateisysteme. Das übergeordnete Problem, Modelle aus unstrukturierten Daten zu erstellen, ist mit diesem Prozess nahe verwandt. Mehrere Arbeiten dazu werden im Abschnitt 4.1 beschrieben. In Arbeiten zu Dateisystem-Reverse-Engineering wurde lediglich ein bestimmtes Dateisystem betrachtet. Hierbei wurde meistens manuell vorgegangen. Einige dieser Arbeiten werden im Abschnitt 4.2 vorgestellt.

4.1 Automatisches Reverse Engineering

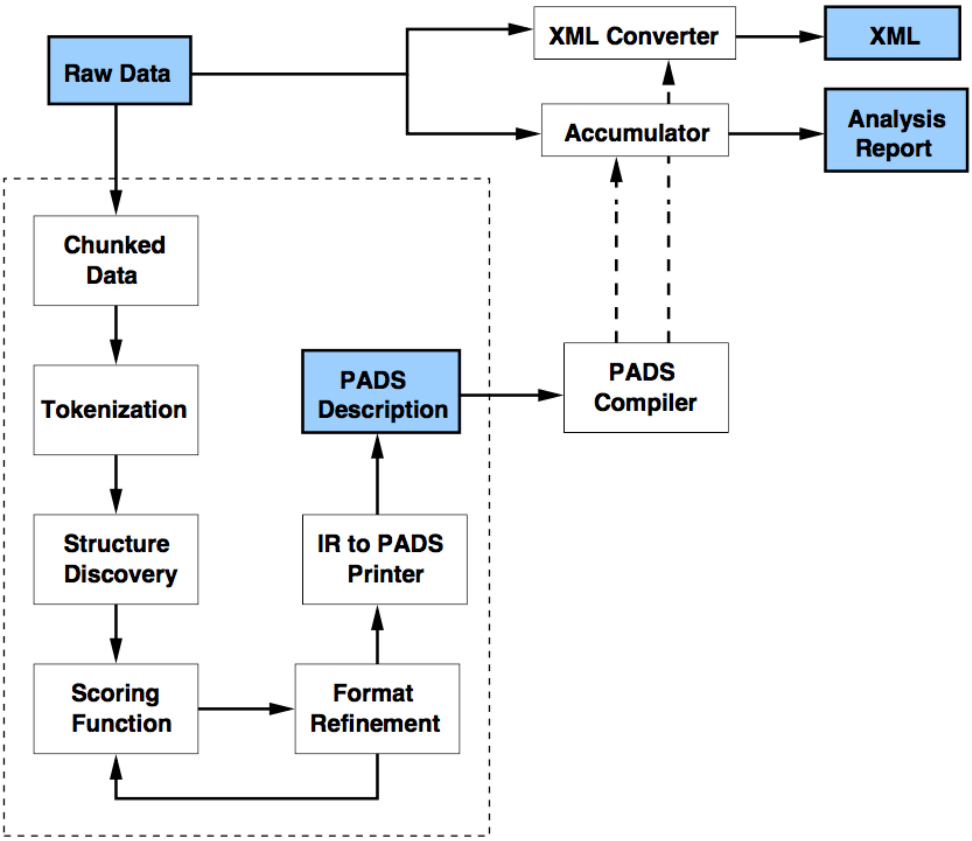
Automatisches Reverse Engineering wurde bisher noch nicht für Dateisysteme durchgeführt. Dennoch gibt es einige Systeme, welche aus wenig- oder unstrukturierten Eingabedaten automatisch Strukturen erkennen.

Ein frühes System zur automatisierten Datenkategorisierung ist Potter's wheel [29]. Hierbei werden strukturierte Eingabedaten in Tokens zerlegt. Token sind eine Folge zusammengehöriger Bytes. Die Tokens werden dann in Gruppen (Zahlen, Wörter mit großem Anfangsbuchstaben, Geldbeträge etc.) eingeordnet. Das System hat zum Ziel einzelne abweichende Datensätze zu erkennen und so leichter bereinigen zu können. Die automatische Bestimmung von Wertebereichen kann auch für Dateisysteme interessant sein, um so z. B. automatisch Datumsfelder oder eingeschränkte Bereiche anderer Felder zu erkennen, die z. B. nur Zweierpotenzen enthalten können. Vor allem durch die Eingabedaten, welche strukturiert vorliegen müssen, sind hier vor allem grundlegende Ideen und Algorithmen zur Bestimmung von Daten interessant.

Ein weitergehendes System wurde 2008 von Fisher *et al.* vorgestellt [15]. Hierbei werden Rohdaten eingegeben und nach mehreren Phasen eine Formatbeschreibung in einer Auszeichnungssprache (PADS) erstellt. Der Prozess dazu ist im gestrichelten Bereich in Abbildung 4.1 dargestellt. Ausgegangen wird hier von Textdaten als Eingabe. Das Chunking ist der erste Schritt des Prozesses. Hierbei werden die Textdaten in einzelne Einheiten, die so genannten Chunks, aufgeteilt. Dieses Chunking kann zeilen- oder dateibasiert geschehen. Anschließend werden die Chunks in Tokens zerteilt. Die Sequenzen von Tokens werden anschließend als Basistyp, Struktur, Array oder Uniontyp eingeordnet. Hierzu wird die Tokensequenzen in Histogrammen betrachtet. Ein Token, der in allen Chunks auftaucht, dabei aber jeweils nur einmal, definiert so z. B. eine Struktur, während ein Token, der in allen Chunks sehr häufig auftaucht, als Array gesehen wird. Anschließend wird anhand einiger fixer Regeln die Struktur bewertet und verfeinert. Zuletzt wird die Struktur in die Auszeichnungssprache PADS umgewandelt.

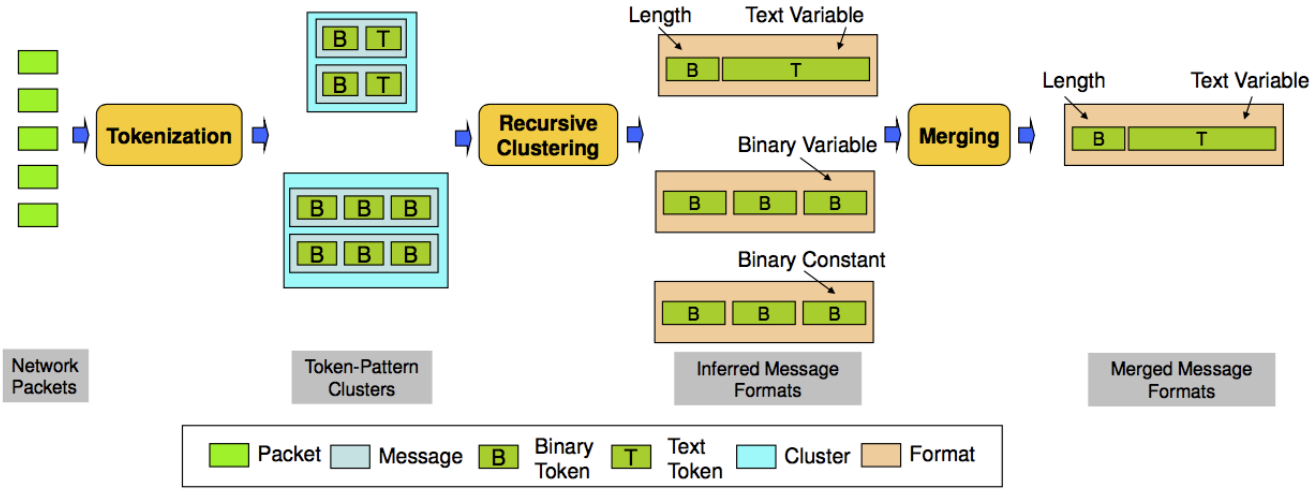
Weidong Cui *et al.* haben außerdem zwei Artikel veröffentlicht, in welchen mittels automatisiertem Reverse Engineering Datenformate ermittelt wurden. Der erste dieser Artikel von 2007 beschreibt das auto-

Abbildung 4.1: Prozess von Fisher *et al.* [15]



matische Reverse Engineering von Netzwerkprotokollen aus Netzwerkverkehr [9]. Dazu wurde das Tool „Discoverer“ entwickelt, welches selbst jedoch nicht veröffentlicht wurde. Grafik 4.2 zeigt eine Übersicht über den Prozess, welcher nach dem Durchlaufen mehrerer Phasen aus den Netzwerkpaketen eine Menge von Nachrichtenformaten erstellt. Dieser Prozess wird im Folgenden etwas genauer beleuchtet.

Abbildung 4.2: Architektur des Discoverer Tools [9]



Die Tokenisierung trennt zunächst die Bytesequenz in Text- und Binärtokens auf. Texttoken sind hierbei Texte mit einer Mindestlänge von 3 Zeichen. Darauf folgt eine weitere Aufteilung der Tokens. Die Binärtokens werden dann in Tokens mit jeweils einzelnen Bytes geteilt; die Texttokens an Leerzeichen in mehrere Texttokens. In der Initial-Clustering-Phase werden dann die einzelnen Nachrichten anhand ihrer Tokenmuster in Cluster eingeteilt. In der anschließenden Recursive-Clustering-Phase wird neben der Einteilung in Text- und Binärtokens innerhalb der Cluster bestimmt, ob ein Token über alle Nachrichten konstant ist oder unterschiedliche Werte hat. Weiterhin werden Differenzen von Tokens an bestimmten Stellen mit Differenzen in der Nachrichtenlänge und des Nachrichtenoffsets verglichen. Korreliert hier der Wert der Tokens mit der Nachrichtenlänge oder dem Nachrichtenoffsets für alle Nachrichten des Clusters, können so Längen- oder Offsettokens bestimmt werden. Nach Bestimmung dieser Tokenattribute (Text/Binär, Konstant/Variabel, Länge, Offset) werden die Tokens wiederum in Untercluster eingeteilt. Außerdem können so genannte Format-Distinguisher-Tokens erkannt werden, anhand derer die Nachrichten in Cluster eingeteilt werden können. Zum Schluss werden die einzelnen Cluster dann wieder zusammengeführt. Hierbei werden die Abfolgen von Tokens dieser Cluster mittels Needleman-Wunsch-Algorithmus [24] ausgerichtet. Anschließend werden paarweise Tokens verglichen. So können konstante Tokens mit variablen Tokens gleichgesetzt werden oder bestimmte Lücken akzeptiert werden. Sind anschließend zwei Cluster kompatibel, werden diese zusammengeführt.

Als nächstes Werk veröffentlichten Cui *et al.* 2008 einen weiteren Artikel zu dem Tool „Tunip“ [10]. „Tunip“ kann dabei zum Reverse Engineering von Netzwerkprotokollen und auch Dateiformaten eingesetzt werden. Hierbei wird jedoch angenommen, dass ein Programm vorliegt, welches die Eingabeformate verarbeiten kann, um mit diesem Analysen auszuführen. Solche Programme liegen nicht für alle Dateisysteme vor oder sind im Betriebssystem integriert und können nicht einfach extrahiert werden. Auch wenn daher einige Ideen dieses Artikels nicht für die Arbeit umgesetzt werden können, werden hier einige Konzepte zum Identifizieren und Kategorisieren von Sequenzen von Tokens vorgestellt.

Alle hier genannten Systeme umfassen einen Prozess, der sich als Pipeline darstellen lässt. Hierbei werden zunächst die Eingabedaten unterteilt (Chunking, Tokenization, Field Identification), dann strukturiert (Structure Discovery, Scoring, Refinement, Clustering, Merging) und zuletzt ausgegeben. Die beschriebenen Prozesse sind gut gegliedert und auf eine hohe Automatisierung ausgelegt. Hierbei gibt es auch Lösungsansätze für Teilprobleme wie die Erkennung von Längentokens oder Unterscheidungsmerkmalen von Clustern von Tokens. Probleme bei der Anwendung dieser Prozesse können getroffene Annahmen bei den jeweiligen Systemen sein, die für Dateisysteme nicht zutreffen. So kann einfach eine große Anzahl an verschiedenen Netzwerkpaketen oder einzelnen Dateien erstellt werden, während dies bei Dateisystemen schwierig zu realisieren ist. Auch die erste Phase, die Eingabedaten zu unterteilen, ist bei Dateisystemen schwierig, da hier in der Regel keine Trennzeichen zwischen den einzelnen Daten vorliegen.

4.2 Dateisystem-Reverse-Engineering

Dateisystem-Reverse-Engineering wurde zu unterschiedlichen Zwecken schon mehrfach betrachtet und durchgeführt. Ein Anwendungsbeispiel ist das Erstellen von Treibern für proprietäre Dateisysteme für

andere Betriebsysteme, so war zum Erstellen eines Linux-NTFS-Treibers das Reverse-Engineering einer NTFS-Partition nötig. Hierzu verfolgten die Entwickler des Treibers die folgenden vier Schritte [25]:

1. Betrachtung der Partition in einem Hex-Editor
2. Ausführung einer Operation (z. B. Erstellen einer Datei)
3. Nochmalige Betrachtung der Partition in einem Hex-Editor
4. Klassifikation und Dokumentation der Änderungen

Diese vier Schritte wiederholten sie, bis eine weitreichende Dokumentation von NTFS entstand. Die Entwickler merken dabei an, wie „hart und zermürbend die Arbeit“ [25] war.

2002 untersuchte Christoph Hellwig das Dateisystem VxFS [20]. Auch in diesem Fall war seine Intention VxFS unter Linux zugänglich zu machen. Hellwig griff zunächst auf Dokumentationen über das Dateisystem zurück. Anschließend untersuchte er die VxFS-Treiber, welche er zur Verfügung hatte. Hierbei nutzte er Reverse Engineering, um die Funktionen der Treiber zu verstehen und diese anschließend selbst zu implementieren.

Ein weiteres Dateisystem aus dem Hause Microsoft ist exFAT, welches speziell für USB-Massenspeicher entwickelt wurde. Um hierfür forensische Analysen möglich zu machen, untersuchte 2009 Robert Shullich das Dateisystem [34]. Als Grundlage nahm Shullich Microsofts Patente zu exFAT, Ähnlichkeiten zu den FAT-Dateisystemen und andere veröffentlichte Informationen zu dem Dateisystem. Daneben führte er eine hardwarenahe Untersuchung durch, bei der immer wieder Abbilder des Dateisystems genommen wurden und dann im Hex-Editor verglichen wurden. Diese Untersuchung entsprach den bei NTFS durchgeführten Schritten. Er nannte dabei als Operationen: Erstellung des Dateisystems, Hinzufügen von Dateien, Löschen von Dateien und anschließend erneutes Hinzufügen von Dateien. Mit den Ergebnissen dieser Untersuchungen erstellte Shullich ein Programm, welches die Inhalte des Dateisystems lesbar ausgibt. Zur Validierung verglich er die Ausgabe mit Systemprogrammen wie DIR, CHKDSK oder dem Windows Explorer.

Schmitt *et al.* [31] untersuchten 2011 YAFFS2. Der Aufbau dieses Dateisystems ist ein offener Standard, der Fokus des Reverse Engineering lag jedoch auf den Wear-Leveling- und Garbage-Collection-Funktionen des Dateisystems, welche eine schnellere Abnutzung von SSD-Festplatten verhindern sollen. Zur Untersuchung des Dateisystems wurde ein Flash-Speicher simuliert und Abbilder dieses Speichers erstellt. Diese wurden dann mit dem Sleuth Kit und Autopsy analysiert [37].

Auch zu ReFS liegen bereits Untersuchungen vor. Paul Green untersuchte in seiner Masterthesis das Dateisystem. Als Basis nutzte er Dokumentationen von Microsoft und einige andere Publikationen von Ballenthin [2] und Metz [22]. Für seine eigenen Untersuchungen verwendete er wieder den Ansatz Datenträgerabbilder zu erstellen und dann im Hex-Editor zu untersuchen. Er erstellte dabei ein leeres Abbild, zwei Abbilder mit jeweils einer Textdatei unterschiedlicher Größe, ein Abbild mit zwei Ordnern mit jeweils unterschiedlich großen Textdateien, ein Abbild mit einer gelöschten Datei im Wurzelverzeichnis und ein Abbild mit einer gelöschten Datei in einem Verzeichnis und einer existierenden Datei in einem Verzeichnis.

Auch Andrew Head untersuchte ReFS [19] und bediente sich der Methode Änderungen auf einer Maschine mittels Abbildern verschiedener Volumes im Hex-Editor zu vergleichen und daraus Schlüsse zu

Tabelle 4.1: Operationen beim Reverse Engineering von Head [19]

Folder Analysis	.doc Analysis	.txt Analysis	.exe Analysis
Metadata Block Changes	Metadata Block Changes	Metadata Block Changes	Metadata Block Changes
Permissions Change	Modify .doc Content	Modifying Content .txt File	Permissions Change
Folder Deletions	Permissions Change	Permissions Change	Deleting .exe File
Renaming Folder	Deleting .doc File	Deleting .txt File	Renaming .exe File
Copying Folder	Renaming .doc File	Renaming .txt File	Copying .exe File
Adding Content	Copying .doc File	Copying .txt File	
Compressing Folder		Creating .txt File	
		Delete .txt File within Folder	

ziehen. Im Gegensatz zu den vorher genannten Arbeiten listete Head viel mehr Operationen auf, welche er durchgeführt hat. Dabei führte er sowohl Operationen auf Verzeichnissen als auch auf verschiedenen Dateitypen aus. Tabelle 4.1 zeigt alle durchgeführten Operationen im Detail.

Proprietäre Dateisysteme tauchen außerdem in Videoüberwachungsanlagen auf. Poole *et al.* [28] gingen bei ihrer Untersuchung wie bei den anderen beschriebenen Ansätzen vor. Ariffin *et al.* [1] bauten einen mehrstufigen Prozess auf, um Videoüberwachungsdaten wiederherzustellen. Hierbei wurde zunächst versucht Signaturen von Videodateien zu bekommen, welche dann per File Carving aus dem Abbild gezogen wurden.

Auch Tobin *et al.* [36] untersuchten Videoüberwachungsanlagen. Sie nutzten hierfür einen so genannten „eavesdropping“-Ansatz, welchen sie aus dem Bereich der Malware-Analyse übernahmen. Bei dem „eavesdropping“-Ansatz werden Zugriffe des Betriebssystems auf die Festplatte überwacht. Hierbei kann zunächst direkt die Blockgröße erkannt werden. Außerdem kann dann zum Beispiel beim Zugriff auf Videos erkannt werden, dass auf bestimmte Blöcke zugegriffen wird. Die Adresse dieser Blöcke wird am Anfang der Partition sowie den jeweils vorhergehenden Blöcken gesucht. Hierdurch kann der hierarchische Zugriff auf Dateien nachvollzogen werden. Auf Basis dieses entwickelten Modells erstellten Tobin *et al.* eine Software, die dann schnell dieses Dateisystem parsen kann. Daten, die jedoch nicht für den einfachen Zugriff auf die Videos genutzt werden, werden nicht beachtet und müssen weiterhin manuell analysiert werden. Außerdem ist die Überwachung der Festplatte nur auf Blockgröße genau, sodass auch hier noch manuelle Analyse nötig ist.

Die beschriebenen Arbeiten ähneln sich sehr stark in ihrer Vorgehensweise, bei der Abbilder des Dateisystems vor und nach einer Operation verglichen werden. Hierbei wird immer von einem vorhandenen Treiber für das Dateisystem ausgegangen. Einen anderen Ansatz beschreibt Andreas Schuster in einer Präsentation zu „Ad-hoc File System Forensics“ [32]. In dieser Präsentation geht Schuster von einem

unbekannten Dateisystem aus, auf welches kein Schreibzugriff besteht. Hiermit fällt die beschriebene Methodik, verschiedene Abbilder zu vergleichen, weg. Er nutzt daher andere Verfahren, um die Struktur des unbekannten Dateisystems herauszufinden. Zunächst berechnet er die Entropie auf dem Abbild des Dateisystems (s. Abschnitt 3.1). Nachdem er diese visualisiert hat, teilt er das Dateisystem in fünf Regionen, welche jeweils unterschiedliche Entropiemuster enthalten. Die erste Region identifiziert er anhand der Signatur `0x55AA` als MBR. In der zweiten Region findet er Informationen zur Blockgröße und Anzahl an Blöcken. Die Blockgröße ermittelt er durch File Carving. In der dritten Region nutzt er Autokorrelation, um die Länge wiederkehrender Datenstrukturen zu identifizieren. In Region fünf erkennt er durch Betrachten der Blöcke im Hex-Editor Verzeichnisse und einzelne Dateien. Aus all diesen Informationen erstellt Schuster den Prototypen eines Parsers für das Dateisystem.

Alle beschriebenen Prozesse können durch die manuelle Bearbeitung sehr genau auf jede Einzelheit von Dateisystemen eingehen. So entsteht hierdurch immer ein gutes Modell der jeweiligen Dateisysteme. Diese Ausführlichkeit erzeugt jedoch einen hohen Aufwand und benötigt sehr viel Zeit.

5 Anforderungen

In der Einleitung (s. Abschnitt 1.2) wurde das Ziel der Arbeit besprochen, ein Konzept zum automatisierten Reverse Engineering von Dateisystemen zu erstellen. In diesem Kapitel sollen nun kurz die Anforderungen an dieses Konzept definiert werden.

5.1 Qualität

Das Reverse-Engineering-Konzept erstellt ein Modell des Dateisystems. Dieses erstellte Modell soll möglichst weit mit dem Modell der Dateisystemspezifikation übereinstimmen. Wichtig ist hier vor allem das Auffinden der essentiellen Daten und Strukturen eines Dateisystems.

5.2 Allgemeingültigkeit

Das erstellte Konzept soll für alle Dateisysteme anwendbar sein und sinnvolle Ergebnisse liefern. Auch für zukünftige Dateisysteme, welche neue Mechanismen und Funktionen bieten, sollen sinnvolle Ergebnisse erstellt werden. Dieser Punkt unterscheidet das erstellte Konzept zu vorhandenen Programmen zur Dateisystemanalyse wie z. B. dem Sleuth Kit. In solchen Programmen sind Spezifikationen für bestimmte Dateisysteme hinterlegt. Hierdurch erfüllen diese Programme alle anderen hier genannten Anforderungen sehr gut. Unbekannte Dateisysteme werden so jedoch nicht erkannt.

5.3 Automatisierung

Der im Konzept definierte Prozess soll möglichst automatisiert durchführbar sein. Dies bedeutet nicht, dass der Prozess vollautomatisch ablaufen muss. Einzelne Teile können weiterhin manuell ausgeführt werden, die Anzahl der benötigten manuellen Schritte soll jedoch minimiert werden. Außerdem soll der Anwender möglichst viel Einfluss auf das Ergebnis des Prozesses haben können, um anderweitig erlangte Erkenntnisse zu dem Dateisystem in die Untersuchung einfließen lassen zu können.

5.4 Effizienz

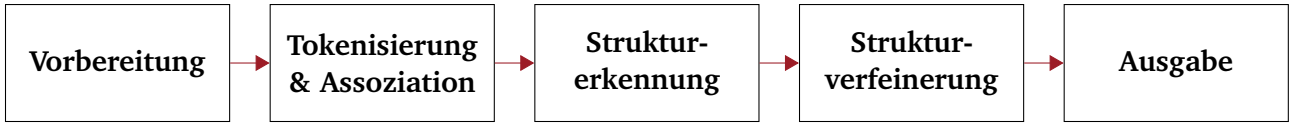
Um einen Vorteil gegenüber manuellem Dateisystem-Reverse-Engineering zu bieten, soll der definierte Prozess mit möglichst wenig Ressourcen durchgeführt werden können. Dieser Punkt überschneidet sich mit dem der Automatisierung, da dieser eine Reduzierung der menschlichen Arbeit vorsieht. Die Effizienz hier beschreibt außerdem eine möglichst geringe Laufzeit, unabhängig von der benötigten Interaktion.

6 Konzept

Im Folgenden wird ein Konzept zum automatisierten Reverse Engineering von Dateisystemen beschrieben. Dieses basiert auf den beschriebenen Reverse-Engineering-Methoden von Dateisystemen und anderen Datenstrukturen aus Kapitel 4 sowie allgemeinen Erkenntnissen zu Dateisystemen aus Kapitel 2. Um diese umzusetzen, werden unter anderem die mathematischen Methoden aus Kapitel 3 genutzt.

Das Konzept beschreibt den Prozess aus einem vorliegenden Dateisystem ein Modell dieses Dateisystems zu erstellen. Dieser Prozess besteht aus fünf einzelnen Phasen, die jeweils auf den Ergebnissen der vorhergehenden Phasen aufbauen. Abbildung 6.1 zeigt einen Überblick über die fünf Phasen zum Dateisystem-Reverse-Engineering. Die einzelnen Phasen werden in den folgenden Abschnitten detailliert erläutert.

Abbildung 6.1: Überblick der Architektur zum Dateisystem-Reverse-Engineering



Zu Beginn des Reverse Engineerings können, analog zur Kryptanalyse [13], verschiedene Situationen vorliegen, die bestimmen, welche Möglichkeiten zur Analyse gegeben sind. Die Analyse von Dateisystemen lässt sich in die folgenden zwei Szenarien einordnen:

Image Only
: Hierbei liegt das Abbild des Datenträgers und nur eine begrenzte Anzahl an Informationen über dessen Bedeutung oder Struktur vor. Ein Beispiel für dieses Szenario ist eine gesicherte Festplatte aus einem Drucker, der zwar eine Anzeige für die gespeicherten Daten besitzt, jedoch keine Möglichkeit diese Daten direkt zu ändern. Dieses Beispiel führt Schuster in seinem Vortrag an [32], der für dieses Szenario Methoden zum Reverse Engineering entwickelt hat. Dieses Szenario ist grundsätzlich der Dateisystemanalyse zuzuordnen, da hier die Verallgemeinerung des erstellten Modells des Dateisystems, und damit das Reverse Engineering, bei nur einem Abbild schwierig ist.

Chosen Data
: Hierbei ist es möglich Daten über ein Betriebssystem in dem Dateisystem zu lesen, zu schreiben und außerdem Abbilder des Dateisystems zu erstellen. Ein Beispiel hierfür ist das Erstellen des NTFS-Treibers für Linux, bei dem die Funktionsweise von NTFS auf einem Windows-Computer untersucht werden konnte.

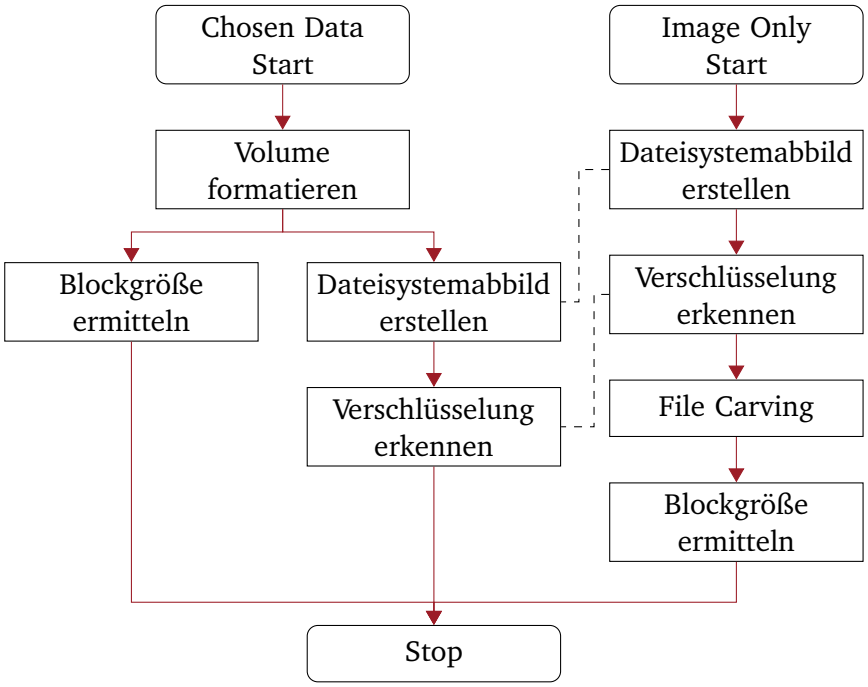
Die Analysemöglichkeiten des *Chosen-Data*-Szenarios schließt die Möglichkeiten des *Image-Only*-Szenarios ein. Eine *Image-Only*-Analyse wäre z. B. das File Carving, welches sich aber auch in einem *Chosen-Data*-Szenario durchführen lässt; das Vergleichen von Abbildern vor und nach einer durchgeführten Operation lässt sich jedoch nur im *Chosen-Data*-Szenario realisieren. Beide Szenarien durchlau-

fen bei dem erstellten Konzept die gleichen fünf Prozessschritte. Innerhalb dieser Schritte werden jedoch teilweise verschiedene Abläufe für die einzelnen Szenarien durchgeführt.

6.1 Vorbereitung

Die Vorbereitung ist die erste Phase im Konzept zum Dateisystem-Reverse-Engineering. Hier wird das Dateisystem für die weitere Untersuchung bereitgestellt und Grundmaße wie die Größe des Dateisystems und die Blockgröße ermittelt. Abbildung 6.2 zeigt die einzelnen Schritte im Ablauf dieser Phase. Abhängigkeiten werden durch die eingezeichneten Pfeile dargestellt. In der Vorbereitung muss unterschieden werden, ob ein *Chosen-Data*- oder ein *Image-Only*-Szenario vorliegt. Im *Chosen-Data*-Szenario wird zunächst ein Volume im zu untersuchenden Dateisystem formatiert. Anschließend kann aus dem Volume oder, im *Image-Only*-Szenario, aus dem Datenträgerabbild ein Dateisystemabbild erstellt werden. Dieses Abbild kann dann auf Verschlüsselung überprüft werden. Die Blockgröße kann im *Chosen-Data*-Szenario einfach ausgelesen werden, während im *Image-Only*-Szenario hierfür vorher ein File Carving notwendig ist. Die einzelnen Schritte der Phase werden in den folgenden Abschnitten genauer erläutert.

Abbildung 6.2: Ablauf der Vorbereitung



6.1.1 Volume formatieren

Bei einem eingebundenen Volume in einem *Chosen-Data*-Szenario ist es sinnvoll dieses vor der eigentlichen Untersuchung zurückzusetzen. Hierbei wird das Volume zunächst mit Null-Bytes (`0x00`) überschrieben, sodass anschließend Änderungen am Volume erkannt werden können. Danach wird das Volume über das Betriebssystem formatiert, wobei das Dateisystem initialisiert und auf das Volume geschrieben wird.

6.1.2 Dateisystemabbild erstellen

Einige Volumes oder Datenträgerabbilder enthalten außer dem zu untersuchenden Dateisystem noch andere Dateisysteme oder Strukturen, wie z. B. den Master Boot Record, welche nicht zum eigentlichen Dateisystem gehören. Das eigentliche Dateisystem muss daher vor der weiteren Analyse extrahiert werden. Diese Extraktion kann entweder aus einem bestehenden Festplattenabbild oder von einem eingebundenem Volume erfolgen. Für diesen Schritt muss die Partitionierung des Volumes bekannt sein, um nur das Dateisystem zu extrahieren. Das Abbild des Dateisystems wird dann als einzelne Datei gespeichert.

6.1.3 Verschlüsselung erkennen

Elektronische Speichermedien oder einzelne Partitionen darauf können komplett verschlüsselt werden. Für eine Nutzung dieser verschlüsselten Bereiche ist zunächst eine Entschlüsselung nötig. Auch ein Reverse Engineering der in solchen Bereichen liegenden verschlüsselten Dateisysteme ist erst nach der Entschlüsselung möglich. Um zu prüfen, ob eine Verschlüsselung vorliegt, kann getestet werden, ob das extrahierte Dateisystem zum einen eine hohe Entropie besitzt (s. Abschnitt 3.1) und zum anderen die darauf gespeicherten Werte zufällig verteilt sind. Letzteres lässt sich durch Berechnung der Chi-Quadrat-Verteilung ermitteln (s. Abschnitt 3.2).

6.1.4 File Carving

Wenn im *Image-Only*-Szenario nur ein Dateisystemabbild vorliegt, ist es sinnvoll, in diesem per File Carving nach Dateien zu suchen. Die Dateien selbst sind hierbei weniger nützlich für den weiteren Prozess, bieten aber dennoch einen Überblick über die im unbekannten Dateisystem gespeicherten Daten. Für den weiteren Prozess werden jedoch vor allem die Positionen der Fragmente der Dateien im Dateisystem benötigt.

6.1.5 Blockgröße ermitteln

Eine erste Gliederung des Dateisystems kann erfolgen, indem das Dateisystem in Blöcke eingeteilt wird, wie sie in den meisten Dateisystemen genutzt werden. FAT macht hier eine Ausnahme, da hier schon vor den regelmäßigen Blöcken (bzw. Clustern) Datenstrukturen vorhanden sind. Die Blockgröße ist in den in Kapitel 2 betrachteten Dateisystemen ein Vielfaches der Sektorgröße. Um die Blockgröße zu erhalten, bieten sich verschiedene Methoden an. Ist das Dateisystem über einen Treiber im Betriebssystem beschreibbar, kann die Blockgröße über das Betriebssystem einfach ausgelesen werden. Ist das Dateisystem nicht beschreibbar, kann die Blockgröße über die Abstände der Dateianfänge herausgefunden werden. Durch das vorhergehende File Carving liegen, wenn im File Carving Dateien gefunden wurden, die Anfänge dieser Dateien vor. Um dann die Blockgröße zu ermitteln, kann der größte gemeinsame Teiler aller gefundenen Dateianfänge als Blockgröße angenommen werden. Hierzu ist jedoch Voraussetzung, dass

mindestens zwei Dateien durch das File Carving gefunden wurden. Werden beim File Carving weniger als zwei Dateien gefunden, kann die Blockgröße auf 512 Byte gesetzt werden. Alle in Kapitel 2 betrachteten Dateisysteme nutzen Vielfache dieses Wertes als Blockgröße. Diese Einteilung hat im folgenden Prozess geringere Auswirkungen, als einen höheren Wert anzunehmen und so einige Blöcke zusammenzufassen, die strukturell nicht zusammen gehören.

6.2 Tokenisierung und Assoziation

Nach dem Extrahieren des Dateisystems und der groben Einteilung in Blöcke in der Vorbereitung werden diese Blöcke nun weiter in einzelne Tokens eingeteilt.

Ein Token ist dabei eine Bytesequenz in einem Block. Definiert wird ein einzelner Token dabei über Offset und Länge. In einem Dateisystemmodell ist ein einzelner Token die kleinste Einheit, welche eine gewisse Information enthält. Die folgenden Abschnitte beschreiben die Tokenisierungsmethoden für das erstellte Konzept des Dateisystem-Reverse-Engineerings. Einzelne semantische Informationen des Dateisystems werden als Attribute bezeichnet. Ein Attribut kann z. B. besagen, dass der Name des Dateisystems „HFS+“ ist oder dass die Länge einer Datei 100 Byte entspricht. Ein Attribut ist dabei definiert über einen Wert und eine Bedeutung. Wenn einem Token ein Attribut zugeordnet wird, wird im Folgenden von einer Assoziation gesprochen. So besteht ein Token z. B. aus den ersten 2 Bytes eines Blocks und eine Assoziation besagt, dass dies das Attribut mit der Bezeichnung des Dateisystems ist.

Abbildung 6.3 zeigt den Ablauf der Tokenisierung und Assoziation. *Chosen-Data*- und *Image-Only*-Szenario laufen in dieser Phase getrennt voneinander ab, auch wenn es innerhalb der Schritte Gemeinsamkeiten gibt.

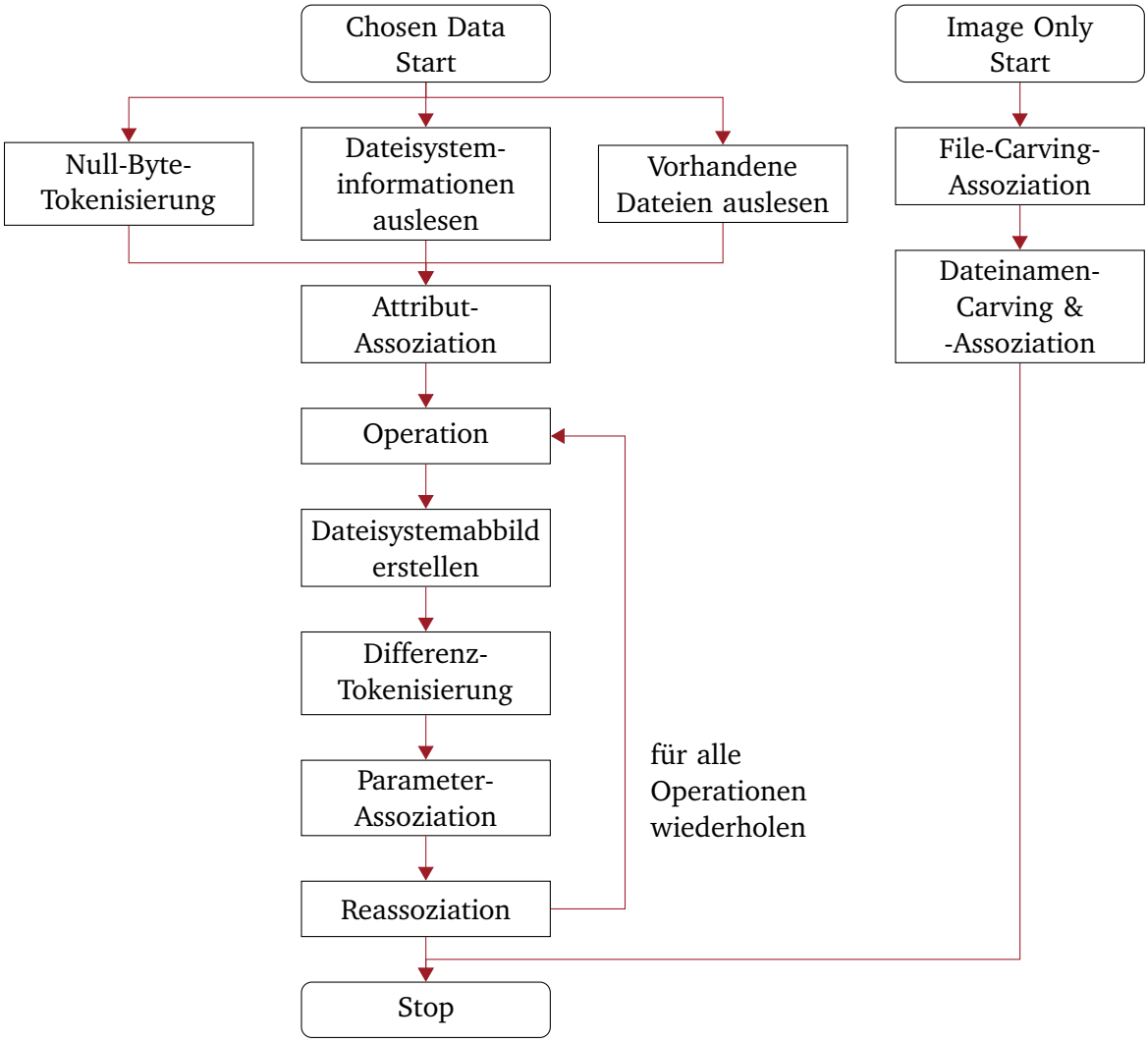
Zu Beginn der Phase im *Chose-Data*-Szenario können auf dem Dateisystem vorhandene Dateien und Dateisysteminformationen ausgelesen werden. Diese werden mit dem in der Vorbereitung erstellten Dateisystemabbild assoziiert, welches vorher anhand von Null-Bytes in Tokens eingeteilt wird. Anschließend werden auf dem Volume verschiedene Operationen durchgeführt. Hierzu wird nach dem gleichen Verfahren vorgegangen wie beim manuellen Reverse Engineering von Dateisystemen (s. Abschnitt 4.2). Nach jeder durchgeführten Operation wird ein Dateisystemabbild erstellt und mittels Differenz-Tokenisierer in einzelne Tokens zerlegt. Die Parameter der Operation werden dann mit diesen neu erstellten Tokens assoziiert. Überschriebene Tokens werden erneut assoziiert. Anschließend kann die nächste Operation ausgeführt werden und der Prozess wiederholt werden.

Das *Image-Only*-Szenario, bietet weniger Möglichkeiten zur Assoziation. Hier können über File-Carving und Dateinamen-Carving jedoch auch Attribute erstellt und Assoziationen gesucht werden.

6.2.1 Dateisysteminformationen und vorhandene Dateien auslesen

Um Assoziationen zu erzeugen, müssen zunächst Attribute des Dateisystems herausgefunden werden. Dazu können Dateisysteminformationen, wie z. B. die Bezeichnung des Dateisystems oder der Name des Volumes, ausgelesen werden. Teilweise werden von den verschiedenen Betriebssystemen beim Erstellen

Abbildung 6.3: Ablauf der Tokenisierung und Assoziation



von Dateisystemen oder Einbinden von Volumes Dateien in die Dateisysteme geschrieben, ohne dass dies der Benutzer initiiert hat. Sowohl die Inhalte dieser Dateien, als auch deren Attribute (z. B. Dateinamen, Zeitstempel, Dateigröße) können ausgelesen werden, um daraus später Assoziationen zu erstellen.

6.2.2 Null-Byte-Tokenisierung

Die einzelnen Blöcke des Dateisystems können über den Null-Byte-Tokenisierer in Tokens zerlegt werden. Hierfür kann zunächst einmal der Inhalt anhand von Null-Bytes (`0x00`) getrennt werden. Grundlage dieser Trennung ist die Annahme, dass das Volume vor Erstellung des Dateisystems komplett mit Null-Bytes überschrieben wurde, wie es auch im *Chosen-Data*-Szenario durchgeführt wird. Alle Bytes, welche nicht den Wert `0x00` haben, sind dementsprechend zu einen späteren Zeitpunkt vom Betriebssystem hinzugefügt worden und können einer Funktion oder Struktur des Dateisystems zugeordnet werden. Ein Nachteil dieser Einteilung ist, dass diese erste Einteilung in Tokens sehr grob ist. Zwei aufeinander folgende Tokens ohne Abstand werden hierbei als ein Token erfasst. Ein weiteres Problem ist, dass zusammengehörige Tokens als einzelne Token angezeigt werden. So wird z. B. in der UTF-16-Kodierung

das Wort „beta.txt“ zu der Bytesequenz `0x62006500740061002E00740078007400`. Diese Bytesequenz wird bei einer Trennung bei Null-Bytes in acht Tokens zerlegt. Tabelle 6.1 veranschaulicht die beiden Probleme bei der Null-Byte-Tokenisierung. Durch diese Schwierigkeiten müssen die Tokens im weiteren Prozess verfeinert werden.

Tabelle 6.1: Probleme der Null-Byte-Tokenisierung

	Beispiel UTF-16	**Beispiel Zeitstempel**
Eingabe	`62006500740061002E00740078007400`	`D257DEFED25BDCBF`
Tokens nach der Null-Byte-Tokenisierung	`62|65|74|61|2E|74|78|74`	`D257DEFED25BDCBF`
Korrekte Tokens	`62006500740061002E00740078007400`	`D257DEFE | D25BDCBF`
Bedeutung	Dateiname: „beta.txt“	HFS-Zeitstempel: 29.10.2015 14:34:38 u. 1.11.2015 15:14:07

6.2.3 Attribut-Assoziation

Bei der Assoziation wird ein Datei- oder Dateisystem-Attribut zu einen Null-Byte-Token hinzugefügt. Hierzu wird für jedes Attribut überprüft, ob der Wert dieses Attributs dem Inhalt eines der erstellten Tokens entspricht. Da Werte in den unterschiedlichen Dateisystemen verschieden kodiert sein können, werden für jedes Attribut diverse Variationen des jeweiligen Wertes erstellt. Wie diese Variationen erstellt werden, wird in Abschnitt 6.2.3.2 beschrieben. Wird eine erstellte Variation in einem Token gefunden, wird die Assoziation zu diesem Token gespeichert. Bei der Assoziierung können Tokens auch schon verfeinert werden. Falls eine gefundene Assoziation über mehrere Tokens geht, können diese zusammengeführt werden und falls eine Assoziation nur den Teil eines Tokens betrifft, kann der ursprüngliche Token in mehrere Tokens aufgeteilt werden. Es können auch mehrere Assoziationen für ein Attribut gefunden werden. Entweder werden dann Informationen redundant im Dateisystem gespeichert oder einer der Funde ist ein false-positive-Fund. Um insgesamt false-positive-Assoziationen zu verhindern, können bestimmte Werte für Assoziationen gefiltert werden, wie im folgenden Abschnitt beschrieben.

6.2.3.1 False Positives

Für kurze zu suchende Attribute ergeben sich viele Treffer. Dies resultiert aus einer hohen Wahrscheinlichkeit, dass die Werte dieser Attribute zufällig auftauchen. Wird z. B. nach der Blockgröße 512 (`0x200`) in allen Blöcken des Dateisystems gesucht, so wird diese Bytesequenz häufig gefunden, auch wenn diese nicht die Blockgröße bezeichnen. Diese Auftrittshäufigkeit lässt sich mathematisch berechnen mit der Formel:

$$\text{erwartete Auftrittshäufigkeit} = \frac{(\text{Blockgröße} - \text{Sequenzlänge} + 1)}{256^{\text{Sequenzlänge}}} \tag{6.1}$$

Mit einer Sequenzlänge von 1 Byte und einer Blockgröße von 4096 zufälligen Bytes ergibt sich ein Wert von 16. Dies bedeutet, dass im Durchschnitt jeder 1-Byte-Wert 16-fach auftaucht. Bei einer Sequenzlänge von 3 Bytes ist die erwartete Auftrittshäufigkeit nur noch bei 0,00024 für ein zufälliges Auftreten der Bytesequenz.

Um auch kleine Attributswerte finden zu können, kann entweder in kleinen Abschnitten gesucht werden, wo der Attributswerte vermutet wird, oder die Entropie, welche in Abschnitt 3.1 beschrieben wird, hinzugezogen werden. Mit dieser kann dann statt der Blockgröße der Informationsgehalt des Blockes als Eingabewert genommen werden. Der Informationsgehalt eines Blockes ist hierbei definiert als:

$$\text{Informationsgehalt} = \text{Blocklänge} * \frac{\text{Blockentropie}}{8} \tag{6.2}$$

Die obige Formel wird dann dementsprechend angepasst:

$$\text{gewichtete erwartete Auftrittshäufigkeit} = \frac{(\text{Informationsgehalt} - \text{Sequenzlänge} + 1)}{256^{\text{Sequenzlänge}}} \tag{6.3}$$

Nach dieser Formel wird z. B. eine Bytesequenz der Länge 1 in einem Block mit dem Informationsgehalt 512 2-fach erwartet.

Beim Suchen von Attributen kann über die gewichtete erwartete Auftrittshäufigkeit berechnet werden, wie oft ein zufälliges Finden des Attributs angenommen wird. Ist die erwartete Häufigkeit für ein zufälliges Auffinden des Attributes zu hoch, wird dieses nicht gesucht, um false positives zu vermeiden.

6.2.3.2 Variationen

Ein Problem beim Suchen von Informationen in Dateisystemen ist, dass diese unterschiedlich kodiert werden können. Um automatisiert diese verschiedenen Kodierungen zu erkennen, können gängige Kodierungsverfahren sowie Kodierungsverfahren aus anderen Dateisystemen genutzt und auf die zu suchenden Informationen angewandt werden. Eine Zahl kann so z. B. mit oder ohne Vorzeichen (signed, unsigned), in unterschiedlicher Byte-Reihenfolge (Big-, Little-Endian) und mit unterschiedlicher Bitanzahl (short, int, long usw.) gespeichert werden. Um möglichst viele Informationen zu assoziieren, müssen daher diese Informationen in möglichst vielen Variationen gesucht werden. Mögliche Variationen werden durch den Datentyp bestimmt. Die folgende Auflistung zeigt mögliche Variationen, die in den in Kapitel 2 beschriebenen Dateisystemen Anwendung finden, und ist beliebig erweiterbar.

Zeichenkette

Groß- und Kleinschreibung; Little- oder Big-Endian; verschiedene Kodierungen (ASCII, ISO 8859-1, Mac OS Roman, UTF-8, UTF-16, UTF-32).

Zahl

Signed oder Unsigned; 8-, 16-, 32-, 64-Bit-Darstellung; IEEE-floating-point-Kodierung; Little- oder Big-Endian

Datum

UTC oder lokale Unixzeit (die jeweils verschieden als Zahl kodiert sein kann); FAT-Kodierung; Little- oder Big-Endian

Byte

Little- oder Big-Endian

Nachdem ein Attribut gefunden wird, wird die Kodierung, welche zu dem Fund geführt hat, mit der Assoziation gespeichert, um später zum Auslesen der Informationen genutzt werden zu können.

6.2.4 Operation

Anstatt nur nach vorhandenen Attributen und Dateifragmenten zu suchen, können diese auch selbst erzeugt werden, falls das Dateisystem beschreibbar ist. Hierzu können verschiedenen Operationen auf dem Dateisystem durchgeführt werden. Die möglichen Operationen unterscheiden sich je nach Dateisystem. Viele grundlegende Operationen, die in den meisten Dateisystemen verfügbar sind, listet Head auf (s. Tabelle 4.1).

6.2.5 Dateisystemabbild erstellen

Nach jeder durchgeführten Operation muss ein Dateisystemabbild erstellt werden. Dieser Prozess funktioniert wie in der Vorbereitung (s. Abschnitt 6.1.2). Dieses Dateisystemabbild kann in den späteren Schritten zum Vergleich mit vorher erstellen Dateisystemabbildern verwendet werden. Es werden hierfür jeweils immer die letzten beiden erstellten Dateisystemabbilder benötigt.

6.2.6 Differenz-Tokenisierung

Bei der Differenz-Tokenisierung werden zwei Abbilder des gleichen Datenträgers benötigt, die zu unterschiedlichen Zeiten erstellt wurden. Diese werden dann blockweise verglichen. Sequenzen von sich unterscheidenden Bytes werden als neue Tokens aufgenommen, gleiche Bytes werden nicht betrachtet. Werden diese neuen Tokens zu schon bestehenden Tokens hinzugefügt, können diese Tokens verfeinert werden. Ein einzelner Token aus der Null-Byte-Tokenisierung kann so in mehrere Tokens aufgeteilt werden, wenn in der Differenz-Tokenisierung sich nur ein Teil des ursprünglichen Tokens ändert.

6.2.7 Parameter-Assoziation

In den neu erstellten Tokens kann im Anschluss nach den Parametern der Operation gesucht werden. Falls diese gefunden werden kann dann eine Assoziation erstellt werden. Nach dem Erstellen einer Datei kann so z. B. deren Inhalt und der Dateiname in den neuen Tokens erkannt und assoziiert werden. Das Verfahren zur Assoziation läuft ebenso ab wie Abschnitt 6.2.3 beschrieben.

6.2.8 Reassoziation

Durch neu erstellte Tokens in der Differenz-Tokenisierung können zuvor gefundene Attribute überschrieben werden und so Assoziationen zwischen Token und Attribut ungültig werden. Hierbei ist es bei noch auf dem Dateisystem vorhandenen Informationen wahrscheinlich, dass ein Attribut nicht gelöscht, sondern nur an eine andere Stelle des Dateisystems verschoben wurde. So kann z. B. der Dateiname einer auf dem Dateisystem vorhandenen Datei beim Erstellen einer neuen Datei an eine andere Position des Dateisystems geschrieben werden. Die Attribute der ungültig gewordenen Assoziation können dann erneut in allen neu erstellten Tokens gesucht werden. Hierbei müssen, im Gegensatz zur ersten Assoziation, keine neuen Variationen erstellt werden, da die Kodierung schon bekannt ist.

6.2.9 File-Carving-Assoziation

In einem *Image-Only*-Szenario können, im Gegensatz zum *Chosen-Data*-Szenario, die für eine Assoziation nötigen Attribute nicht durch einfaches Auslesen oder Durchführung von Operationen erstellt werden. Als Alternative können die Attribute durch verschiedene Carving-Verfahren gefunden werden. File Carving bietet die Möglichkeit Dateiinhalte zu erkennen und mit den erstellten Tokens zu assoziieren. Hierfür werden die in der Vorbereitung ermittelten Positionen und Längen der Dateien in Blöcke umgerechnet und dann für diese Blöcke jeweils ein Token mit Assoziation erstellt.

6.2.10 Dateinamen-Carving und -Assoziation

In den Blöcken, welche nicht zu Dateien gehören, besteht die Möglichkeit Dateisystemstrukturen zu erkennen. Hierfür werden in diesen Blöcken Dateinamen gesucht. Zum Erkennen von Dateinamen können zunächst Zeichenketten einer bestimmten Länge erkannt werden. Für diese Erkennung werden Sequenzen mit druckbaren Zeichen gesucht. Um nicht alle druckbaren Zeichen zu erkennen wird hier eine Mindestlänge hinzugezogen. Diese Vorgehensweise ist analog zu dem Kommandozeilenprogramm `strings` unter UNIX. Diese Zeichenketten werden anschließend auf häufige Dateiendungen überprüft, um Dateinamen in den Blöcken zu erkennen. Falls ein Dateiname erkannt wurde kann hier ein Token mit Assoziation erstellt werden.

6.3 Strukturerkennung

Nachdem für das Dateisystem grundlegende Maße festgelegt wurden und die Dateisystemblöcke in einzelne Tokens zerlegt sind, können in der dritten Phase Strukturen im Dateisystem erkannt werden. Die zu suchenden Strukturen orientieren sich hierbei an den Strukturen, die in allen in Kapitel 2 genannten Dateisystemen vorhanden waren. In dieser und den nächsten Phasen werden die Strukturen dabei wie folgt bezeichnet:

- Dateisystem-Header

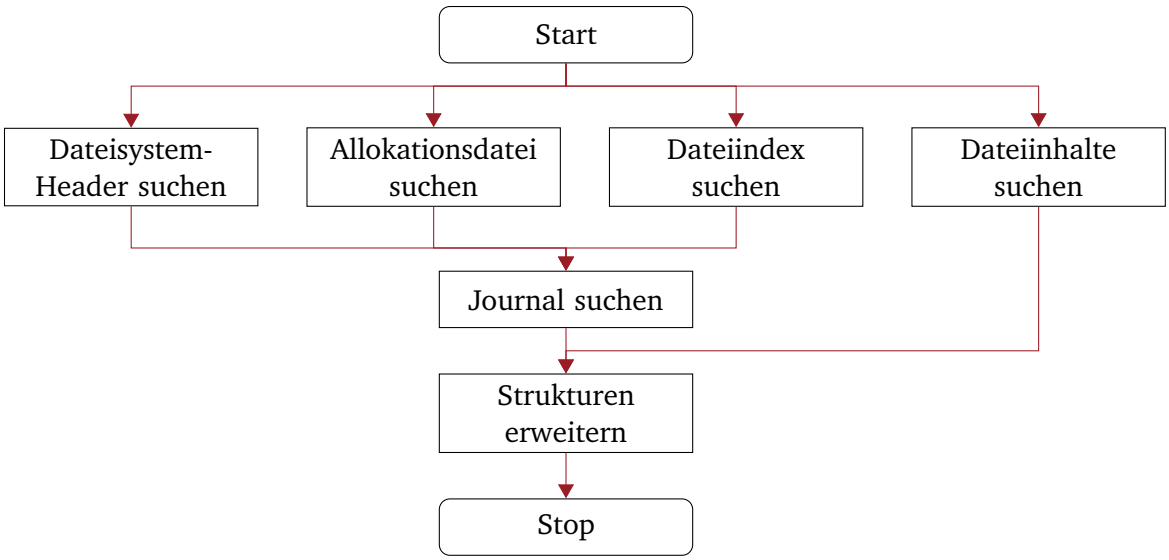
- Allokationsdatei
- Dateiindex
- Journal
- Dateiinhalte

Bei der Erkennung all dieser Strukturen werden Blöcke markiert, in welchen die Strukturen vermutet werden. Die Zuordnung von Strukturen zu den Blöcken wird in den nächsten Abschnitten für die einzelnen Strukturen beschrieben. Hierbei sind teilweise verschiedene Verfahren möglich, um eine Struktur zu erkennen. Durch eine Kombination der Ergebnisse dieser Verfahren sollten einzelne Strukturen möglichst genau bestimmt werden können.

Zu diesen Zuordnungen von Strukturen zu Blöcken werden Konfidenzwerte angegeben, die bestimmen, wie wahrscheinlich die Struktur erkannt wurde. Für jeden Block wird dann die Struktur mit den höchsten Konfidenzwerten als dessen Inhalt angenommen. Durch dieses Verfahren kann ein Block mit dem höchsten Konfidenzwert für eine bestimmt Struktur nicht gewählt werden, wenn der Konfidenzwert einer anderen Struktur in diesem Block höher ist.

Abbildung 6.4 zeigt den Ablauf der Strukturerkennung. Hierbei wird nur auf den Ergebnissen der vorherigen Phasen gearbeitet, wodurch nicht weiter zwischen *Chosen-Data*- und *Image-Only*-Szenarien unterschieden werden muss. Einige Analysen können jedoch aufgrund fehlender Informationen aus den vorherigen Phasen nicht im *Image-Only*-Szenario durchgeführt werden. In der Strukturerkennung wird zunächst nach dem Dateisystem-Header, der Allokationsdatei, dem Dateiindex und den Dateiinhalten gesucht. Diese Untersuchungen sind unabhängig voneinander. Anschließend kann das Journal gesucht werden und nach dem Erkennen aller Strukturen können diese erweitert werden.

Abbildung 6.4: Ablauf der Strukturerkennung



6.3.1 Dateisystem-Header

Als Dateisystem-Header werden die Dateisystemstrukturen zusammengefasst, die den Einstieg in das jeweilige Dateisystem ermöglichen und auf andere Dateisystemstrukturen verweisen. Der Begriff fasst somit den FAT- und NTFS-Bootsektor, den HFS+ Volume Header sowie den Superblock bei ext-Dateisystemen zusammen. Alle in Kapitel 2 beschriebenen Dateisysteme haben zu Beginn des Dateisystems Informationen zum Zugriff auf weitere Dateisystemstrukturen. Dies kann auch von neuen Dateisystemen angenommen werden, da in der Spezifikation eine bestimmte Position für diese Daten festgelegt werden muss.

Aus der vorhergegangenen Phase, der Tokenisierung und Assoziation, gibt es außerdem gegebenenfalls Assoziationen, welche den Typ *Dateisystem* besitzen. Diese gefundenen Tokens können auch auf weitere Strukturen zum Dateisystem oder Volume hinweisen.

6.3.2 Allokationsdatei

Als Allokationsdatei werden in den folgenden Abschnitten die Dateisystemstrukturen bezeichnet, welche den Belegungszustand von einzelnen Dateisystemblöcken angeben. Bei NTFS entspricht dies der Cluster Bitmap, bei HFS+ der Allocation File und bei ext der Block Bitmap und der Inode Bitmap. Allokationsdateien sind in diesen Dateisystemen als Bitmaps realisiert. Hierbei entspricht ein Bit in dieser Bitmap einem Block im Dateisystem. Ist das Bit auf 1 gesetzt, ist der zugehörige Block belegt, ist das Bit auf 0 gesetzt ist der Block frei. Dateien werden in der Regel in konsekutiven Blöcken gespeichert und nur größere Dateien in mehrere Blockreihen fragmentiert. Hierdurch entstehen in den Bitmaps Reihen von voll belegten Bytes (`0xFF`) und Reihen von Null-Bytes (`0x00`), während andere Byte-Werte seltener vorkommen. Durch diese Tatsache kann eine Allocation Bitmap durch eine überwiegende Anzahl von Null-Bytes und voll belegten Bytes erkannt werden. Anschließend kann die Belegung der Blöcke mit der Belegung des Dateisystems abgeglichen werden. Nach der Tokenisierung und Assoziation bekanntermaßen belegte Blöcke müssen in der Bitmap auch an der entsprechenden Stelle mit einem 1er-Bit gesetzt sein. Dieses Verfahren beruht auf der Annahme, dass die Allokationsdatei als Bitmap umgesetzt ist. Das folgende Verfahren kann auch davon unabhängig durchgeführt werden.

Eine weitere Möglichkeit zur Erkennung von Allokationsdateien ist, dass diese beim Speichern und Löschen von Dateien im Dateisystem angepasst werden müssen. Dateien, die einen oder mehrere Blöcke belegen, wirken sich dabei auf ein oder mehrere Bits in der Allokationsdatei aus. Andere Aktionen wie das Erstellen einer leeren Datei oder das Ändern von Metadaten wirken sich jedoch in der Regel nicht auf die Allokationsdatei aus. Auch durch diese Analyse können Allokationsdateien erkannt werden. Das Erstellen einer Datei mit nur wenig Inhalt führt z. B. bei NTFS nicht immer zur Änderung der Allokationsdatei, da die Dateiinhalte dann direkt in der MFT gespeichert werden. Diese Untersuchung ist jedoch nur in einem *Chosen-Data*-Szenario durchführbar, in einem *Image-Only*-Szenario kann lediglich geprüft werden, inwieweit die Allokationsdatei mit der Belegung der Blöcke übereinstimmt.

6.3.3 Dateiindex

Ein Dateiindex beschreibt im Folgenden die Strukturen, welche auf einzelne Dateien verweisen und einen hierarchischen Aufbau von Dateien und Ordnern ermöglichen. Die MFT bei NTFS sowie die Catalog File bei HFS+ sind Beispiele für eine solche Struktur. Ein Dateiindex lässt sich durch gefundene Tokens mit Dateinamen erkennen. Treten diese mehrfach in einem Block auf, kann hier ein Dateiindex oder ein Verzeichnis vorliegen. Dateiindizes und Verzeichnisse sind für die Strukturierung von Dateien zuständig. Eine Abgrenzung der beiden Strukturen ist schwierig. So existieren unter HFS+ keine eigenständigen Strukturen für Verzeichnisse und deren Informationen sind vollständig in der Catalog File vorhanden. Unter NTFS dagegen gibt es neben dem Eintrag in der MFT einen weiteren Eintrag für Verzeichnisse. Für den weiteren Prozess werden daher beide Strukturen als Dateiindex betrachtet.

Neben den Tokens gibt es auch hier im *Chosen-Data*-Szenario die Möglichkeit die Auswirkungen von durchgeführten Aktionen zu betrachten. Alle Aktionen, welche Dateinamen betreffen, z. B. Datei zu erstellen oder Dateien umzubenennen, erzeugen Änderungen an Blöcken des Dateiindex und können so erkannt werden.

6.3.4 Dateiinhalte

Bei einem nicht leeren Dateisystem sind die meisten Blöcke mit den eigentlichen Dateiinhalten belegt. Diese sind als Tokens mit der Länge eines Blocks und der Assoziation mit der Datenkategorie *Inhalt* einfach zu erkennen. Hierbei sind in der Regel Dateiinhalte in mehreren konsekutiven Blöcken vorhanden. Sind zwischen verschiedenen Blöcken mit Dateiinhalten nicht definierte, nicht leere Blöcke vorhanden, können diese manuell geprüft werden, ob auch hier Dateiinhalte gespeichert sind.

6.3.5 Journal

Das Journal enthält Einträge, die im Falle von Fehlern einen integren Dateisystemzustand wiederherstellen können. Hierzu werden im Journal Aktionen und Strukturen des Dateisystems gespeichert. Ein Journal kann so z. B. vorherige Versionen des Dateisystem-Headers oder des Dateiindex enthalten. Bei der Erkennung der anderen Strukturen werden fälschlicherweise auch diese Strukturen gefunden, obwohl diese eigentlich Teil des Journals sind. Einziger Unterschied zu den realen Strukturen ist, dass die Strukturen im Journal eine veraltete Version der Strukturen bieten. Diese Tatsache kann genutzt werden, um das Journal zu erkennen.

6.3.6 Strukturerweiterung

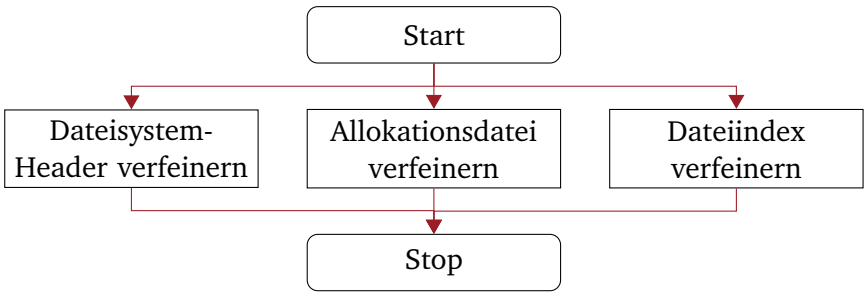
Sind den einzelnen Blöcken des Dateisystems Strukturen zugeordnet worden, können diese anschließend noch erweitert werden. Eine Möglichkeit besteht darin, in angrenzenden Blöcken dieselbe Struktur zu erkennen. Hierzu können die schon bestimmten Blöcke mit den jeweils angrenzenden verglichen

werden. Die Anzahl übereinstimmender Bytes an der gleichen Position bestimmt dabei die Ähnlichkeit. Dieses Verfahren ähnelt dem der Autokorrelation (s. Abschnitt 3.3), wobei hier nicht die Blöcke mit sich selbst verglichen werden, sondern mit anderen Blöcken. Erreicht die Ähnlichkeit einen bestimmten Wert, werden die Strukturen als zusammengehörig erkannt.

6.4 Strukturverfeinerung

In der vorherigen Phase wurden verschiedene Strukturen im Dateisystem erkannt. Diese sind dabei jedoch nur auf der Blockebene bekannt, das heißt einzelne Dateisystemblöcke können den Strukturen zugeordnet werden. So kann ein grober Überblick über das Dateisystem gewonnen werden, die genaue innere Struktur dieser Blöcke ist jedoch weiter nicht bekannt. Diese innere Struktur wird nun weiter analysiert. Der Fokus liegt hierbei auf den essentiellen Strukturen, die für die Funktion des Dateisystems unbedingt nötig sind (s. Kapitel 2), wie dem Dateiindex und dem Dateisystem-Header. Wie in Abbildung 6.5 zu sehen, sind die Verfeinerungen der Strukturen voneinander unabhängig.

Abbildung 6.5: Ablauf der Strukturverfeinerung



6.4.1 Dateisystem-Header

Der Dateisystem-Header verfügt in den in Kapitel 2 betrachteten Dateisystemen über eine Struktur, die nicht weiter in gleichmäßige einzelne Einträge gegliedert ist. Vielmehr werden hier unterschiedlich kodierte Werte nach einer Spezifikation aneinandergereiht, die nicht automatisiert erkannt werden kann. Dennoch ist es möglich hier Verweise auf die anderen Dateisystemstrukturen zu suchen, da deren Position in der Strukturerkennung bestimmt wurde. Es können dabei, wie in der Tokenisierung und Assoziation, Variationen der Verweise auf andere Blöcke erstellt und gesucht werden.

6.4.2 Allokationsdatei

Wurde eine Allokationsdatei in Form einer Bitmap erkannt, kann diese mit demselben Parser für alle Dateisysteme genutzt werden, um zu bestimmen, ob ein Block belegt ist oder nicht. Weitere Verfeinerungen sind hier nicht nötig. Es können jedoch die Tokens, die in den Blöcken der Allokationsdatei gefunden wurden, zu einem Token zusammengeführt werden.

6.4.3 Dateiindex

Der Dateiindex kann, wie in den folgenden Abschnitten beschrieben, in einzelne Einträge zerlegt, mit Feldern mit Abhängigkeiten versehen und als Struktur zusammengeführt werden. Durch diese Schritte schafft man die Grundlage für einen Zugriff auf einzelne Dateien.

6.4.3.1 Einträge finden

Manche Dateisystemstrukturen haben einzelne Einträge, die kleiner als ein Block sind. Vor allem ist dies bei Dateiindizes der Fall, aber auch Attribut-Definitionen bei NTFS und HFS+ nutzen solche Einträge. Um die Länge dieser einzelnen Einträge zu bestimmen, kann bei Einträgen konstanter Länge die in Abschnitt 3.3 beschriebene Autokorrelation genutzt werden. Liegt der höchste Wert der Autokorrelation über einem bestimmten Grenzwert, kann für die Einträge eine feste Länge angenommen werden. Die Länge entspricht hierbei der Verschiebung der Autokorrelation. Die untersuchten Strukturen können dann in Einträge dieser Länge getrennt werden.

Einträge mit variabler Länge sind komplizierter zu bestimmen. Hängt die variable Länge nur von der Länge eines bekannten Tokens, z. B. des Dateinamens, ab, kann über mehrere bekannte Dateinamen bestimmt werden, wie viele Bytes ein Eintrag außerhalb der Dateinamen umfasst. Hierzu wird die Distanz zwischen dem Ende eines bekannten Tokens und dem Start des nächsten bekannten Tokens für alle bekannten Tokens berechnet. Bei ausreichend erkannten Tokens ist die am häufigsten auftretende Entfernung zwischen den Tokens die Anzahl an Bytes, die den bekannten Token umschließen. Es fehlt dann noch die Information, wie viele dieser Bytes vor bzw. hinter dem bekannten Token liegen. Dies kann nur über den ersten Eintrag im Block herausgefunden werden, indem die Distanz zwischen Blockanfang und erstem Token gemessen wird. Somit ist dann bekannt, wie viele Bytes die einzelnen Einträge bestimmen und aus gefundenen Token können Einträge erstellt werden.

6.4.3.2 Felder mit Abhängigkeiten finden

Nachdem einzelne Einträge bekannt sind, können weitere Felder erkannt werden. Hierzu zählen vor allem Längen- und Offsetfelder. Hierzu können Bytes an bestimmten Positionen in den Einträgen z. B. mit dem Byteoffset des Dateiinhalts oder der Länge des Tokens verglichen werden. Durch die verschiedenen Implementierungen dieser Werte in Dateisystemen wird ein Gleichungssystem mit Gleichungen ersten Grades aufgestellt. Gibt es dabei die gleiche ganzzahlige Lösung für alle Einträge, kann dieses Feld als Längen- oder Offsetfeld markiert werden. Bei genügend erkannten Einträgen liegt ein überbestimmtes Gleichungssystem vor, für das es jedoch, falls die Felder existieren, genau eine Lösung gibt. Durch die höhere Anzahl an Gleichungssystemen werden somit false-positive-Funde für die Felder weniger wahrscheinlich als bei den einfachen Assoziationen in vorherigen Phasen. Insgesamt ist diese Methode eine Alternative zu den Variationen (s. Anschnitt 6.2.3.2), die jedoch nur bei mehreren gleichartigen Strukturen angewendet werden kann.

Tabelle 6.2 zeigt als Beispiel das Auffinden von Feldern, die die Länge des Dateinamen beschreiben aus einer Liste von Einträgen. Die dargestellten Einträge stammen aus der Catalog File unter HFS+. Für das Auffinden von Feldern mit Abhängigkeiten werden alle Bytes der Einträge einzeln geprüft, Tabelle 6.2 zeigt die Prüfung für das erste Byte des Eintrags. Aus dem Wert des Bytes an dieser Position sowie der zu prüfenden Abhängigkeit, hier der Länge des Dateinamens, wird dann eine Gleichung pro Eintrag erstellt. Alle gezeigten Gleichungen lassen sich mit $a = 2$ und $b = 6$ lösen, somit wurde hier ein Feld, welches die Länge des Dateinamens angibt, gefunden.

Tabelle 6.2: Finden von Feldern mit Abhängigkeiten

Dateieintrag	Byte an Position 0 des Eintrags	Länge des Dateinamens	Gleichung
alpha	16 (`0x10`)	5	$5 * a + b = 16$
omega	16 (`0x10`)	5	$5 * a + b = 16$
beta.txt	22 (`0x16`)	8	$8 * a + b = 22$
gamma.txt	24 (`0x18`)	9	$9 * a + b = 24$
delta.jpg	24 (`0x18`)	9	$9 * a + b = 24$

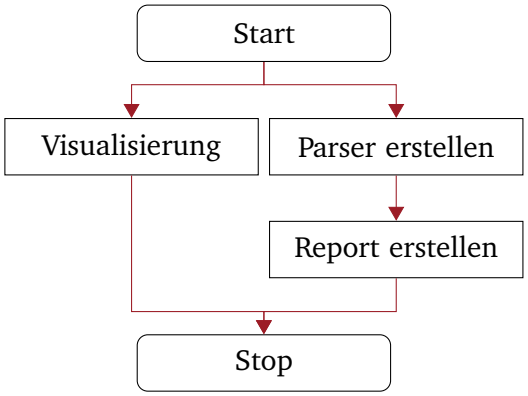
6.4.3.3 Einträge zusammenführen

Neben dem Finden von Feldern mit Abhängigkeiten können die gefundenen Einträge untereinander verglichen und zusammengeführt werden. Dazu werden die Typen der einzelnen Tokens in den Einträgen, welche an der gleichen Position liegen, verglichen und dann in einer Struktur zusammengeführt. Der Typ eines Tokens wird über die assoziierte Kodierung des Tokens sowie die Länge des Tokens bestimmt. Diese Kodierung wurde in der Tokenisierung und Assoziation hinzugefügt (s. Abschnitt 6.2.3). Aus diesem Prozess entsteht eine Struktur, die alle gegebenen Einträge widerspiegelt. Bei ausreichend verschiedenen Einträgen kann diese Struktur anschließend dazu genutzt werden, beliebige Einträge zu parsen.

6.5 Ausgabe

Nachdem die gefundenen Dateisystemstrukturen weiter verfeinert wurden, ist ein Modell des Dateisystems entstanden. Dieses interne Modell soll nun im letzten Schritt für die weitere Verwendung ausgegeben werden. Zum einen wird aus dem Model ein Parser für das Dateisystem erstellt, welcher Abbilder des Dateisystems einlesen und die darin enthaltenen Dateien anzeigen soll, zum anderen werden erstellte Artefakte und das Modell sowohl textbasiert als auch visualisiert ausgegeben. Diese Ausgaben bieten einen Überblick über das erstellte Modell und bilden die Basis für weitere Untersuchungen am Dateisystem. Abbildung 6.6 zeigt die verschiedenen Schritte der Ausgabe.

Abbildung 6.6: Ablauf der Ausgabe



6.5.1 Parser

Wenn alle Phasen des Prozesses erfolgreich durchlaufen wurden, kann aus dem erstellten Modell ein Parser für das Dateisystem entwickelt werden. Dieser besteht aus den erkannten und verfeinerten Strukturen für den Dateisystem-Header, den Einträgen im Dateiindex sowie der Struktur der Allokationsdatei. Mit dem folgenden Algorithmus können so die im Dateisystem gespeicherten Daten angezeigt werden:

1. Dateisystem-Header parsen
2. Blockgröße auslesen (oder erkannte Blockgröße als Standard wählen)
3. Position und Länge des Dateiindex auslesen
4. Dateiindex parsen
5. Dateiname, Position und Länge der Dateien auslesen
6. Dateien darstellen

6.5.2 Visualisierung

In Kapitel 2 wurde zur Visualisierung des Aufbaus von Dateisystemen eine einzeilige Tabelle gewählt, die die verschiedenen Strukturen aufzeigt. Diese Darstellung wird auch in der Literatur gewählt [6]. Aus den gewonnenen Daten kann eine solche Darstellung generiert werden. Diese zeigt die Lage der verschiedenen erkannten Strukturen im Dateisystem.

6.5.3 Report

Neben der Visualisierung werden Daten über die Strukturen und den Aufbau des Dateisystems als Report ausgegeben. Der Report enthält Erkenntnisse aus allen fünf Phasen. Diese Informationen können zum manuellen Auslesen des Dateisystems genutzt werden.

7 Implementierung

Nachdem das vorherige Kapitel das theoretische Konzept zum Dateisystem-Reverse-Engineering beschrieben hat, wird in diesem Kapitel die praktische Umsetzung des Konzeptes beschrieben. Implementiert wurde das Konzept in mehreren Python-Programmen. Genutzt wurde die Python-Version 3.4. Neben der Standardbibliothek wurden auch die folgenden Pakete genutzt:

construct Zum Erstellen eines deklarativen Parsers. Dieser Parser definiert eine Datenstruktur, welcher aus Binärdaten ein Datenmodell erstellen kann.

dill Zur Serialisierung von Programmteilen, z. B. erstellten Tokens, um diese zwischen den Programmen zu übergeben.

PyQt5 Zum Erstellen von grafischen Benutzeroberflächen zur Visualisierung der Ergebnisse.

sympy Zum Lösen von Gleichungssystemen in der Verfeinerung von Strukturen (s. Abschnitt 6.4.3.2).

Neben diesen Python-Paketen werden noch Kommandozeilenprogramme des jeweiligen Betriebssystems genutzt. Unter Windows sind dies `mountvol` und `format` und unter OS X das Tool `diskutil`. Als externe Programme werden neben Python noch die folgenden zwei Programme genutzt:

mmls Tool aus dem Sleuth Kit zum Extrahieren von Partitionen.

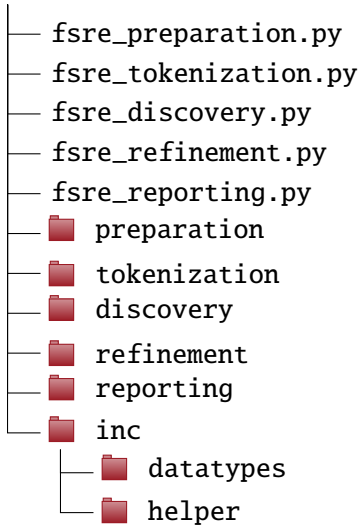
photorec File-Carving-Programm, welches eine Liste der gefundenen Dateien ausgibt.

7.1 Projektstruktur der Implementierung

Die Implementierung des Konzepts teilt sich in fünf einzelne Programme, die jeweils eine Phase des Konzeptes durchführen. Hierdurch ist es möglich Ergebnisse der Programme zwischen den Phasen zu überprüfen, gegebenenfalls anzupassen und so Folgefehler zu vermeiden. Es können auch alle Programmteile über ein Skript hintereinander ausgeführt werden. Für jedes der Programme gibt es eine Hauptdatei, welche ausgeführt werden kann. So z. B. `fsre_preparation`[1] für die Vorbereitungsphase und `fsre_reporting` für die Ausgabephase. Der Code zu den einzelnen Phasen befindet sich jeweils in einem Unterordner, welcher nach der jeweiligen Phase benannt ist. Daneben gibt es ein Verzeichnis `inc`, welches Code enthält, welcher in mehreren Phasen genutzt wird. Abbildung 7.1 zeigt den Verzeichnisbaum der Implementierung. Die einzelnen Pakete und Module werden in den folgenden Abschnitten beschrieben.

[1] Zur besseren Lesbarkeit werden die Dateinamen ohne die Dateiendung „.py“ dargestellt.

Abbildung 7.1: Übersicht über die Verzeichnisstruktur der Implementierung



7.2 Vorbereitung

Der Code für die erste Phase des Konzeptes, die Vorbereitung, liegt im Paket `preparation`. Während der Ablauf der Phase in der Datei `fsre_preparation` implementiert ist, liegen im Paket vier Module, die in dieser Phase genutzt werden. Daneben wird noch das `VolumeCreator`-Module aus dem `inc`-Verzeichnis verwendet. Dieses Modul wird verwendet, um im *Chosen-Data*-Szenario ein Volume zu erstellen und mit dem zu untersuchenden Dateisystem zu formatieren. Mittels der `dd`-Funktion wird dann ein Abbild dieses Volumes erstellt. Die `dd`-Funktion erstellt, analog zum UNIX-Befehl `dd`, eine blockweise Kopie einer Quelldatei in eine Zieldatei. Die Quelldatei kann dabei auch ein eingebundenes Volume sein. Das `Extractor`-Modul kann dazu genutzt werden, das Dateisystem aus einem Festplattenabbild zu extrahieren. Hierbei wird das `mmls`-Tool aus dem Sleuth Kit verwendet. Bei mehr als einer nutzbaren Partition gibt es eine Rückfrage an den Nutzer, welche Partition ausgewählt werden soll. Mit diesem Dateisystemabbild kann dann mittels `EncryptionChecker` in 4096-Byte-Blöcken überprüft werden, ob dieses verschlüsselt ist.

Ist einer der geprüften Blöcke nicht verschlüsselt, bricht der Test ab und der Rest der Phase läuft weiter. Bei einer Verschlüsselung aller Blöcke des Dateisystems bricht das Programm ab, da dann keine weiteren Informationen über das Dateisystem ermittelt werden können. Wenn das Dateisystem nicht verschlüsselt ist, kann dieser Test sehr schnell durchgeführt werden.

In einem *Image-Only*-Szenario wird in der Vorbereitung ein File Carving durchgeführt. Hierzu wird das externe Programm `photorec` aufgerufen. Vom Ergebnis dieses Programms wird nur die erstellte `report.xml` genutzt, die gefundenen Dateien werden nicht beachtet. Die `report.xml` ist nach dem DFXML-Format aufgebaut, welches Garfinkel 2011 vorgestellt hat [16]. `photorec` kann einfach gegen andere File-Carving-Programme ausgetauscht werden, wenn diese auch DFXML-Dateien erzeugen.

Zuletzt wird die Blockgröße, welche eine wichtige Rolle im weiteren Prozess einnimmt, im `Measurer`-Modul ermittelt. Hierbei wird entweder die vom Betriebssystemtreiber ermittelte Größe genommen oder die Größe über eine DFXML-Datei ermittelt.

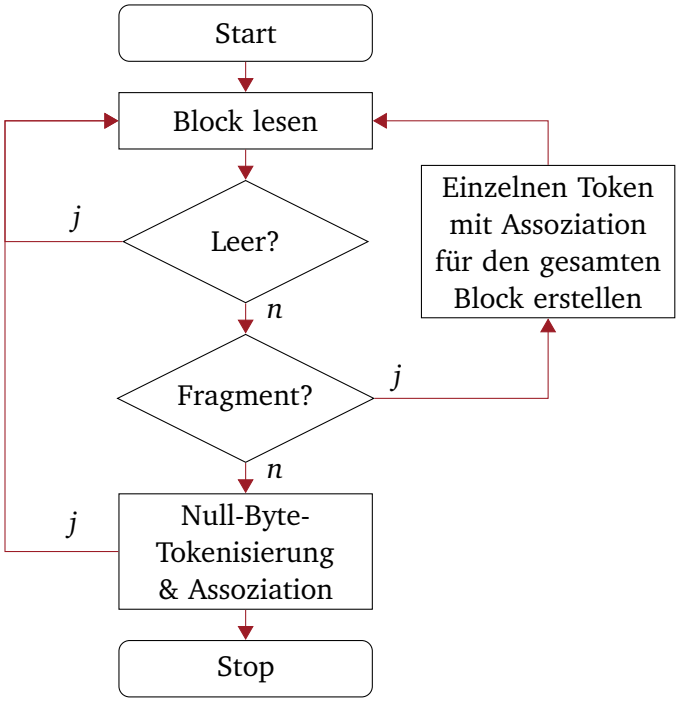
7.3 Tokenisierung und Assoziation

Die Tokenisierung und Assoziation ist die komplexeste Phase. *Chosen-Data*- und *Image-Only*-Szenario laufen in dieser Phase getrennt voneinander ab.

Im *Chosen-Data*-Szenario wird zunächst wieder das `VolumeCreator`-Modul verwendet, was im Konzept abweichend beschrieben ist. Beim mehrfachen Durchführen der Tokenisierung, z. B. mit unterschiedlichen Blockgrößen, würde ohne Überschreiben des Volumes im `VolumeCreator` bei den späteren Durchläufen versucht die gleichen Dateien zu erzeugen. Dadurch würden Konflikte entstehen, die durch ein erneutes Anlegen des Dateisystems vermieden werden. Ein einzelner Durchlauf der Tokenisierung im Anschluss an die Vorbereitung benötigt keinen `VolumeCreator`. Danach erfolgt die Durchführung der Phase nach dem Konzept. Über den `FSInfoAttributer` und `KnownFilesAttributer` werden Attribute aus Dateisystem- und Dateiinformationen erstellt. Der `FSInfoAttributer` ist dabei wieder betriebssystemabhängig und erzeugt verschiedenen Attribute auf unterschiedlichen Betriebssystemen.

In dem in der Vorbereitung erstellten Dateisystemabbild können, nach dem Ermitteln der vorhandenen Attribute und Dateiinhalte, Assoziationen gesucht werden. Dabei wird, wie in Abbildung 7.2 dargestellt, jeder einzelne Block im Dateisystem einzeln betrachtet.

Abbildung 7.2: Ablauf der Null-Byte-Tokenisierung und Assoziation



Ist der Block leer, wird er nicht weiter bearbeitet und der nächste Block gelesen, was in neu formatierten Dateisystemen einen Großteil der Blöcke ausmacht und so zur Performanz dieser Phase beiträgt. Ist der

Block nicht leer, wird er im `ContentAttributer` mit den Inhalten der gefundenen Dateien verglichen. Diese werden dabei in Fragmente in der Größe der Blöcke unterteilt. Wird ein Dateifragment gefunden, wird ein einzelner Token für den Block erstellt und der nächste Block behandelt. Wird kein Dateifragment gefunden, wird der Block mittels des `NullByteTokenizer` in Tokens zerlegt. Der grundlegende Algorithmus dazu ist in Algorithmus 1 dargestellt. Diese Tokens werden dann mit den gefundenen Attributen assoziiert. Im Anschluss wird auch hier der nächste Block gelesen und bearbeitet.

Algorithmus 1 Null-Byte-Tokenisierung

```
 1: function CREATE TOKEN(data)
 2:     tokens = []
 3:     for byte in data :
 4:         if byte is not 0x00 :
 5:             if currenttoken is None :
 6:                 Create newtoken at this byte offset with length = 1
 7:                 currenttoken = newtoken
 8:             else :
 9:                 Increase length of currenttoken
10:         elif currenttoken is not None :
11:             Add currenttoken to tokens
12:             currenttoken = None
13:     if currenttoken is not None :
14:         Add currenttoken to tokens
15:     return tokens
```

Erstellte Tokens werden dem `TokenStore` hinzugefügt. Diese Klasse ist eine Unterklasse des Python-Typs `list()`. Hierdurch kann der `TokenStore` als normale Liste von Tokens genutzt werden und um andere Listen von Tokens erweitert werden. Daneben bietet der `TokenStore` noch einige Funktionen, um nur bestimmte Tokens der Liste zu erhalten. Diese gleichen Datenbankabfragen und nutzen die Listen-Abstraktion (engl. List Comprehension) von Python. Hiermit können z. B. aus dem `TokenStore` Tokens abgefragt werden, die in einem bestimmten Bereich liegen oder eine Assoziation besitzen. Außerdem bietet der `TokenStore` die Möglichkeit Tokens hinzuzufügen und dabei Überschneidungen zwischen den Tokens zu vermeiden. Algorithmus 2 zeigt wie das Einfügen eines Tokens durchgeführt werden kann. Im Gegensatz zur Definition der Tokens über Start und Länge wird hier zur Vereinfachung des Algorithmus statt der Länge das Ende der Tokens genutzt. Dies ist für einen einzelnen Token definiert als Ende = Start + Länge − 1.

Für alle schon existierenden Tokens wird dabei in Zeile 3 des Algorithmus überprüft, ob es eine Überschneidung mit dem neuen Token gibt. Ist dies der Fall, wird in Zeile 4 geprüft, ob der neue Token vor dem schon existierenden Token anfängt und dahinter endet. Der schon existierende Token kann dann gelöscht werden. In den anderen beiden Fällen überdeckt der neue Token den anderen nur teilweise, sodass der schon existierende Token angepasst werden muss. Zuletzt kann der neue Token in die Sequenz der Token eingetragen werden, ohne dass Überschneidungen zwischen den Tokens existieren.

Die erstellten und in den `TokenStore` eingefügten Tokens können mit den vorher erstellten Attributen verglichen werden. Hierzu werden im `Associator` die Attribute zunächst wie in Abschnitt 6.2.3.2 beschrieben variiert. Diese Variationen werden in den Modulen im Ordner `tokenization/variation`

Algorithmus 2 Tokens *ins* in die Tokensequenz *tokens* einfügen

```
1:  function INSERT TOKEN(ins, tokens)
2:      for token in tokens :
3:          if ins.block == token.block and ins.end ≥ token.start and ins.start ≤ token.end :
4:              if ins.start ≤ token.start and ins.end ≥ token.end :
5:                  Delete token
6:              elif ins.start > token.start :
7:                  token.end = ins.start - 1
8:              elif ins.end < token.end :
9:                  token.start = ins.end + 1
10:     Add ins to tokens
11:     return tokens
```

umgesetzt. Hierbei können verschiedene Datentypen an die Funktion `vary` im `Variator` übergeben werden, welcher diese für den Datentyp entsprechende Module weiterleitet. Als Rückgabe entstehen hier drei Elemente. Zum einen wird die Variation des Eingabewertes als Bytes zurückgegeben. Zum anderen werden noch die Kodierung zum Parsen des Wertes sowie der Quellcode zum Parsen der Kodierung zurückgegeben. Beides wird in späteren Phasen benötigt. Das Modul `construct_extras` erweitert das `construct`-Paket um neue Kodierungen, die vom `DateTimeVariator` genutzt werden.

Im `Associator` werden die Variationen für den Wert eines Attributes zwischengespeichert, um diese nicht immer neu berechnen zu müssen. Vor der Speicherung wird die Länge der Variationswerte geprüft und zu kurze Variationswerte verworfen. Dazu wird anhand der Blockgröße und der Entropie eines Blocks eine Minimallänge nach dem im Abschnitt 6.2.3.1 beschriebene Verfahren für die Werte berechnet. Hierbei wird die Formel nicht für jeden Variationswert berechnet, sondern für jeden Block die Minimallänge bestimmt, welche ein Token haben muss, um mit einer Wahrscheinlichkeit kleiner als die Konstante `CONFIDENCE_THRESHOLD` zufällig aufzutauchen. Diese Umstellung führt zu weniger Berechnungen und somit zu einem schnelleren Programmablauf.

Abschließend passiert der eigentliche Vergleich der Variationswerte mit den Token. Hierzu werden die Tokens und die umschließenden Bytes in Größe der Variation betrachtet. Es wird dabei nicht allein der Token betrachtet, da dieser keine Null-Bytes enthalten kann, die jedoch in den Variationswerten auftauchen können. Hier wird dann auch noch die genaue Wahrscheinlichkeit für das zufällige Auftreten des Tokens berechnet und mittels der Konstante `ASSOC_CONFIDENCE` gewichtet. Das Ergebnis davon wird als Konfidenzwert dieser Assoziation gespeichert. Konfidenzwerte bestimmen bei mehreren gefundenen Assoziationen, die sich überschneiden, welche davon angenommen und welche verworfen werden.

Eine Assoziation enthält dabei eine Kategorie, die den Dateisystemkategorien von Brian Carrier entspricht (s. Abschnitt 2) sowie eine Beschreibung. Außerdem enthalten Assoziationen Attribute des Dateisystems und Volumes (z. B. Name des Volumes) sowie der im Dateisystem gespeicherten Daten (Dateinamen, Dateiattribute, Dateiinhalte, Ordnernamen).

Nach dem Durchlaufen aller Blöcke sowie deren Tokenisierung und Assoziation können dann die Dateisystemoperationen durchgeführt werden. Hierbei ist es wichtig, dass nach den durchgeführten Operationen die Änderungen dieser auf das Speichermedium geschrieben werden, bevor ein Abbild des Mediums

erstellt wird. Durch Buffering-Methoden werden teilweise Änderungen erst nach einiger Zeit auf Dateisysteme geschrieben. Mittels der Python-Funktionen `file.flush()` und `os.fsync(file.fileno())` können Inhalte einer Datei auf das Speichermedium übertragen werden. Um Änderungen an Metadaten (z. B. Dateinamen) einer Datei oder eines Ordners auf das Speichermedium zu schreiben, ist es meist möglich `os.fsync()` auf den übergeordneten Ordner anzuwenden. Unter HFS+ wurden jedoch auch hier die Änderungen nicht immer direkt in das Dateisystem eingetragen. Auch beim Warten über mehrere Sekunden kann nicht sichergestellt werden, dass alle Änderungen auf den Datenträger übernommen werden, da nicht alle Strukturen gleichzeitig geschrieben werden. Die einzige Möglichkeit zum sicheren Schreiben aller Metadaten ist daher das Aus- und Einhängen des Datenträgers. Hierbei können jedoch auch ungewollte Tokens entstehen.

Nach dem Durchführen der Operation und Extrahieren des Dateisystems wird abermals eine Schleife über alle Blöcke durchgeführt. Abbildung 7.3 zeigt eine Übersicht dieses Ablaufs. Auch hier werden zunächst leere Blöcke verworfen. Anschließend wird der Block direkt über den `DiffTokenizer` tokenisiert. Hierdurch können veränderte Blöcke erkannt werden. Veränderte Blöcke werden dann wieder über den `ContentAttributer` mit den Dateifragmenten der Operation verglichen. Die schon erstellten Tokens werden anschließend, wie oben beschrieben, assoziiert. Wenn durch den `DiffTokenizer` vorher vorhandene Assoziationen überschrieben, werden die vorher assoziierten Attribute gesammelt. Diese gesammelten Attribute werden im Anschluss an die Schleife erneut gesucht.

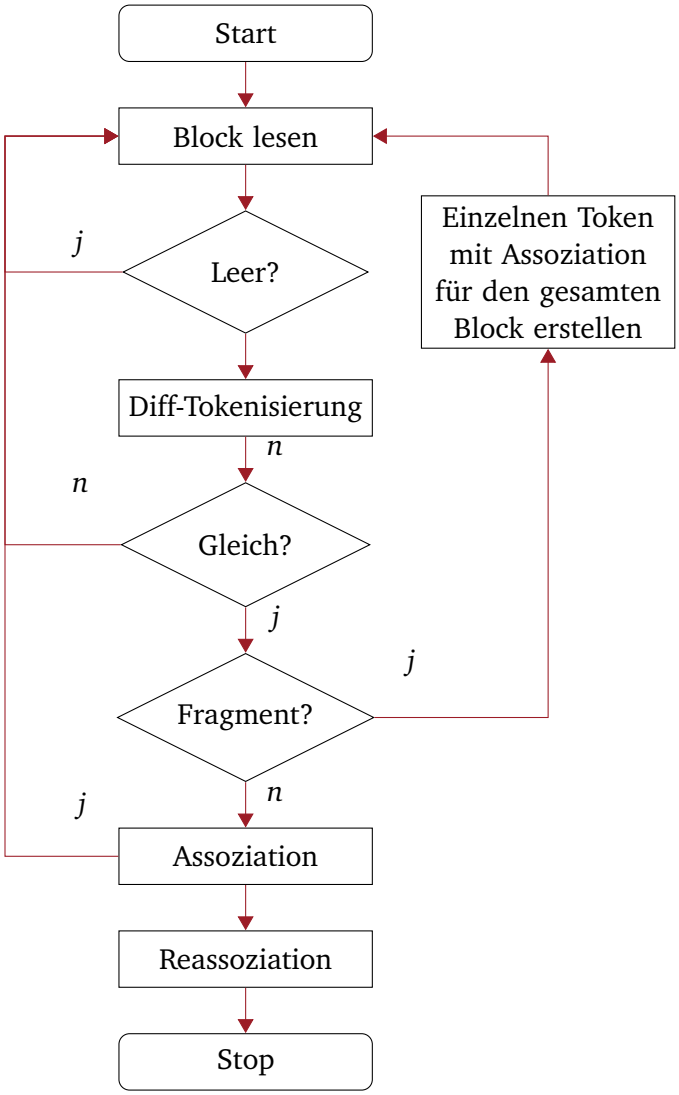
Im *Image-Only*-Szenario wird über den `DfxmlAttributer` bestimmt, welche Blöcke Dateifragmente enthalten. Die Blöcke werden wie im *Chosen-Data*-Szenario alle durchlaufen. Hier wird zuerst geprüft, ob die Blöcke in der DFXML-Datei auftauchen, um den Block direkt damit zu assoziieren und nicht auslesen zu müssen. Erst danach wird der Block ausgelesen und auf Inhalt geprüft. Tokens werden nicht anhand von Null-Bytes erstellt, da dieses Prozedere bei vollen Dateisystemen sehr lange dauern kann. Stattdessen wird mittels des `FilenameCarvingAttributer` über einen regulären Ausdruck nach ASCII- und UTF-Zeichenketten gesucht. Bei den gesuchten UTF-Zeichenketten werden nur Buchstaben beachtet, die auch im ASCII-Alphabets vorkommen und nicht alle UTF-Buchstaben beachtet, da dies zu vielen false positives führen würde. Dateinamen mit Zeichen außerhalb dieses Alphabets, wie z. B. die deutschen Umlaute, werden so jedoch nicht erkannt.

7.4 Strukturerkennung

In der Strukturerkennung werden die vorher erstellten Tokens genutzt. Diese werden am Ende der Tokenisierung und Assoziation serialisiert gespeichert und können dann in der Strukturerkennung geladen werden.

Das `discovery`-Paket besteht vor allem aus mehreren Discoverer-Klassen zur Erkennung der verschiedenen Dateisystemstrukturen (`FileSystemHeadDiscoverer`, `AllocationFileDiscoverer` usw.). Diese implementieren jeweils eine oder mehrere Funktionen zum Auffinden der Strukturen. Jede Funktion gibt dabei eine Liste mit Tupeln mit den gefundenen Blöcken und einem Konfidenzwert zwischen 0 und 1 wieder. Die meisten dieser Konfidenzwerte sind als Konstanten in `inc/const` definiert. Die Werte wer-

Abbildung 7.3: Ablauf der Tokenisierung und Assoziation des Operationen



den für jeden Block im `AbstractDiscoverer` über alle Funktionen gemittelt und dann dem jeweiligen Block zugeordnet.

`AllocationFileDiscoverer` und `FileIndexDiscoverer` nutzen, neben den Tokens und dem Abbild, auch noch die Informationen, auf welche Blöcke die durchgeführten Operationen Auswirkungen hatten. Im *Chosen-Data*-Szenario wurden diese Auswirkungen der durchgeführten Operationen auf Blöcke in der Tokenisierung und Assoziation in einer Datei gespeichert. Diese Datei wird in den beiden Discoverern ausgelesen und anschließend werden gemeinsame oder unterschiedliche Blöcke der einzelnen Operationen erkannt. Dies geschieht über die `set`-Funktion, welche in Python-Operationen der Mengenlehre (z. B. Vereinigung, Schnittmenge) unterstützt.

Einzelne Blöcke sind im `Block`-Modul definiert. Hier liegt außerdem die BlockAssociation-Klasse, welche zum Zuordnen von Strukturen zu den Blöcken genutzt werden kann. Das `BlockStore`-Modul ist vom Python-Typs `dict()` abgeleitet und bietet wie der `TokenStore` die Möglichkeit bestimmte Blöcke abzufragen.

Neben diesen Modulen gibt es das `StructureEnlarger`-Modul, welches den in Abschnitt 6.3.6 genannten Ansatz zum Erweitern von Strukturen umsetzt.

7.5 Strukturverfeinerung

Analog zu den Discoverer-Modulen im `discovery`-Paket gibt es im `refinement`-Paket die drei Refinment-Module `FileSystemHeadRefinement`, `AllocationFileRefinement` und `FileIndexRefinement`. Diese Module dienen zum Erkennen der inneren Struktur der jeweilige Dateisystemstrukturen.

Zum Auffinden von Verweisen auf andere Dateisystemstrukturen nutzt `FileSystemHeadRefinement`, wie in der Tokenisierung und Assoziation, Variationen der Positionen der Strukturen, um diese zu erkennen.

Zum Aufteilen des Dateiindex in einzelne Einträge und anschließendem Zusammenführen dieser Einträge zu einer Struktur dienen die Module `EntryDiscoverer` und `EntryMerger`. Beide nutzen die `Entry`- und die `StructureTemplate`-Klasse aus der jeweils gleichnamigen Datei, um Einträge oder Strukturen zu speichern.

Zum Gliedern einer Struktur in Einträge versucht der `EntryDiscoverer` zunächst mittels Autokorrelation Einträge mit fester Länge zu bestimmen. Wird der Grenzwert `AUTOCORRELATION_THRESHOLD` überschritten, werden Einträge fester Länge angenommen und erstellt. Wird der Grenzwert nicht erreicht, werden die Abstände zwischen den in dieser Struktur erkannten Dateinamen-Tokens genommen und überprüft, ob eine bestimmte Mindestanzahl (`VARIABLE_ENTY_SIZE_THRESHOLD`) dieser Werte gleich ist. Dieser Wert muss anschließend in die Abstände vor und nach dem Dateinamen eingeteilt werden, um aus den Dateinamen-Tokens Einträge zu erstellen. Dazu wird der erste Dateinamen-Tokens in der Struktur genommen und kontrolliert, ob dieser einen kleineren Abstand zum Blockanfang hat, als der vorher bestimmte Abstand zwischen den Tokens. Liegt hier ein kleinerer Abstand vor sind die Abstände vor und nach den Dateinamen-Tokens bekannt und die Einträge zu den Tokens können erstellt werden. Wird die Größe und Art der Einträge nicht über dieses Verfahren herausgefunden, müssen die Einträge manuell geprüft und eingegeben werden.

Nach dem Erkennen der Einträge sucht der `DependencyFieldDiscoverer` in diesen nach Längen und Offsetfeldern. Längenfelder sind Felder, die mit der Länge des Dateinamens zusammenhängen. Zum Erkennen dieser Felder werden Bytes an der gleichen Position in verschiedenen Einträgen betrachtet. Aus diesen Bytes stellt der `DependencyFieldDiscoverer` ein Gleichungssystem mit Gleichungen für jeden Eintrag auf. Dieses Gleichungssystem wird mit der `nsolve`-Funktion des `sympy`-Pakets gelöst. Die Lösung wird dahingehend überprüft, ob sie nur ganzzahlige Werte enthält. Ist dies der Fall, wurde hier ein Längenfeld erkannt.

Nach dem gleichen Verfahren werden Offsetfelder erkannt. Den Dateinameneinträgen werden dabei die entsprechenden Dateiinhalte über das gid-Attribut zugeordnet und dann über das Aufstellen der Gleichungssysteme geprüft.

Der `EntryMerger` leitet aus allen erstellten Einträgen eine gemeinsame Struktur ab. Hierzu werden die Tokens, die innerhalb der einzelnen Einträge liegen, genutzt. Ist keine Assoziation für einen Token vor-

handen, also die Bedeutung des Tokens unbekannt, wird ein einfaches Bytefeld angenommen. Ist an einer Position in den Einträgen kein Token vorhanden, wird ein Platzhalterfeld (Padding) eingefügt.

Die Vereinigung dieser Felder läuft nach mehreren Regeln ab. Liegen in allen Einträgen Tokens desselben Typs vor, wird dieser Typ für die zusammengeführte Struktur übernommen. Ist diese z. B. ein Datum in allen Tokens, wird auch für die zusammengeführte Struktur ein Datum eingefügt. Liegt für einige Einträge ein definiertes Feld wie z. B. ein Datum oder eine Zahl vor, während in den anderen Einträgen Byte- oder Platzhalterfelder an derselben Position existieren, wird das definierte Feld in die zusammengeführte Struktur übernommen. Existieren nur Byte- und Platzhalterfelder wird ein Bytefeld eingefügt. Vor allem bei Byte- und Platzhalterfeldern kann sich die Länge der Felder von der eines definierten Feldes unterscheiden. Liegt ein längeres Byte- oder Platzhalterfeld vor, wird dieses in mehrere Felder des jeweils gleichen Typs unterteilt.

Liegen mehrere definierte Einträge unterschiedlichen Typs vor, wird eine Unterstruktur erzeugt. Bei Einträgen fester Länge geht diese Unterstruktur so weit, bis die Felder aller Einträge wieder zusammengeführt werden können; bei Einträgen variabler Länge geht die Unterstruktur bis zum Ende aller Einträge. Um möglichst wenige Zweige der Unterstruktur zu erstellen, wird versucht die Einträge in der Unterstruktur jeweils paarweise zu einer Struktur zusammenzuführen. Dazu wird der im letzten Absatz beschriebene Algorithmus verwendet.

Nach dem Erstellen dieser Zweige wird versucht die Anzahl der Zweige über einige Regeln zu minimieren. Zweige, die nur Byte- oder Platzhalterfelder enthalten, werden verworfen. Außerdem werden Zweige zusammengeführt, welche Felder mit Zeichenketten und Füllzeichen enthalten, die sich zwar in der Länge der Zeichenketten unterscheiden, aber sonst nur Paddingfelder enthalten.

Nach all diesen Schritten entsteht eine Struktur, die alle gegebenen Einträge widerspiegelt. Bei ausreichend verschiedenen Einträgen kann diese Struktur dann genutzt werden, um beliebige Einträge zu parsen.

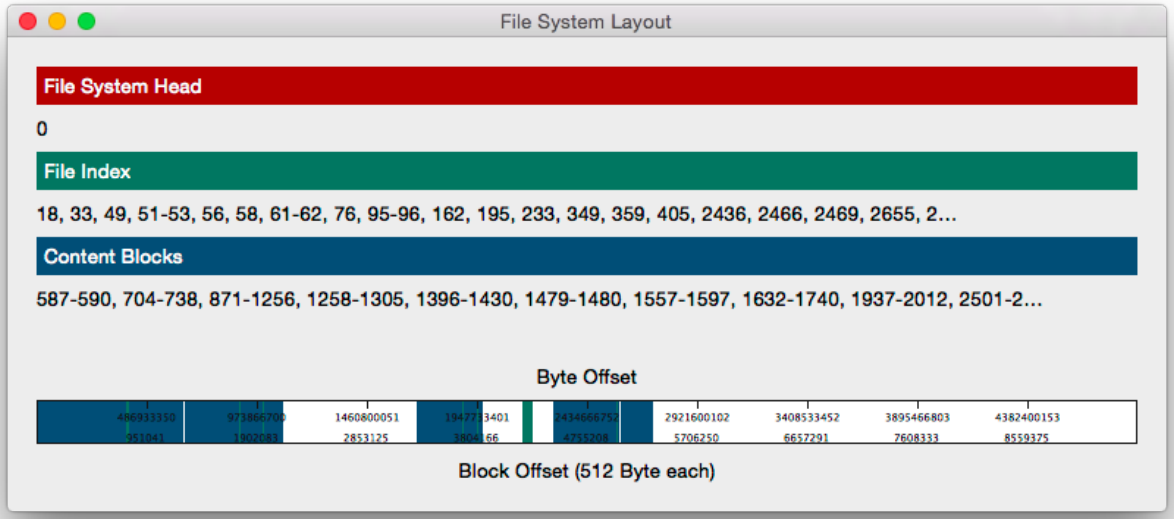
7.6 Ausgabe

Die Module im `reporting`-Paket sind sehr unterschiedlich. Der `FileListerGenerator` liest die in der Verfeinerung erstellten Strukturen ein und fügt diese in die Datei `FileListerTemplate.py.txt` ein. Durch durch dieses Einfügen entsteht ein Programm, welches genutzt werden kann, um Abbilder des Dateisystems auszulesen. Eingabe ist dafür das Abbild des Dateisystems und als Ausgabe werden die aus diesem Dateisystem vorhandenen Dateien angezeigt. Wurden benötigte Komponenten für dieses Programm in den vorherigen Phasen nicht erkannt, müssen diese manuell ergänzt werden. Daneben kann das Programm auch durch Auslesen der Allokationsdatei die Belegung von Blöcken anzeigen.

Der `ReportCreator` erstellt einen Report über alle Phasen des Prozesses. Hier werden Informationen zu Blockgröße, Tokens, Blöcken, Dateisystemstrukturen, Aufbau der Strukturen und Ergebnisse des erstellten Parsers eingearbeitet. Dieser Report kann zum manuellen Auslesen des Dateisystems oder zur weiteren Analyse des Dateisystems genutzt werden.

Das letzte Modul im `reporting`-Paket ist das `DiskLayoutWidget`. Dieses erstellt auf Grundlage der zugeordneten Blöcke im Dateisystem eine Visualisierung des Dateisystems. Für diese Visualisierung wird ein einfaches `PyQt5`-Fenster erstellt und angezeigt. Abbildung 7.4 zeigt das Fenster für ein untersuchtes Dateisystem in einem *Image-Only*-Szenario. Die Allokationsdatei und ein Journal wurden hier nicht erkannt.

Abbildung 7.4: Screenshot der Visualisierung



7.7 Include-Ordner

Im `inc`-Ordner ist Code gespeichert, der in mehreren Phasen genutzt wird. Dazu gehören Konstanten, globale Variablen, verwendete Datentypen, diverse Funktionen und der Code zum Variieren von Werten.

In dem Modul `const` werden alle im Programm genutzten Konstanten gespeichert. Diese Konstanten können angepasst werden, um die Ergebnisse des Programms zu verbessern, falls z. B. bestimmte Tokens oder Strukturen nicht erkannt werden. Das Modul `inc/phase` dient als abstrakte Superklasse für die einzelnen Phasen. Hier werden Logging und Zeitmessungen für alle Phasen initialisiert.

7.7.1 Datentypen

Im `inc/datatypes`-Ordner sind die von allen Programmteilen genutzten Datentypen als Klassen aufgelistet. Die wichtigsten dieser Datentypen sind Token, Assoziation und Attribut, welche alle im `Token`-Modul definiert sind. Zur Speicherung der Tokens existiert das `TokenStore`-Modul, welches in Abschnitt 7.3 ausführlich beschrieben ist. Daneben gibt es noch das Modul `FileSystemModel`, welche als Aufzählungstyp die Datenkategorien von Carrier (s. Abschnitt 2) und die verschiedenen Strukturen (s. Abschnitt 6.3) enthält.

Das `Token`-Modul implementiert die einzelnen Tokens. Diese sind, wie in Abschnitt 6.2 beschrieben, über Position des Blocks, Offset im Block sowie Länge definiert. Zum einfacheren Verarbeiten von Tokens ist außerdem das Ende des Tokens definiert, welches über Start und Länge der Tokens berechnet wird. Daneben wird der Ursprung der einzelnen Tokens gespeichert, um später die Bedeutung einzelner Tokens zu erkennen. Bei Tokens mit Assoziation wird daneben noch eine Assoziation zu den Tokens gespeichert, welche das zugeordnete Attribut sowie den Konfidenzwert der Zuordnung enthält. Außerdem sind hier die Kodierung zum Parsen des Wertes und der Quellcode zum Erzeugen der Kodierung gespeichert. Die Attribute enthalten Herkunft und Beschreibung des Attributes, den Wert des Attributes und dessen Datenkategorie. Außerdem enthalten die Attribute einen Wert, der die Zusammengehörigkeit von Attributen beschreibt. So enthalten z. B. Attribute zu Dateiname und Dateigröße derselben Datei denselben Wert. Zuletzt gibt es noch einen Wert, der angibt, ob das Attribut gefunden wurde.

7.7.2 Helferfunktionen

Unter `inc/helper` sind Module vorhanden, welche allgemeine Funktionen für die verschiedenen Phasen zur Verfügung stellen. Sowohl `VolumeCreator` als auch `DfxmlParser` werden in den ersten beiden Phasen genutzt. Das `VolumeCreator`-Modul erstellt mittels externer Kommandos Volumes und formatiert diese. Hierfür ist für jedes Betriebssystem jeweils eigener Code nötig, da die Erstellung von Volumes unter verschiedenen Betriebssystemen unterschiedlich behandelt wird. Der `DfxmlParser` liest DFXML-Dokumente ein und parst diese für den weiteren Gebrauch. Im `formated_logging`-Modul wird das Logging des Programms angepasst.

8 Evaluation

Nach dem Aufstellen des Konzeptes und der anschließenden Implementierung folgt nun ihre Evaluation. Hierbei wird geprüft, in welchem Maße die Implementierung und somit das Konzept genutzt werden kann den Aufbau eines Dateisystems herauszufinden und somit die zentrale Problemstellung zu lösen. Die Evaluation ist eingeteilt in die einzelnen Anforderungen an das Dateisystem-Reverse-Engineering: Qualität, Allgemeingültigkeit, Automatisierung und Effizienz (s. Kapitel 5). Alle vier Anforderungen werden in den nächsten Abschnitten einzeln evaluiert. Für diese Evaluation wurden sechs Testfälle definiert, für die das Programm jeweils ausgeführt und die Ergebnisse dann überprüft wurden. Tabelle 8.1 bietet eine Übersicht über die durchgeführten Tests.

Tabelle 8.1: Übersicht über die Testfälle

Testname	Dateisystem	Testtyp	Betriebssystem	Größe des Volumes/Abbildes
ExFAT-Volume	ExFAT	Volume	OS X	42 MB
HFS+-Volume	HFS+	Volume	OS X	42 MB
NTFS-Volume	NTFS	Volume	Windows	1024 MB
FAT32-Volume	FAT32	Volume	Windows	1024 MB
ReFS-Volume	ReFS	Volume	Windows	1024 MB
NTFS-Abbild	NTFS	Abbild	OS X	4871 MB

Die Testfälle untersuchen mit ExFAT, HFS+ und NTFS drei aktuell genutzte Dateisysteme unter Windows und OS X. Daneben wird das ältere, aber noch verbreitete FAT32 untersucht. Außerdem wird die Implementierung an ReFS evaluiert, welches als neues Dateisystem unter Windows mit Windows Server 2008 eingeführt wurde. Die meisten durchgeführten Testfälle arbeiten auf Volumes, es wird daneben aber auch ein Abbild untersucht. Dieses Abbild ist das Abbild eines 4,5 GB großen NTFS und stammt aus einer Übung des amerikanischen National Institute of Standards and Technology[1]. Anforderung an das Abbild war ein größeres Dateisystem, welches einer normalen Nutzung eines Computers entspricht.

Die Tests unter Windows wurden auf einer virtuellen Maschine mit Windows 2012 Server R2 mit 8 GB RAM durchgeführt. Die Volume-Tests laufen hierbei in einer 1024-MB-RAM-Disk, bei der ein Volume im RAM erzeugt und bereitgestellt wird. Hierdurch werden höhere Geschwindigkeiten für Zugriffe auf das Volume erreicht, als wenn dieses auf einer Festplatte erstellt würde. Die RAM-Disk wurde mit dem Programm StarWind erzeugt[2]. Der Host für die virtuelle Maschine ist ein Windows-7-System mit Intel

[1] http://www.cfreds.nist.gov/Hacking_Case.html
[2] https://www.starwindsoftware.com/

i5-3570 CPU mit 3.4 GHz und 32 GB RAM. Das Ergebnis des Tests mit dem NTFS-Abbild ist unabhängig vom Betriebssystem, nur die Laufzeit variiert hier ja nach System. Die Tests für OS X wurden auf einem OS-X-10.10.5-System mit 1.7 GHz i5, 4 GB RAM und 128 GB Samsung-SSD-Festplatte durchgeführt. Hier liefen die Tests nicht von einer RAM-Disk, sondern direkt von der Festplatte aus.

8.1 Qualität

Die wichtigste Anforderung an das Konzept ist die Qualität der erzeugten Ergebnisse. Die erstellten Modelle der einzelnen Dateisysteme sollen möglichst weit mit den Spezifikationen der Dateisysteme übereinstimmen. Für eine Evaluation der erstellten Modelle bietet das Sleuth Kit Unterstützung für alle untersuchten Dateisysteme, abgesehen von ReFS. Die Ergebnisse zu diesen Dateisysteme können so auf Korrektheit überprüft werden. Bei ReFS ist mehr manueller Aufwand nötig, um die Ergebnisse zu evaluieren, da hier noch keine vollständige Dokumentation des Dateisystems vorliegt.

Die Evaluation der sieben Testfälle wird, eingeteilt in die einzelnen Phasen, in den nächsten Abschnitten beschrieben.

8.1.1 Evaluation der Vorbereitung

In der Vorbereitung wird in den Dateisystemen die Blockgröße ausgelesen. Für die Evaluation wurde dieser Wert dann mit dem von `fsstat` ermittelten Wert verglichen. `fsstat` ist ein Kommandozeilenprogramm aus dem Sleuth Kit, welches grundlegende Parameter eines Dateisystems, wie z. B. die Blockgröße, darstellt. Da für ReFS das Sleuth Kit noch keine Unterstützung bietet, wurde hier die Dokumentation von Metz [23] hinzugenommen, welche die nötigen Parameter zum Berechnen der Blockgröße beschreibt. Tabelle 8.2 zeigt die von der Implementierung ermittelten Blockgrößen, sowie die tatsächlichen Blockgrößen der Dateisysteme. Beide Größen stimmen hier bei allen Tests überein.

Tabelle 8.2: Vergleich von ermittelten und tatsächlichen Blockgrößen

Test	Ermittelte Blockgröße	Tatsächliche Blockgröße
ExFAT-Volume	4096	4096
HFS+-Volume	4096	4096
NTFS-Volume	4096	4096
FAT32-Volume	4096	4096
ReFS-Volume	65536	65536
NTFS-Abbild	512	512

8.1.2 Evaluation der Tokenisierung und Assoziation

In der Tokenisierung ist eine externe Evaluierung gegen andere Programme oder manuell nur schwer möglich, da man hierfür ein Dateisystem in einzelne Tokens zerlegen müsste, welche dann verglichen werden müssten. Hierfür ist momentan kein Programm erhältlich und da die Anzahl der Tokens hierbei teilweise sehr hoch ist (mehr als 20.000 Tokens im NTFS-Volume-Test), ist eine manuelle Überprüfung praktisch nur mit sehr hohem Aufwand durchführbar.

Das Programm evaluiert für untersuchte Volumes jedoch einige Maße intern, welche Auskunft über den Erfolg der Tokenisierung geben. In der Tokenisierung und Assoziation werden sieben Operationen auf dem Dateisystem durchgeführt. Diese Operationen orientieren sich an den durchgeführten Operationen in anderen Reverse-Engineering-Verfahren (s. Abschnitt 4.2). Bei jeder Operation werden dabei einige der genutzten Parameter gesucht. Daneben werden noch Attribute des Dateisystems gesucht. In der folgenden Liste werden diese kurz beschrieben.

Attr Name des Dateisystems, des Volumes sowie der Volume- oder Partitions-UUID.

Par1 Dateiname und Zeitstempel beim Erstellen einer leeren Datei mit dem Namen „alpha“.

Par2 Dateiname und Zeitstempel beim Erstellen einer leeren Datei mit dem Namen „omega“.

Par3 Dateiname, Zeitstempel und Inhalt beim Erstellen einer Datei mit dem Namen „beta.txt“ und 13.312 Bytes ASCII Text.

Par4 Dateiname, Zeitstempel und Inhalt beim Erstellen einer Datei mit dem Namen „gamma.txt“ und zufälligem Inhalt in 100-facher Blockgröße.

Par5 Dateiname, Zeitstempel und Inhalt beim Erstellen einer Datei mit dem Namen „delta.txt“ und 2.000 Bytes `0xAA` als Inhalt.

Par6 Dateiname und Zeitstempel beim Erstellen eines Ordners mit dem Namen „epsilon“.

Par7 Dateinamen und Zeitstempel beim Erstellen einer leeren Datei im gerade erzeugten Ordner mit dem Namen „zeta.txt“.

Wie viele und vor allem welche dieser Attribute und Parameter gefunden werden, bestimmt inwieweit die Ergebnisse der Tokenisierung und Assoziation für die folgenden Phasen genutzt werden können. Tabelle 8.3 zeigt das Verhältnis von gesuchten zu gefundenen Attribute der Operationen und des Dateisystems. Dass die Anzahl der gesuchten Attribute bei Dateisystemen (Attr) hier variiert, liegt daran, dass bei verschiedenen Betriebssystemen unterschiedliche Attribute gesucht werden können. Die Daten des Dateisystems (Attr) wurden bei allen Tests unter Windows gefunden. Unter OS X wurden einige Attribute nicht gefunden oder lagen nicht im Dateisystem vor. In den weiteren Tests wurden unter OS X (ExFAT und HFS+) alle Dateinamen und Zeitstempel der Dateien gefunden. Im FAT32-Test wurden bei manchen Operationen Dateinamen nicht gefunden, da bei FAT32 Dateinamen und Dateiendungen getrennt gespeichert werden. Unter NTFS und ReFS wurden alle Dateinamen gefunden, die Zeitstempel der Dateien jedoch nicht erkannt.

Neben diesen gefundenen Assoziationen werden überschriebene Assoziationen erneut gesucht (s. Abschnitt 6.2.8). Tabelle 8.4 zeigt die wiederholt gesuchten und gefundenen Assoziationen. ExFAT und

Tabelle 8.3: Relation von gefundenen und gesuchten Attributen und Parametern

Test	Attr	Par1	Par2	Par3	Par4	Par5	Par6	Par7
ExFAT-Volume	12/13	4/4	4/4	4/4	4/4	4/4	4/4	4/4
HFS+-Volume	1/9	4/4	4/4	4/4	4/4	4/4	4/4	4/4
NTFS-Volume	2/2	1/4	1/4	1/4	1/4	1/4	1/4	1/4
FAT32-Volume	2/2	4/4	4/4	3/4	3/4	3/4	4/4	3/4
ReFS-Volume	2/2	1/4	1/4	1/4	1/4	1/4	1/4	1/4

ReFS haben keine gefundenen Assoziationen überschrieben, während bei NTFS und HFS+ viele Assoziationen erneut gesucht werden mussten. Alle erneut gesuchten Assoziationen wurden gefunden.

Tabelle 8.4: Relation von erneut gesuchten und gefundenen Assoziationen

Test	Par2	Par3	Par4	Par5	Par6	Par7
ExFAT-Volume	-	-	-	-	-	-
HFS+-Volume	3/3	12/12	7/7	10/10	11/11	12/12
NTFS-Volume	2/2	6/6	7/7	6/6	6/6	2/2
FAT32-Volume	-	4/4	-	-	-	-
ReFS-Volume	-	-	-	-	-	-

Neben Parametern und Attributen wird in der Tokenisierung und Assoziation außerdem bei einigen Operationen nach erstellten Dateiinhalten gesucht. Die Relation der gesuchten und gefundenen Fragmente von Dateiinhalten findet sich in Tabelle 8.5 wieder. Alle gesuchten Dateiinhalte wurden gefunden. Die unterschiedliche Anzahl an zu suchenden Dateifragmenten ergibt sich durch die unterschiedlichen Blockgrößen der Dateisysteme, sodass hier die Inhalte in unterschiedliche viele Fragmente aufgeteilt werden. Die gleiche Anzahl an Dateifragmenten bei Par4 ergibt sich dadurch, dass die Dateiinhaltsgröße von der Blockgröße abhängig gemacht wurde.

Tabelle 8.5: Relation von gesuchten und gefundenen Dateiinhalten

Test	Attr	Par3	Par4	Par5
ExFAT-Volume	2/2	4/4	100/100	1/1
HFS+-Volume	1/1	4/4	100/100	1/1
NTFS-Volume	-	4/4	100/100	1/1
FAT32-Volume	-	4/4	100/100	1/1
ReFS-Volume	-	1/1	100/100	1/1

Der Test des NTFS-Abbildes hängt ab von den Ergebnissen des genutzten File-Carving-Programms und der Suche nach Zeichenketten. Das National Institute of Standards and Technology hat in zwei Berichten [26,27] die Qualität der Ergebnisse des `photorec`-File-Carving-Tools für Bild- und Videodateien getestet. Beide Berichte bestätigen die Qualität des File-Carvers, solange die zu suchenden Dateien an den Sektorgrenzen ausgerichtet sind. Nicht untersucht wird die Qualität für das Auffinden anderer Dateitypen. Neben den Dateien und damit Dateifragmenten werden durch das Dateinamen-Carving insgesamt 113.047 Zeichenketten als Dateinamentokens erkannt. Viele dieser Funde liegen auch in nicht erkannten Dateifragmenten und sind daher keine einzelne Tokens.

8.1.3 Evaluation der Strukturerkennung

Wie in Kapitel 6.3 beschrieben, werden zu allen Strukturen alle Blöcke, in denen diese Struktur möglicherweise existiert, mit einem Konfidenzwert ausgegeben. Diese werden in den folgenden Tabellen als „gefundene Blöcke“ angegeben. Die Prozentzahlen bei den gefundenen Blöcken sind der Konfidenzwert für die jeweils dahinter gelisteten Positionen der Blöcke. Daneben werden für jede Struktur aus diesen gefundenen Blöcken diejenigen Positionen von Blöcken festgelegt, in denen die Struktur vorhanden ist („Gewählte Blöcke“ in den Tabellen). Neben diesen beiden Gruppen von Blöcken werden die Positionen der korrekten Blöcke für die verschiedenen Strukturen angegeben. Diese wurden mittels `fsstat` und `istat` aus dem Sleuth Kit sowie dem in iBored (s. Kapitel 3) enthaltenen Templates ermittelt.

Aus den gewählten Blöcken und den korrekten Blöcken lassen sich Precision und Recall berechnen. Formel 8.1 beschreibt die Berechnung der Precision, welche den Anteil der korrekt gewählten Blöcke an allen gewählten Blöcken angibt. Die Precision lässt sich nicht berechnen, wenn keine Blöcke gewählt wurden.

$$\text{Precision} = \frac{\text{Korrekt gewählte Blöcke}}{\text{Gewählte Blöcke}} \tag{8.1}$$

Formel 8.2 beschreibt die Berechnung des Recalls. Der Recall ist definiert als Anteil der korrekt gewählten Blöcke an allen gewählten Blöcken. Der Recall lässt sich nicht berechnen, wenn es keine korrekten Blöcke gibt.

$$\text{Recall} = \frac{\text{Korrekt gewählte Blöcke}}{\text{Korrekte Blöcke}} \tag{8.2}$$

Für den weiteren automatischen Reverse-Engineering-Prozess ist vor allem eine hohe Precision wichtig. Falsch gewählte Blöcke führen zu Fehlern im den weiteren Phasen des Prozesses. Ein geringer Recall und somit nicht erkannte Blöcke, sind weniger kritisch für die weiteren Phasen. Eine geringe Mindestanzahl an korrekt erkannten Blöcken ist ausreichend.

Die Lage der Strukturen wird oft bei der Erstellung des Dateisystems festgelegt, kann sich aber im Nachhinein noch ändern. Auch werden zu Beginn oft mehr Blöcke belegt, als benötigt werden. Bei den durchgeführten Test wurden sehr wenige Daten in die Dateisysteme geschrieben, um schnelle Laufzeiten zu

erhalten, sodass oft nur nur die ersten Blöcke der Strukturen erkannt wurden. Die folgenden Blöcke sind in manchen Dateisystemen auch leer und werden erst bei der Belegung formatiert.

Tabelle 8.6 zeigt die Positionen der Blöcke, die als Dateisystem-Header erkannt wurden. Da hier alle Dateisysteme den ersten Block des Dateisystems nutzen, welches auch ein Kriterium in der Untersuchung ist, wurden alle Blöcke richtig gewählt. Weitere gefundene Blöcke (1024 bei FAT32, 35 bei ExFAT, 87380 bei NTFS) enthalten den Namen des Volumes, sind aber Dateiindexstrukturen. Der $Boot-Bereich bei NTFS belegt zwei Blöcke, wobei hier nur ein Block erkannt wurde.

Tabelle 8.6: Blöcke des Dateisystem-Header

Test	Gefundene Blöcke	Gewählte Blöcke	Korrekte Blöcke	Precision	Recall
ExFAT-Volume	75%: 0 25%: 1, 35	0	0	1,00	1,00
HFS+-Volume	50%: 0 25%: 941 - 942	0	0	1,00	1,00
NTFS-Volume	75%: 0 25%: 2, 83722, 87380	0	0, 1	1,00	0,50
FAT32-Volume	75%: 0 25%: 1024	0	0	1,00	1,00
ReFS-Volume	75%: 0 25%: 32, 15359	0	0	1,00	1,00
NTFS-Abbild	50%: 0	0	0	1,00	1,00

In den verschiedenen Tests wurden Allokationsdateien gesucht. Tabelle 8.7 zeigt die dafür vermuteten Blöcke. Unter FAT32 wurden die File Allocation Tables korrekt als Allokationsdateien erkannt. Unter ExFAT wurde zusätzlich zu den File Allocation Tables die Bitmap erkannt, welche als eigentliche Allokationsdatei genutzt wird, und nur diese wurde ausgewählt. In den HFS+- und NTFS-Volumes wurde jeweils korrekt die Allokationsdatei gewählt. Im NTFS-Abbild wurde die Allokationsdatei aus Performanzgründen nicht gesucht und daher auch nicht erkannt. Unter ReFS existiert keine typische Allokationsdatei mehr. Hier werden Informationen zur Belegung von Blöcken in der Metadata-Struktur gespeichert [2]. Diese Metadata-Struktur dient zur Speicherung aller Dateisystemdaten, getrennte Strukturen dafür existieren nicht. Die Metadata-Struktur ist jedoch eingeteilt in einzelne Blöcke, die jeweils verschiedene Zwecke erfüllen. Block 11, der als Allokationsdatei erkannt wurde, speichert unter anderem auch die Belegung des Dateisystems, was hier erkannt wurde.

Die Blöcke, in denen ein Dateiindex vermutet wird, werden in Tabelle 8.8 aufgelistet. Unter ExFAT und FAT32 werden hier jeweils die Wurzelverzeichnisse korrekt erkannt. Unter NTFS werden sowohl das Wurzelverzeichnis als auch Teile der MFT richtig gewählt. Unter HFS+ ist der Dateiindex als Baum organisiert. Hier ist der erste Knoten des Baums in Block 939, es wird jedoch Block 940 erkannt, welcher

Tabelle 8.7: Blöcke der Allokationsdatei

Test	Gefundene Blöcke	Gewählte Blöcke	Korrekte Blöcke	Precision	Recall
ExFAT-Volume	57%: 32 56%: 16 26%: 44 - 45, 144 - 145	32	32	1,00	1,00
HFS+-Volume	66%: 1 26%: 941	1	1	1,00	1,00
NTFS-Volume	26%: 87370, 87389	87370	87370 - 87377	1,00	1,00
FAT32-Volume	26%: 0, 514, 769	514, 769	514 - 768, 769 - 1023	1,00	1,00
ReFS-Volume	26%: 11	11	-	0,00	-
NTFS-Abbild	-	-	4755295 - 4757616	-	0,00

den ersten Knoten mit Inhalt darstellt. Bei NTFS und HFS+ wurde der Start der jeweiligen Dateiindex-Struktur nicht erkannt. Ein Problem ist, dass hierdurch auch keine Verweise auf die Struktur gefunden werden können. ReFS nutzt, wie im letzten Absatz beschrieben, eine Struktur zum Speichern aller Informationen des Dateisystems und hat daher keinen eigenen Dateiindex. Die gewählten Blöcke enthalten jedoch Verweise auf die Dateien.

Im NTFS-Abbild werden 54.631 Blöcke als Dateiindex erkannt, wovon nur 160 Blöcke Teil der NTFS MFT sind. Da bei Abbildern keine Aktionen durchgeführt werden, werden Dateiindices nur durch das Dateinamen-Carving erkannt. Hierbei entstehen viele false-positive-Funde, welche dann auch zu falsch erkannten Funden bei Dateiindices führen.

Tabelle 8.9 listet erkannte Journal-Blöcke auf. Bei FAT32, ExFAT und dem untersuchen HFS+-Volume gibt es kein Journal. Dies wird bei dem FAT32 und HFS+ korrekt erkannt, bei ExFAT werden falsche Blöcke gewählt. Im NTFS-Volume werden neben den korrekten Blöcken des Journals auch fälschlicherweise die MFT-Kopie ($MFTMirr, Block 2) und ein Block der MFT (Block 87380) gewählt. Beim NTFS-Abbild wird kein Journal erkannt, obwohl hier eines vorhanden ist. Auch ReFS besitzt kein Journal, stattdessen wird hier ein copy-on-write-Ansatz gewählt [35]. Bei diesem Ansatz werden Inhalte einer Datei nicht verändert, sondern an anderer Stelle neu geschrieben und anschließend die Verweise zur Datei auf diese neue Position geändert. Das Ändern der Verweise ist dabei eine atomare Aktion, bei der keine Unterbrechung geschehen. Im Gegensatz dazu dauern Änderungen von Dateien in Dateisystemen mit Journal länger. Das Journal hat somit in copy-on-write-Dateisystemen keinen Nutzen mehr.

Im Gegensatz zu den anderen Strukturen ist die Position von Dateiinhalten in Dateisystemen nicht vorgegeben. Dateiinhalte werden an alle freien Stellen des Dateisystems geschrieben. Die neu erstellten und

Tabelle 8.8: Blöcke des Dateiindex

Test	Gefundene Blöcke	Gewählte Blöcke	Korrekte Blöcke	Precision	Recall
ExFAT-Volume	45%: 35, 38, 145	35, 38, 145	35, 38	0,67	1,00
HFS+-Volume	45%: 940 - 941 40%: 942	940 - 941	939 - 1018	1,00	0,03
NTFS-Volume	45%: 44, 83724 - 83725, 83740, 83769 - 83773, 87388 - 87390 40%: 87386 - 87387	44, 83724 - 83725, 83740, 83769 - 83773, 87386 - 87390	44, 87380 - 87443	0,38	0,08
FAT32-Volume	45%: 1024	1024	1024	1,00	1,00
ReFS-Volume	45%: 36, 141 40%: 37	36 - 37, 141	-	0,00	-
NTFS-Abbild	54.631 Blöcke (s. Text)	54.631 Blöcke (s. Text)	2097152 - 2104519, 2106016 - 2123258	< 0,01	0,02

Tabelle 8.9: Blöcke des Journal

Test	Gefundene Blöcke	Gewählte Blöcke	Korrekte Blöcke	Precision	Recall
ExFAT-Volume	60%: 1, 16, 44 - 45, 144	1, 16	-	0,00	-
HFS+-Volume	60%: 942	-	-	-	-
NTFS-Volume	60%: 2, 83722, 87380	2, 83722, 87380	83722 - 85545	0,67	< 0,01
FAT32-Volume	-	-	-	-	-
ReFS-Volume	60%:32, 15359	32, 15359	-	0,00	-
NTFS-Abbild	-	-	2045456 - 2097135	-	0,00

vorhandenen Dateiinhalte wurden in der Tokenisierung und Assoziation vollständig erkannt (s. Tabelle 8.5). Hierdurch sind in allen Dateisystemen alle Blöcke mit Dateiinhalten korrekt gewählt worden, wie auch Tabelle 8.10 zeigt. Es wurden alle gefundenen Dateiinhalte korrekt erkannt, es kann jedoch daneben noch weitere, nicht erkannte Dateiinhalte geben. Der Recall-Wert ist daher für diese Tabelle nicht angegeben.

Beim NTFS-Abbild werden 2.332.715 Blöcke als Fragmente von Dateien erkannt. Hierbei hängt die Korrektheit dieser Blöcke, wie in der Evaluation der Tokenisierung, von der Genauigkeit des File-Carving-Tools ab.

Tabelle 8.10: Blöcke mit Dateiinhalten

Test	Gewählte Blöcke	Davon korrekte Blöcke	Precision
ExFAT-Volume	37, 39-144	37, 39-144	1,00
HFS+-Volume	1808, 1813 - 1816, 1819-1918, 1921	1808, 1813 - 1816, 1819 - 1918, 1921	1,00
NTFS-Volume	36 - 40, 86141 - 86240	36 - 40, 86141 - 86240	1,00
FAT32-Volume	1025 - 1129	1025 - 1129	1,00
ReFS-Volume	39 - 140	39 - 140	1,00
NTFS-Abbild	2.332.715 Blöcke (s. Text)	? (s. Text)	-

8.1.4 Evaluation der Verfeinerung

Zur Evaluation der Verfeinerung können die Ergebnisse der drei Verfeinerungsmodule getrennt betrachtet werden.

Die Verfeinerung des Dateiindex besteht aus mehreren Schritten. Im ersten Schritt wird der Dateiindex in Einträge eingeteilt. Bei FAT32, ExFAT, NTFS und ReFS wird hier die richtige Einteilung getroffen. All diese Dateisysteme haben gemeinsam, dass Einträge hier in fester Länge vorliegen. Bei HFS+ wird erkannt, dass keine Einträge mit fester Länge vorliegen, die Berechnung der Dateiabstände ist jedoch falsch. Tabelle 8.11 zeigt die jeweils erkannten Arten und Längen von Datei-Einträgen und die jeweils korrekten Arten und Längen von Datei-Einträgen. Bei Dateieinträgen mit variabler Länge ist als Eintragslänge die Länge der Dateieinträge ohne die Länge des Dateinamens angegeben.

Der nächste Schritt in der Verfeinerung des Dateiindex ist die Erkennung von Längen- und Offsetfeldern. Hier wird lediglich das Längenfeld bei ExFAT erkannt. Da sowohl Längen- als auch Offsetfeld zur Erstellung eines Parsers benötigt werden, ist die automatische Generierung eines Parsers für alle Testfälle im Anschluss nicht vollständig möglich.

Die weitere Verfeinerung der Dateieinträge wird bei ExFAT und FAT am besten durchgeführt. Hier werden jeweils der Dateiname sowie drei Zeitstempel erkannt. Auch bei den MFT-Einträgen bei dem NTFS-Volume werden die Dateinamen erkannt, die Zusammenführung der Dateinamen funktioniert jedoch nur eingeschränkt. Bei HFS+, dem NTFS-Abbild sowie ReFS werden keine weiteren Strukturen innerhalb der Einträge erkannt.

Bei der Verfeinerung des Dateisystem-Headers werden Referenzen auf andere Strukturen gesucht. Hier liefert der Test von NTFS sowohl bei dem Volume als auch dem Abbild ein richtiges Ergebnis, bei dem der

Tabelle 8.11: Länge von Datei-Einträgen

Test	Blöcke des Dateiindex	Erkannte Struktur	Korrekte Struktur	Erkannte Eintragslänge	Korrekte Eintragslänge
ExFAT-Volume	35	fest	fest	96	96
HFS+-Volume	941	variabel	variabel	256	192
NTFS-Volume	87389-87396	fest	fest	1024	1024
FAT32-Volume	1024	fest	fest	32	32
ReFS-Volume	36-37	fest	fest	16384	16384
NTFS-Abbild	2122820-2122827	fest	fest	1024	1024

Verweis auf die MFT gefunden wird. Verweise auf andere Strukturen werden in keinem der Dateisysteme gefunden.

Die Verfeinerung der Allokationsdatei besteht in der Annahme, dass alle zukünftigen Dateisysteme diese als Bitmap über einen kompletten Block aufbauen. Für ExFAT, NTFS, HFS+ trifft diese Aussage zu, bei dem älteren FAT32 und ReFS wird jedoch kein kompletter Block als Bitmap verwendet. Bei diesen Dateisystemen ist die Verfeinerung der Allokationsdatei als Bitmap falsch.

8.1.5 Evaluation der Ausgabe

In der Ausgabe wird ein Parser erstellt, welcher zum Auslesen von Dateien aus dem Dateisystem genutzt werden kann. Dieser Parser soll dabei beliebige Instanzen des Dateisystems auslesen können und nicht speziell an die Parameter des untersuchten Dateisystems gebunden sein.

Abschnitt 6.5.1 beschreibt die Funktion eines einfachen Parser. Die zur Ausführung des Parsers nötigen Informationen sind dabei:

- Blockgröße
- Position des Dateiindex
- Länge des Dateiindex
- Dateiname
- Dateigröße
- Position des Dateiinhalts

Diese Informationen müssen dabei zur Erstellung eines generischen Parsers nicht nur für eine Instanz des Dateisystems erkannt werden, sondern über ein Attribut ausgelesen werden. Tabelle 8.12 zeigt, welche der benötigten Informationen zum Erstellen des Parsers vorliegen. Fehlende Informationen bestehen oft aus mehreren Feldern in den Strukturen. So berechnet sich die Position des Dateiindex unter FAT32 aus drei verschiedenen Feldern im Dateisystem-Header. Rein automatisch kann aus den gefundenen

Informationen kein Parser erstellt werden. Die jeweils fehlenden Informationen erfordern die manuelle Nachbearbeitung des Parsers.

Tabelle 8.12: Erkannte Informationen für den Parser

Test	Blockgröße	Position des Dateiindex	Länge des Dateiindex	Dateiname	Dateigröße	Position des Dateiinhalts
ExFAT-Volume	✗	✗	✗	✓	✓	✗
HFS+-Volume	✗	✗	✗	✗	✗	✗
NTFS-Volume	✗	✓	✗	✓	✗	✗
FAT32-Volume	✗	✗	✗	✓	✗	✗
ReFS-Volume	✗	✗	✗	✗	✗	✗
NTFS-Abbild	✗	✗	✗	✗	✗	✗

8.1.6 ReFS

Das Resilient File System (ReFS) wurde 2012 von Microsoft vorgestellt. Es ist als Nachfolger für NTFS vorgesehen und bietet einige neue Funktionen. Treiber für das Dateisystem sind in Windows Server 2012 enthalten.

Der interne Aufbau des Dateisystems wird über eine übergeordnete Metadatenstruktur bestimmt, anstatt wie andere Dateisysteme mehrere Strukturen zu nutzen. Diese Metadatenstruktur ist in einzelne Metadatenblöcke eingeteilt [3,19]. Von diesen gibt es verschiedene Arten, welche unterschiedliche Funktionen erfüllen. So gibt es z. B. einen Master File Table Metadata Block, welcher die Funktion des Dateiindex einnimmt. Diese Blöcke sind wiederum in einen Header und mehrere Tabellen eingeteilt, welche sich nach Art des Metadatenblocks unterscheiden. Vollständig sind diese Tabellen bisher nicht definiert [18]. Die Standard-Blockgröße des Dateisystems ist 65.536 Byte. Die Metadatenblöcke haben eine Größe von 16.384 Byte.

Bei den durchgeführten Tests wurde sowohl die Blockgröße als auch die Unterteilung in 16.384 große Einträge erkannt. Daneben wurden Metadatenblöcke erkannt, die Informationen zur Belegung des Dateisystems oder Metadaten zu einzelnen Dateien enthalten. Eine innere Struktur mit den Tabellen und deren weitere Unterteilung wurde automatisiert nicht erkannt. Diese Informationen müssen im Anschluss manuell hinzugefügt werden.

8.2 Allgemeingültigkeit

Das übergeordnete fünfphasige Konzept beruht auf nur wenigen Annahmen und lässt sich auf alle bekannten Dateisysteme anwenden. Der innere Ablauf der einzelnen Phasen schließt jedoch schon mehr

einzelne Annahmen ein, die nicht für alle Dateisysteme gegeben sein müssen. So werden explizit verschiedene Strukturen behandelt, die so nicht bei ReFS existieren. Eine uneingeschränkte Allgemeingültigkeit der automatisierten Implementierung existiert daher nicht.

Durch das Testen der verschiedenen Dateisysteme sowie von Volumes und Abbildern soll eine möglichst breite Abdeckung von Dateisystemen gezeigt werden. Qualitative Fehler in den Ergebnissen (s. Abschnitt 8.1) beruhen in der Regel auf nicht erfüllten Annahmen.

8.3 Automatisierung

Das erstellte Konzept soll möglichst automatisiert ablaufen, es kann jedoch nach jeder Phase in den Prozess eingegriffen werden. Nach der Ausgabe muss bei allen Tests eingegriffen werden, um das erstellte Modell zu vervollständigen. Daneben war in den Tests ein manueller Eingriff nur nach der Strukturerkennung bei beiden NTFS-Tests sowie dem ReFS-Test notwendig. Bei allen anderen Testschritten konnten die Ausgaben der jeweils vorherigen Schritte direkt übernommen werden.

8.4 Effizienz

Neben der Qualität der Ergebnisse, welche in den letzten Abschnitten evaluiert wurde, ist die Effizienz des Programms ein entscheidender Faktor im Vergleich zu einem manuell durchgeführten Reverse-Engineering-Vorgang. Wie in Abschnitt 8 beschrieben wurden alle Tests auf handelsüblicher Hardware durchgeführt. Tabelle 8.13 zeigt die Laufzeit der einzelnen Phasen für die durchgeführten Tests. Diese Zeit ist abhängig vom ausführendem Computer und anderen parallel laufenden Programmen sowie dem Betriebssystem. Die schnellste Ausführung gab es beim ExFAT-Volume unter OS X. Dieses Volume war mit etwa 42 MB auch eines der kleinsten im Tests. Ein weiterer stark abweichender Wert ist die Dauer der Verfeinerung für ReFS. Diese ist bedingt durch die 16-fach höhere Blockgröße gegenüber den anderen Tests. Daneben ist die Verarbeitung des NTFS-Abbildes langsamer als die anderen Tests, da hier eine viel größere Menge an Daten verarbeitet wurde.

Tabelle 8.13: Laufzeiten der einzelnen Programme
gerundet auf Sekunden, * = Zeit inklusive manueller Eingabe

Test	Vorbereitung	Strukturerkennung			Ausgabe	
		Tokenisierung		Verfeinerung		Gesamt
ExFAT-Volume	11 s	38 s	< 1 s	4 s	*6 s	**59 s**
HFS+-Volume	6 s	54 s	< 1 s	11 s	*2 s	**1 min 13 s**
NTFS-Volume	4 s	285 s	10 s	2 s	*4 s	**5 min 5 s**
FAT32-Volume	4 s	122 s	< 1 s	15 s	*3 s	**2 min 24 s**
ReFS-Volume	3 s	114 s	< 1 s	1362 s	*6 s	**24 min 45 s**
NTFS-Abbild	198 s	348 s	111 s	239 s	*83 s	**16 min 19 s**

9 Fazit und Ausblick

Die vorliegende Arbeit erstellt ein Konzept zum automatisierten Reverse Engineering von Dateisystemen. Dieses ermöglicht es, aus unstrukturierten Daten in einem Abbild oder Volume ein Modell zu generieren, welches den Aufbau des Dateisystems beschreibt.

Im Gegensatz zu den bisherigen Verfahren im Dateisystem-Reverse-Engineering wird im aufgestellten Konzept eine klare Definition des Prozesses in fünf Phasen gegeben. Diese Gliederung ermöglicht eine Automatisierung der einzelnen Phasen, bietet aber dennoch die Möglichkeit, in den Prozess einzugreifen. Hierdurch wird im Gegensatz zu bisherigen manuellen Analysen die Effizienz des Dateisystem-Reverse-Engineering erhöht, ohne Genauigkeit einbüßen zu müssen. Wie die Evaluation gezeigt hat, funktioniert die Automatisierung des Konzeptes in den ersten Phasen des Prozesses sehr gut. Grundlegende Maße des Dateisystems und Positionen von Dateisystemstrukturen werden gut erkannt. Auch die Einteilung der Dateiindices in Einträge fester Länge funktioniert gut.

Probleme gibt es in der Erkennung von Beziehungen im Dateisystem, wie Verweisen auf andere Strukturen oder Inhalte von Dateien. Zur Umsetzung dieser Beziehung werden in den verschiedenen Dateisystemen zum Teil sehr unterschiedliche Ansätze gewählt, wodurch eine automatisierte Suche nach diesen Beziehungen erschwert wird und somit eine manuelle Interaktion notwendig wird. Abhilfe könnte hier das Erstellen vieler Abbilder desselben Dateisystems mit unterschiedlichen Rahmenbedingungen (z. B. Partitonsgröße, Sektorgröße, Dateisystemtreiber) und das Vergleichen dieser Abbilder schaffen.

Es gibt mit dem Image-Only-, welches eher der Dateisystem-Analyse zuzuordnen ist, und dem Chosen-Data-Szenario, welches zum Dateisystem-Reverse-Engineering gehört, zwei Ausgangsszenarien, welche sehr unterschiedliche Anforderungen an das Konzept stellen. Im vorgestellten Konzept wird dabei versucht für beide Szenarien einen Ablauf zu erstellen, was sich als schwierig erweist. Auch wenn beide Szenarien als Reverse Engineering bezeichnet werden, sollte hier zukünftig besser differenziert werden und in wissenschaftlichen Arbeiten die beiden Szenarien getrennt betrachtet werden.

Weitere Ansätze zum Fortführen der Arbeit gibt es vor allem in den Bereichen, die in der Evaluation weniger gute Ergebnisse hervorbrachten. Ein erweitertes Verfahren zum Erkennen und Zusammenführen von Einträgen variabler Länge, analog zur Autokorrelation bei Einträgen fester Länge, wäre hilfreich. Ebenso kann beim Image-Only-Szenario die Erkennung von Dateiindices verbessert werden. Hier könnten neben den Dateinamen z. B. noch andere Daten gesucht werden, um die Erkennung zu verbessern.

Neben der Verbesserung bei der Erkennung und der Verfeinerung von Strukturen ist auch eine größere Breite an Visualisierungen des Dateisystems möglich. Programme wie binvis[1] oder binglide[2] bieten dem Nutzer verschiedene, nicht textbasierte Darstellungen von Binärdaten und können so den Reverse-Engineering-Prozess unterstützen.

[1] http://binvis.io
[2] https://github.com/wapiflapi/binglide

Auch die Implementierung könnte dahingehend angepasst werden, dass ein Framework entsteht, welches dem Benutzer mehr Interaktionsmöglichkeiten bietet. Statt des statischen Ablaufes könnte ein Programm hilfreich sein, welches eine beliebige Verkettung der einzelnen Schritte zulässt und es dem Benutzer so ermöglicht eigene Pipelines aufzubauen. Ein solches Framework könnte dann auch zum Reverse Engineering anderer Datentypen genutzt werden, die keine Dateisysteme sind.

Literatur

[1] Aswami Ariffin, Jill Slay und Kim-Kwang Choo: “Data Recovery from Proprietary Formatted Cctv Hard Disks”. In: *Advances in Digital Forensics IX*. Hrsg. von Gilbert Peterson und Sujeet Shenoi. IFIP Advances in Information and Communication Technology 410. Springer Berlin Heidelberg, 2013, S. 213–223. ISBN: 978-3-642-41147-2 978-3-642-41148-9. URL: http://link.springer.com/chapter/10.1007/978-3-642-41148-9_15 (abgerufen am 06. 07. 2015).

[2] Willi Ballenthin: The Microsoft ReFS File System. 2014. URL: http://www.williballenthin.com/forensics/refs/index.html (abgerufen am 25. 08. 2015).

[3] Willi Ballenthin: The Microsoft ReFS On-Disk Layout. 2015. URL: http://www.williballenthin.com/forensics/refs/disk/index.html (abgerufen am 06. 12. 2015).

[4] Remy Card, Theodore Ts'o und Stephen Tweedie: “Design and implementation of the second extended filesystem”. In: *Proceedings of the first Dutch international symposium on Linux*. Bd. 1. 1994.

[5] Brian Carrier: “Defining digital forensic examination and analysis tools using abstraction layers”. In: *International Journal of digital evidence* 1.4 (2003), S. 1–12. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.9813&rep=rep1&type=pdf (abgerufen am 25. 08. 2015).

[6] Brian Carrier: File system forensic analysis. Bd. 3. Addison-Wesley Reading, 2005.

[7] Andrew Case: Introducing bstrings, a Better Strings utility! 7. Nov. 2015. URL: http://binaryforay.blogspot.de/2015/07/introducing-bstrings-better-strings.html (abgerufen am 25. 08. 2015).

[8] Craving for time? Carve some timestamps out... – TimeCraver v0.1. 22. Aug. 2015. URL: http://www.hexacorn.com/blog/2015/08/22/craving-for-time-carve-some-timestamps-out-timecraver-v0-1/ (abgerufen am 25. 08. 2015).

[9] Weidong Cui, Jayanthkumar Kannan und Helen J. Wang: “Discoverer: Automatic Protocol Reverse Engineering from Network Traces.” In: *USENIX Security*. 2007, S. 199–212. URL: http://static.usenix.org/event/sec07/tech/full_papers/cui/cui.pdf (abgerufen am 20. 03. 2015).

[10] Weidong Cui u. a.: “Tupni: Automatic reverse engineering of input formats”. In: *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, S. 391–402. URL: http://dl.acm.org/citation.cfm?id=1455820 (abgerufen am 20. 03. 2015).

[11] Oliver Diedrich: Android 2.3 mit Ext4-Dateisystem. heise open. 14. Dez. 2010. URL: http://www.heise.de/open/meldung/Android-2-3-mit-Ext4-Dateisystem-1152964.html (abgerufen am 20. 12. 2015).

[12] Faust: Der Tragödie erster Teil by Johann Wolfgang von Goethe - Freies Ebook. URL: http://www.gutenberg.org/ebooks/2229 (abgerufen am 25. 08. 2015).

[13] Niels Ferguson und Bruce Schneier: Practical cryptography. Bd. 23. Wiley New York, 2003.

[14] FILETIME structure (Windows). URL: https://msdn.microsoft.com/en-us/library/ms724284.aspx (abgerufen am 25. 08. 2015).

[15] Kathleen Fisher u. a.: “From dirt to shovels: fully automatic tool generation from ad hoc data”. In: *ACM SIGPLAN Notices*. Bd. 43. ACM, 2008, S. 421–434. URL: http://dl.acm.org/citation.cfm?id=1328488 (abgerufen am 28. 04. 2015).

[16] Simson Garfinkel: “Digital forensics XML and the DFXML toolset”. In: *Digital Investigation* 8.3 (2012), S. 161–174. URL: http://www.sciencedirect.com/science/article/pii/S1742287611000910 (abgerufen am 13. 08. 2015).

[17] Gartner Says Tablet Sales Continue to Be Slow in 2015. 5. Jan. 2015. URL: http://www.gartner.com/newsroom/id/2954317 (abgerufen am 25. 08. 2015).

[18] Paul K. Green: “Resilient File System - Paul Green.pdf”. Diss. 8. Nov. 2013.

[19] Andrew Head: Forensic Investigation of Microsoft’s Resilient File System (ReFS). 2015. URL: http://resilientfilesystem.co.uk/index (abgerufen am 25. 08. 2015).

[20] Christoph Hellwig: “Reverse engineering an advanced filesystem”. In: *Ottawa Linux Symposium*. 2002, S. 191. URL: https://www.kernel.org/doc/mirror/ols2002.pdf#page=191 (abgerufen am 13. 04. 2015).

[21] Market share for mobile, browsers, operating systems and search engines | NetMarketShare. URL: http://www.netmarketshare.com/ (abgerufen am 25. 08. 2015).

[22] Joachim Metz: Library and tools to access the Resilient File System (ReFS). GitHub. URL: https://github.com/libyal/libfsrefs (abgerufen am 25. 08. 2015).

[23] Joachim Metz: Resilient File System (ReFS) - Analysis of the Windows Resilient File System. Nov. 2013. URL: https://30ece49dd0252ef7e0b531d907551568e0fb0914.googledrive.com/host/0B3fBvzttpiiSSmwxUWxManBEMG8/Resilient%20File%20System%20(ReFS).pdf (abgerufen am 29. 09. 2015).

[24] Saul B. Needleman und Christian D. Wunsch: “A general method applicable to the search for similarities in the amino acid sequence of two proteins”. In: *Journal of molecular biology* 48.3 (1970), S. 443–453. URL: http://www.sciencedirect.com/science/article/pii/0022283670900574 (abgerufen am 15. 12. 2015).

[25] NTFS FAQ (en). URL: https://flatcap.org/linux-ntfs/info/ntfs.html#3.8 (abgerufen am 25. 08. 2015).

[26] Office of Law Enforcement Standards of the National Institute of Standards and Technology: Test Results for Graphic File Carving Tool: PhotoRec 7.0-WIP. 16. Juli 2014.

[27] Office of Law Enforcement Standards of the National Institute of Standards and Technology: Test Results for Video File Carving Tool: PhotoRec v7.0-WIP. 22. Okt. 2014.

[28] N. R. Poole, Q. Zhou und P. Abatis: “Analysis of CCTV digital video recorder hard disk storage system”. In: *Digital Investigation* 5.3 (März 2009), S. 85–92. ISSN: 1742-2876. DOI: 10.1016/j.diin.2008.11.001. URL: http://www.sciencedirect.com/science/article/pii/S1742287608000984 (abgerufen am 06. 07. 2015).

[29] Vijayshankar Raman und Joseph M. Hellerstein: “Potter’s wheel: An interactive data cleaning system”. In: *VLDB*. Bd. 1. 2001, S. 381–390. URL: http://www.vldb.org/conf/2001/P381.pdf (abgerufen am 28. 04. 2015).

[30] Recovering NTFS Boot Sector on NTFS Partitions. URL: https://support.microsoft.com/en-us/kb/153973 (abgerufen am 25. 08. 2015).

[31] Sven Schmitt, Michael Spreitzenbarth und Christian Zimmermann: “Reverse engineering of the Android file system (YAFFS2)”. In: *Friedrich-Alexander-Universität Eriangen-Nürnberg, Tech. Rep. CS-2011-06* (2011).

[32] Andreas Schuster: “Ad-hoc File System Forensics”. In: EU Digital Forensics and Incident Response Summit 2011. 2011.

[33] Claude Elwood Shannon: “A mathematical theory of communication”. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 5.1 (2001), S. 3–55. URL: http://dl.acm.org/citation.cfm?id=584093 (abgerufen am 26. 08. 2015).

[34] Robert Shullich: Reverse Engineering the Microsoft Extended FAT File System (exFAT). 1. Dez. 2009. URL: http://www.sans.org/reading-room/whitepapers/forensics/reverse-engineering-microsoft-exfat-file-system-33274.

[35] Steven Sinofsky: Building the next generation file system for Windows: ReFS - Building Windows 8 - Site Home - MSDN Blogs. 16. Jan. 2012. URL: http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx?Redirected=true (abgerufen am 20. 10. 2015).

[36] Lee Tobin, Ahmed Shosha und Pavel Gladyshev: “Reverse engineering a CCTV system, a case study”. In: *Digital Investigation*. Special Issue: Embedded Forensics 11.3 (Sep. 2014), S. 179–186. ISSN: 1742-2876. DOI: 10.1016/j.diin.2014.07.002. URL: http://www.sciencedirect.com/science/article/pii/S1742287614000917 (abgerufen am 06. 07. 2015).

[37] Christian Zimmermann: “Mobile Phone Forensics: Analysis of the Android Filesystem (YAFFS2)”. In: (30. Apr. 2011). URL: http://www1.cs.fau.de/filepool/thesis/diplomarbeit-2011-zimmermann.pdf (abgerufen am 08. 04. 2015).