h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

Hochschule Darmstadt

- Fachbereich Informatik -

`Analyse und Extraktion forensisch relevanter Informationen aus dem Ext4 Dateisystem Journal`

Abschlussarbeit zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt von
David Pelka

Referent:	Prof. Dr. Harald Baier
Korreferent:	Prof. Dr. Andreas Heinemann

Ausgabedatum:	01.10.2014
Abgabedatum:	01.04.2015

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 01. April 2015

David Pelka

Kurzfassung

Während es in anderen Dateisystemen möglich ist, gelöschte Dateien anhand der hinterlassenen Metadaten wiederherzustellen, werden alle dafür relevanten Informationen innerhalb des Ext4 Dateisystems überschrieben. Dies macht eine Wiederherstellung gelöschter Dateien ohne weitere Informationen nicht möglich. Als Standard-Dateisystem für moderne Linux-Distributionen und Android, einem weitverbreiteten Betriebssystem für mobile Geräte, ist Ext4 jedoch omnipräsent im Einsatz.

Werden im Rahmen einer Computerstraftat beweiskräftige Spuren vernichtet, kann eine Wiederherstellung gelöschter Daten ausschlaggebend für die Überführung des Täters sein. Aus diesem Grund beschäftigt sich die vorliegende Arbeit mit der Analyse des Ext4 Dateisystem Journals. Hierbei gilt es festzustellen, ob das Journal als Quelle für alte Informationen geeignet ist und inwiefern es möglich ist, gelöschte Dateien mit Hilfe extrahierter Journaldaten zu rekonstruieren.

Für diesen Zweck werden die notwendigen Grundlagen und die internen Strukturen des Ext4 Dateisystems detailliert beschrieben. Darüber hinaus wird das Journal auf seinen Aufbau und die Funktionalität untersucht, woraus im Laufe der Arbeit ein forensischer Nutzen abgeleitet wird. Auf Basis der gewonnenen Erkenntnisse wird ein Konzept erarbeitet, das die Rekonstruktion gelöschter Dateien ermöglicht. Zudem wird das Konzept in Form einer prototypischen Implementierung realisiert und der Mehrwert anhand der Wiederherstellung fragmentierter Dateien evaluiert.

Abstract

While in other file systems it is possible to reclaim deleted files in the remaining metadata, in Ext4 file systems all relevant information for this purpose is deleted. This makes a reconstruction of deleted files without further information impossible. As standard file system for modern Linux distributions and Android, a broadly implemented operating system for mobile devices, Ext4 is omnipresent in its use.

In the case of computer criminality, when evidence traces have been destroyed, a restoration of deleted data is crucial to convicting the offender. For this reason, the present work on the analysis of the Ext4 file system journal is taken to task. Hereby it will be to establish, if the Journal, as source for old information is suitable and, as far as possible, to reconstruct deleted files from extracted journal data.

To this end, the necessary basics and the structures of the Ext4 file system will be described in detail. Beyond this, the structure of the journal and its functionalities will be examined, thereby, in the course of this work, a forensic usefulness will be derived. Based on the gained knowledge, a concept will be developed to make possible the reconstruction of deleted files. In addition, the concept shall be implemented in the form of a prototype and the results based on the reconstruction of the fragmented files evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

KAPITEL 1

EINLEITUNG

Digitale Straftaten sowohl externer als auch interner Täter gegen Computersysteme machen beinahe täglich Schlagzeilen. Mit der umfangreichen Rolle, die IT-Systeme in unserer Gesellschaft eingenommen haben, wächst auch das Potenzial der Computerkriminalität. Um Straftaten entgegenzuwirken, müssen Spuren auf digitalen Geräten so gesichert und untersucht werden, dass sie als Beweismittel in einem Strafverfahren vor Gericht eingesetzt werden können. Mit dieser Thematik beschäftigt sich die digitale Forensik.

Ein wichtiger Bestandteil der digitalen Forensik ist die Datenanalyse, die sich mit der Untersuchung der sichergestellten Daten beschäftigt. Um möglichst alle Spuren auffinden zu können, wird dafür ein tiefgreifendes Know-how im Bereich der Betriebs- und Dateisysteme benötigt. Egal ob innerhalb eines Unternehmens die Spuren von Wirtschaftskriminalität vernichtet werden, ein versehentlich abgesetztes *„rm“* wichtige Dokumente löscht oder ein Straftäter vor dem Zugriff der Behörden relevante Beweismittel vernichtet: Die Rekonstruktion von gelöschten Dateien ist in jeder forensischen Untersuchung von Belang.

Diese Arbeit beschäftigt sich der *„Analyse und Extraktion forensisch relevanter Daten aus dem Ext4 Dateisystem Journal“* und untersucht das Journal auf potenzielle relevante Informationen, die aus dem Journal gewonnen werden können, die sonst nicht mehr im Dateisystem zu finden sind.

1.1. Motivation

Aktuelle Dateisysteme wie Ext4, NTFS oder HFS+ verwenden Journale, um Änderungen am System aufzuzeichnen. Mit Hilfe dieser ist es möglich jederzeit einen konsistenten Zustand des Dateisystems zu gewährleisten. Selbst bei einem Systemabsturz oder einem fehlgeschlagenen Schreibvorgang bleibt das Dateisystem in einem definierten Zustand.

Sämtliche Dateiänderungen werden zu diesem Zweck in das Journal geschrieben, bevor sie auf ihre endgültige Position auf dem Datenträger geschrieben werden. Dieser Vorgang wird als *Journaling* bezeichnet und geschieht vorwiegend im Hintergrund, sodass der Endanwender[1] davon in der Regel nichts mitbekommt. Das bedeutet, dass das Journal eine Art Logbuch über alle Vorgänge auf dem Dateisystem darstellt und langwierige Dateisystem-Tests nach fehlgeschlagenen Schreibvorgängen überflüssig macht.

In der klassischen Forensik gibt es das sogenannte *Locardsche Austauschprinzip*, das besagt, dass kein Kontakt zwischen Täter, Opfer und Tatort vollzogen werden kann, ohne dass diese einen gegenseitigen Austausch an Spuren hinterlassen (vgl. [Loc30]).

Übertragen auf die digitale Forensik bedeutet das, dass es in jedem komplexen digitalen System bei der Datenverarbeitung zu einer Hinterlassung von Spuren kommt (vgl. [Bai14]). Dies spiegelt sich in dem Verhalten eines Dateisystem Journals wieder. Die Eigenschaft, das Daten nach jeder Änderung in das Journal geschrieben werden, machen es aus forensischer Sicht zu einer potenziellen Quelle für alte Informationen, die so gegebenenfalls nicht mehr im Dateisystem zu finden sind.

1.2. Ziel der Arbeit

Diese Arbeit beschäftigt sich mit der aktuellen Version des *Extended Filesystems* Ext4 und untersucht hierbei, welche Rolle die vom Dateisystem verwendeten Journale bei einer forensischen Untersuchung spielen. Zu Beginn wird untersucht welche Journale auf dem Dateisystem existieren, wie mit diesen umgegangen wird und welche Informationen erfasst werden. Dabei gilt es relevante Informationen für eine forensische Untersuchung zu finden. Untersucht wird, welche Erkenntnisse mit Hilfe der Journale gewonnen und wie diese Informationen anschließend extrahiert werden können. Vom besonderen Interesse ist die Frage, ob Rückschlüsse auf gelöschte Informationen gezogen werden können.

[1] Aus Gründen der besseren Lesbarkeit wird in dieser Arbeit das generische Maskulinum verwendet.

Des Weiteren wird im Rahmen dieser Arbeit ein Tool implementiert, das den Umgang mit dem Journal erleichtert und eine Extraktion von Informationen beziehungsweise die Rekonstruktion von gelöschten Dateien automatisiert.

1.3. Aufbau der Arbeit

Die Arbeit ist in acht Kapitel unterteilt. Die Struktur setzt sich wie folgt zusammen:

KAPITEL 1: Dieses Kapitel gibt eine erste Einführung in die Thematik. Hierzu werden Motivation und Zielsetzung der Arbeit aufgezeigt. Des Weiteren wird eine Übersicht über den Aufbau sowie über die einzelnen Kapitel der Arbeit gegeben.

KAPITEL 2: Behandelt verwandte Arbeiten, die sich mit der digitalen Forensik und dem Ext4 Dateisystem beschäftigen. Im zweiten Abschnitt dieses Kapitels wird der Beitrag dieser Arbeit erläutert.

KAPITEL 3: Für ein besseres Verständnis dieser Arbeit, werden die benötigten Hintergrundinformationen in diesem Kapitel aufgearbeitet. Neben einer Einführung in die digitale Forensik wird hier auch ein Überblick über die forensische Toolsammlung *The Sleuth Kit (TSK)* gegeben.

KAPITEL 4: Kapitel vier geht detailliert auf das Ext4 Dateisystem ein. Neben dem generellen Aufbau des Dateisystems werden wichtige Strukturen, Eigenschaften und Besonderheiten von Ext4 ausführlich aufgezeigt.

KAPITEL 5: Dieses Kapitel behandelt ausschließlich das Journal des Ext4 Dateisystems. Entsprechend dem vorherigen Kapitel werden Aufbau und Strukturen des Journals erläutert.

KAPITEL 6: Im sechsten Kapitel wird anhand ausgewählter Beispiele analysiert, welche Informationen sich im Journal auffinden lassen und wie diese interpretiert wer-

den können. Dies ist die grundlegende Vorbereitung auf die spätere Implementierung eines Werkzeugs, welches die Untersuchung und Extraktion von Informationen aus dem Journal gewährleisten soll.

KAPITEL 7: Der praktische Anteil dieser Arbeit beschäftigt sich mit einer prototypischen Implementierung eines Tools zur Extraktion der gewonnenen forensischen Informationen aus den Dateisystem Journal. Dieses Kapitel beschreibt das Konzept sowie die Umsetzung und Evaluation.

KAPITEL 8: Schließlich werden das Ergebnis und die errungenen Kenntnisse diskutiert. Außerdem wird ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

KAPITEL 2

STAND DER TECHNIK

In diesem Kapitel wird ein Überblick bereits existierender Arbeiten für verwandte Themen im Bereich der digitalen Forensik und dem Ext4 Dateisystem oder dem Ext4 Journal gegeben. Der zweite Abschnitt dieses Kapitels beinhaltet eine Erläuterung über den Beitrag dieser Arbeit.

2.1. Verwandte Arbeiten

Bereits 2004 beschreibt Eckstein [Eck04] das interne Layout von UNIX-Dateisystemen. Darüber hinaus teilt er die Metadaten des Dateisystems in vier Kategorien ein und untersucht diese auf ihren forensischen Wert. Unter anderem beschreibt er das Journal dabei als interessante Quelle für forensische Informationen.

Carrier schrieb mit seinem Buch *File System Forensic Analysis* [Car05] eines der wichtigsten Referenzwerke der digitalen Forensik. Auch wenn zu diesem Zeitpunkt das Ext4 Dateisystem noch nicht existierte, beschrieb er Analysemethoden, Strukturen und Konzepte für Ext2 und Ext3, die teilweise für Ext4 übernommen werden können. Carrier wirft den Gedanken in den Raum innerhalb des Journals nach alten Kopien einer Inode zu suchen, um die Blockadressen einer gelöschten Datei zu ermitteln (vgl. [Car05, S. 317]), geht jedoch nicht weiter auf diesen ein. Weiterhin hat Carrier mit dem Sleuth Kit (siehe Kapitel 3.3.6) eine umfangreiche Softwaresammlung entwickelt, die auch heute noch eine große Rolle in der digitalen Forensik spielt.

In [Eck05] beschreibt Eckstein erneut die Kategorien des UNIX-Dateisystems und ihren forensischen Nutzen. Dieses Mal verwendet er jedoch die nach [Car05] gewählten vier Kategorien: Dateisystem, Datenblock, Metadaten und Namen. Auch hier wird das Journal als forensische Informationsquelle dargestellt, eine nähere Beschreibung bleibt allerdings aus.

Swenson et al. [SPS07] diskutieren die Dateiwiederherstellung von gelöschten bzw. überschriebenen Dateien unter den Dateisystemen ReiserFS und Ext3. Dabei werden die jeweiligen Dateisysteme und Journale beschrieben und anhand eines Experiments gezeigt, wie Dateiinhalte unter gewissen Voraussetzungen wiederhergestellt werden können.

Fairbanks et al. [FLO10] geben einen Überblick über Ext4 und beschreiben Unterschiede und Herausforderungen im Aspekt der digitalen Forensik zu vorherigen Ext-Dateisystemversionen. In einem Experiment zeigen sie wie Ext4 mit gelöschten Dateien umgeht und welche Referenzen innerhalb des Dateisystems überschrieben werden.

Fairbanks [Fai12] geht in seinem Artikel *An analysis of Ext4 for digital forensics* ein weiteres Mal auf Ext4 innerhalb forensischer Untersuchen ein. Hier beschreibt er vor allem die internen Strukturen des Dateisystems.

Kim et al. [KPLL12] untersuchen in ihrer Arbeit mobile Geräte, die ebenfalls Ext4 als Dateisystem verwenden. Das Ziel dieser Arbeit besteht in der Analyse des Journals und dem Entwurf von schnelleren Methoden der Dateiwiederherstellung als File Carving. Dabei zeigen sie in einem Experiment die Wiederherstellung einer SQLite-Datenbank auf einem Android Smartphone.

Ballethin [Bal14] stellte eine Reihe von Patches für das Sleuth Kit (TSK) zur Verfügung, die eine wesentliche Unterstützung für Ext4 bieten. Diese Aktualisierungen wurden von Fairbanks überprüft und weiterentwickelt. Mit Version 4.1.0 wurden sie von Carrier in das Sleuth Kit integriert (vgl. [Car14c]).

2.2. Beitrag dieser Arbeit

Bereits in [Eck04], [Car05], [Eck05], [SPS07], [FLO10] und [Fai12] wird das Ext3/Ext4 Dateisystem Journal als potenzielle Quelle für gelöschte Informationen genannt. Im Rahmen dieser Arbeit wird der Aufbau und die Inhalte des Ext4 Dateisystem Journals beschrieben. Anhand von ausgewählten Beispielen wird gezeigt, welche Informationen sich nach dem Erstellen, Ändern oder Löschen im Journal auffinden lassen.

Diese Arbeit geht auf die Frage ein, ob sich anhand der gewonnenen Informationen aus dem Journal gelöschte Dateien wiederherstellen lassen. Insbesondere von Interesse ist eine Wiederherstellung von fragmentierten Daten und der damit verbundene Mehrwert gegenüber anderen Konzepten wie dem File Carving.

Zu diesem Zweck wird ein Konzept entworfen, welches ein allgemeines Vorgehen beschreibt, um dieses Ziel bestmöglich zu erfüllen. Die Realisierbarkeit des in dieser Arbeit vorstellten Konzeptes wird anhand einer prototypischen Implementierung evaluiert.

KAPITEL 3

GRUNDLAGEN

Das Kapitel Grundlagen geht allgemein auf Dateisysteme ein und erklärt grundlegend das Nutzen von Journaling. Überdies wird in die digitale Forensik eingeführt und notwendige Hintergrundinformationen aufgeführt, um diese Arbeit nachvollziehen zu können.

3.1. Dateisysteme

Der Duden definiert ein Dateisystem als „Computerprogramm, das als Bestandteil des Betriebssystems das Speichern, Lesen und Löschen von Dateien auf einem Datenträger organisiert“ [Bib14].

Einfach formuliert ist das Dateisystem die Schnittstelle zwischen dem Betriebssystem und den Partitionen auf dem Datenträger. Mit Hilfe von Dateisystemen werden Daten organisiert und geordnet abgelegt. Der Endbenutzer bekommt davon in der Regel wenig mit und sieht nur, dass Verzeichnisse und Dateien vorhanden, jedoch nicht wie diese auf dem Datenträger organisiert sind.

Die Dateisysteme der meisten Betriebssysteme sind untereinander inkompatibel. Das bedeutet es existieren verschiedene Dateisysteme für die unterschiedlichen Betriebssysteme. Teilweise ist es allerdings möglich durch zusätzliche Treiber eine gewisse Kompatibilität oder zumindest eine Leseoperation zu gewährleisten. Zu den verbreitetsten aktuellen Dateisystemen gehören NTFS von Microsoft, HFS von Apple sowie das für den Linux-Kernel entwickelte Ext4, auf welches sich diese Arbeit beschränkt.

3.2. Journaling

Die meisten modernen Dateisysteme nutzen Journale, um Änderungen am System aufzuzeichnen. Durch Journale ist es möglich, das Dateisystem selbst nach einem Systemausfall in einem konsistenten Zustand zu halten.

Bevor eine Änderung im Dateisystem vorgenommen wird, werden diese in einen reservierten Speicherbereich, dem Journal, abgelegt. Durch das Journal werden alle geplanten Aktionen aufgezeichnet und können im Störungsfall zurückgenommen werden. Ein solches Zurücknehmen wird als Rollback bezeichnet (vgl. [Bun11]).

Bei Dateisystemen ohne Journaling muss nach einem nicht ordnungsgemäßen Herunterfahren des Betriebssystems das gesamte Dateisystem auf Fehler überprüft werden. Dieser Vorgang ist bei den heutigen Festplattengrößen sehr zeitaufwendig und für ein Produktivsystem nicht akzeptabel.

Grundsätzlich wird zwischen den zwei folgenden Arten des Journaling unterschieden:

3.2.1. Metadaten-Journaling

Das Metadaten-Journaling schreibt nicht alle Daten durch das Journal auf den Datenträger. Aus Performancegründen werden in diesem Modus nur die Metadaten protokolliert. Demnach nur Informationen über die Daten, die erstellt beziehungsweise geändert oder gelöscht werden.

3.2.2. Full-Journaling

Beim Full-Journaling werden im Gegensatz zum Metadaten-Journaling alle Daten durch das Journal geschrieben. Also neben den Metadaten auch die tatsächlichen Inhalte einer Datei. Das bedeutet, dass sämtliche Schreibvorgänge zweifach ausgeführt werden: ein Schreibvorgang in das Journal und ein zweiter Schreibvorgang auf den Datenträger.

3.3. Digitale Forensik

Die digitale Forensik hat in den letzten Jahren enorm an Bedeutung gewonnen. Während anfängliche Definitionen den Begriff Computer-Forensik verwendeten, wird heute vermehrt auf Bezeichnungen wie digitale Forensik oder auch IT-Forensik gesetzt. Längst spielen nicht nur noch Computer-Systeme sondern auch Mobilgeräte wie Smartphones und Tablets eine wichtige Rolle und enthalten Unmengen an Informationen, die bei einer Untersuchung berücksichtigt werden müssen.

Generell befasst sich die digitale Forensik mit der Untersuchung von verdächtigen Vorfällen an IT-Systemen. Dazu gehört neben dem Feststellen des Tatbestandes auch die Dokumentation und Präsentation der errungenen Erkenntnisse. Da Untersuchungsberichte vor Gericht standhalten müssen, sind sie ein fundamentaler Bestandteil des Gesamtprozesses.

Geschonneck definiert den Begriff der digitalen Forensik als „*[. . .] den Nachweis und die Ermittlung von Straftaten im Bereich der Computerkriminalität*“ [Ges12, S. 2]. Eine alternative Definition aus dem Leitfaden „IT-Forensik“ des Bundesamt für Sicherheit in der Informationstechnik (BSI) bezeichnet diesen Begriff folgendermaßen:

„*IT-Forensik ist die streng methodisch vorgenommene Datenanalyse auf Datenträgern und in Computernetzen zur Aufklärung von Vorfällen unter Einbeziehung der Möglichkeiten der strategischen Vorbereitung insbesondere aus der Sicht des Anlagenbetreibers eines IT-Systems*“ [Bun11, S. 13].

Neben den polizeilichen Behörden ist die digitale Forensik mittlerweile in vielen IT- und Beratungsunternehmen vertreten. Unter anderem auch in den Big Four[2] der Wirtschaftsprüfungsgesellschaften, die ihre Dienste und das damit verbundene Fachwissen auf diesem Gebiet erfolgreich anbieten.

3.3.1. Ziele einer Ermittlung

Das zentrale Ziel einer Ermittlung ist es den Richter von der Täterschaft eines Angeklagten zu überzeugen oder seine Unschuld zu beweisen. Diesem Zweck dient die Beantwortung der folgenden kriminalistischen Fragestellungen (vgl. [Bun11, S. 22]):

- Was ist geschehen?

[2] Als Big Four werden die vier Wirtschaftsprüfungsgesellschaften bezeichnet, die die überwiegende Mehrheit der börsennotierten Kapitalgesellschaften weltweit prüfen und beraten [Wik15].

- Wo ist es passiert?
- Wann ist es passiert?
- Wie ist es geschehen?
- Wer hat es getan?
- Was kann gegen eine Wiederholung getan werden?

3.3.2. Anforderungen an eine forensische Untersuchung

Damit die gewählten Methoden und eingesetzten Hilfsmittel auch vor Gericht Bestand haben, ist es unumgänglich diese angemessen und passend einzusetzen. Zudem werden an eine forensische Untersuchung Anforderungen an die Vorgehensweise gestellt (vgl. [Ges12, S. 66 f.]):

Akzeptanz

Angewandte Methoden und Schritte müssen in der Fachwelt beschrieben und allgemein anerkannt sein. Der Einsatz neuer Techniken ist grundsätzlich möglich, die korrekte Arbeitsweise muss jedoch nachgewiesen werden können.

Glaubwürdigkeit

Die Funktionalität und Robustheit der Methoden müssen glaubwürdig sein. Der Nachweis dessen sollte gegebenenfalls aufgezeigt werden.

Wiederholbarkeit

Sämtliche Schritte und aufgeführte Ergebnisse müssen reproduzierbar sein. Eine dritte Person, die die gleichen Schritte mit dem selben Ausgangsmaterial durchführt, muss die gleichen Ergebnisse erhalten.

Integrität

Im Rahmen des Untersuchungsprozesses dürfen die sichergestellten Spuren nicht verändert werden. Die Integrität der Beweise muss jederzeit belegbar sein.

Ursache und Auswirkungen

Die gewählten Methoden müssen die Möglichkeit bieten eine nachvollziehbare Verbindung zwischen Personen, Ereignissen und Beweisspuren herzustellen.

Dokumentation

Jeder Schritt des Ermittlungsprozesses muss angemessen dokumentiert werden.

3.3.3. Untersuchung

Allgemein lässt sich die digitale Forensik in zwei Bereiche bzgl. des Zeitpunktes der Untersuchung einteilen. Die Live-Analyse und die Post-mortem-Analyse.

Live-Analyse

Bei der Live-Analyse (auch bekannt als Online-Analyse) findet die Analyse zur Laufzeit des zu untersuchenden Systems statt. In erster Linie wird hier versucht sogenannte flüchtige Daten für eine spätere Untersuchung zu sichern, da diese ansonsten verloren gehen könnten. Dazu gehören unter anderem der Hauptspeicherinhalt (RAM) und Informationen über bestehende Netzwerkverbindungen und laufende Prozesse (vgl. [Bun11, S. 13]).

Post-mortem-Analyse

Demgegenüber steht die Post-mortem-Analyse (oder auch Offline-Analyse). Hier findet die Untersuchung meist auf Datenträgerabbildern, sogenannten Images statt. Eines der Hauptziele liegt dabei auf der Gewinnung und Untersuchung von gelöschten oder versteckten Dateien (vgl. [Bun11, S. 13]). Kopien des Original-Datenträgers ermöglichen es Beweismittel zu sammeln ohne die Dateien zu verändern (vgl. [Ges12, S. 105]).

3.3.4. Untersuchungsphasen

Für die Durchführung von forensischen Untersuchungen existieren verschiedene Vorgehensmodelle, die den Ablauf erleichtern sollen. Beispielsweise sei hier das Secure-

Analyse-Present (S-A-P) Modell aus [Ges12, S. 68 f.] genannt, das die Untersuchung in drei große Phasen unterteilt:

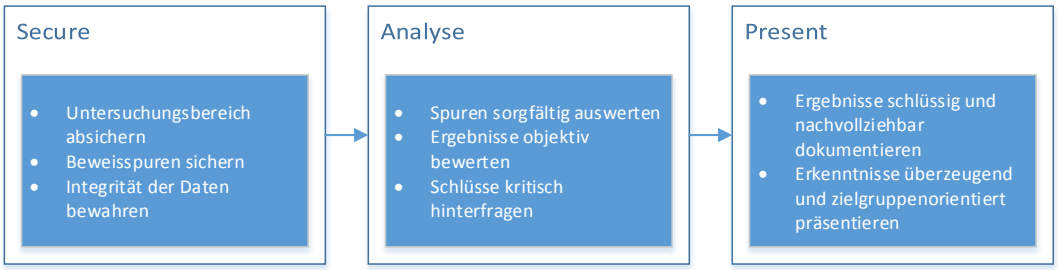


Abbildung 3.1.: Phasen des S-A-P-Modells

Secure

Das Erste, was nach der Feststellung eines Vorfalls geschieht, ist die Sicherung von Spuren und Tatort. In der Secure-Phase werden alle Daten sorgfältig erfasst. Zudem wird durch geeignete Methoden dafür gesorgt, dass die gesammelten Informationen in einer eventuell späteren juristischen Würdigung ihre Beweiskraft nicht verlieren. Aus diesem Grund werden alle Tätigkeiten sorgfältig protokolliert. Darüber hinaus müssen gesammelte Daten vor versehentlicher oder gar beabsichtigter Manipulation geschützt werden. Dies geschieht beispielsweise mit Hashfunktion und dem Vier-Augen-Prinzip.

Analyse

Nachdem die Daten in der ersten Phase forensisch korrekt gesichert wurden, müssen sie analysiert werden. Dies geschieht in der Analyse-Phase, in der alle Spuren akribisch genau analysiert und die gewonnenen Erkenntnisse objektiv bewertet und dokumentiert werden. Sämtliche Schlüsse müssen kritisch hinterfragt werden, um eventuelle Lücken in der Argumentationskette selbstständig und sicher zu identifizieren.

Present

Am Schluss der forensischen Untersuchung werden die Ergebnisse aufbereitet und präsentiert. In welcher Form dies erfolgt, hängt in dieser Phase oft von der konkreten Fragestellung des Sicherheitsvorfalls ab. Grundsätzlich muss das Ergebnis jedoch Personen

überzeugen, die während der Ermittlung nicht zugegen waren und eventuell nicht den technischen Sachverstand mitbringen, um sämtliche Details zu verstehen. Das bedeutet, dass alle Erkenntnisse schlüssig und nachvollziehbar dokumentiert und anschließend überzeugend und zielgruppenorientiert präsentiert werden müssen.

3.3.5. File Carving

File Carving ist eine Methode, für die Wiederstellung von Dateien auf Datenträgern ohne die Hilfe des Dateisystems. Da File Carver sich nicht auf das Dateisystem stützen, um herauszufinden, wo eine Datei beginnt und wo sie endet, brauchen sie andere Informationsquellen. Zu diesem Zweck nutzen sie die besonderen Eigenschaften der verschiedenen Dateitypen. Von besonderem Interesse sind hier der Dateianfang (Header) und das Dateiende (Footer). File Carver durchsuchen den vollständigen Datenträger, inklusive des nicht-allozierten Speicherbereichs, nach den ihnen bekannten Headern und Footern. Die Datenblöcke dazwischen werden als neue Datei herausgeschrieben. Da jedoch nicht alle Dateiformate ein eindeutiges Dateiende besitzen, werden zusätzliche Maßnahmen beim Carving verwendet. Ein Beispiel hierfür ist eine maximale Dateigröße. Wird ein Header gefunden und innerhalb der zulässigen Größe kein dazugehöriger Footer, wird das herausschreiben beendet (vgl. [Spe08, Ste14]).

Existieren in einem Datenträger noch Header von alten Dateien, können File Carver diese identifizieren und extrahieren. Dabei spielt es keine Rolle, ob die Datei vom Dateisystem gelöscht wurde. Eine Herausforderung bei File Carving ist die sogenannte Fragmentierung einer Datei. Dies bedeutet, dass eine Datei physikalisch auf nicht hintereinanderliegenden Datenblöcken der Festplatte geschrieben ist. Die Datei ist verteilt abgelegt und in mehrere logisch zusammengehörige Datenblöcke aufgeteilt.

In solchen Fällen können File Carver immer noch den Header einer Datei erkennen, es ist allerdings nicht mehr möglich die Datei vollständig herauszuschreiben. Bestenfalls werden einzelne Fragmente erkannt, die unter Umständen Hinweise zur Interpretation der Datei geben. Ein Beispiel hierfür ist in Abbildung 3.2 zu sehen. Diese Darstellung zeigt ein, mittels File Carving, rekonstruiertes Bildfragment.

Abbildung 3.2.: Durch File Carving erkanntes Bildfragment

3.3.6. The Sleuth Kit

Das *The Sleuth Kit (TSK)*[3] ist eine Sammlung von Kommandozeilen-Tools, die eine forensische Untersuchung von Datenträgern ermöglichen. Entwickelt wurde diese Werkzeugsammlung hauptsächlich von Brian Carrier, dem Autor des Buches *File System Forensic Analysis* [Car05].

Mit Hilfe der Tools können Datenträger auf ihr Layout und die Partitionierung untersucht und einzelne Bereiche für eine Datenträgeranalyse extrahiert werden. Die Dateisystem-Tools erlauben dabei eine nichtinvasive Untersuchung eines Computersystems. Die Werkzeuge arbeiten unabhängig von dem zugrundeliegenden Betriebssystem und können Daten aus der internen Dateisystemstruktur extrahieren, die normalerweise vor dem Betriebssystem verborgen sind.

Das Sleuth Kit unterstützt folgende Dateisysteme: NTFS, FAT, UFS 1, UFS 2, HFS, YAFFS 2, Ext2, Ext3 und seit Version 4.1.0 (Juni 2013) auch Ext4 (vgl. [Car14c]).

Um den Umgang mit dem Sleuth Kit zu erleichtern, wurde die grafische Oberfläche Autopsy[4] geschaffen, die genauso wie das Sleuth Kit in C und Perl geschrieben wurde und für alle gängigen Betriebssysteme frei zur Verfügung steht (vgl. [Car14a]).

Die wesentlichen Tools des Sleuth Kits sind inklusive ihrer Funktionen auf nachfolgender Seite aufgeführt (vgl. [Car14b]).

[3] http://www.sleuthkit.org/sleuthkit/
[4] http://www.sleuthkit.org/autopsy/

Dateisystem-Ebene

fsstat: Zeigt Details zum Dateisystem und Statistiken inklusive Layout, Größen und Bezeichnungen.

Dateinamen-Ebene

ffind: Findet allozierte sowie nicht-allozierte Dateinamen, die auf eine gegebene Metadatenstruktur verweisen.

fls: Listen allozierte und gelöschte Dateinamen in einem Verzeichnis.

Metadaten-Ebene

icat: Extrahiert Dateneinheiten einer Datei anhand ihrer Metadatenadresse.

ifind: Findet zu einem gegebenen Dateinamen die entsprechenden Metadaten oder die Metadaten, die zu einer bestimmten Datei verweisen.

ils: Listet die Metadatenstruktur und deren Inhalte auf.

istat: Zeigt Statistiken und Details zu einer gegebenen Metadatenstruktur.

Datenebene

blkcat: Extrahiert den Inhalt eines Datenblocks.

blkls: Listet Details zu Dateneinheiten und kann nicht-allozierten Speicherbereich extrahieren.

blkstat: Zeigt Statistiken zu einer gegebenen Dateneinheit.

blkcalc: Berechnet den Ort an dem Daten am Originaldatenträger zu finden sind, die am Image im unallozierten Speicherbereich gefunden wurden.

Dateisystemjournal-Ebene

jcat: Zeigt den Inhalt eines bestimmten Journalblocks.

jls: Listet die Einträge im Dateisystemjournal auf.

Datenträger-Ebene

mmls: Zeigt das Layout einer Festplatte, einschließlich der unallozierten Bereiche.

mmstat: Zeigt Details über den Datenträger.

mmcat: Extrahiert den Inhalt eines bestimmten Datenträgers.

KAPITEL 4

EXT4 DATEISYSTEM

In diesem Kapitel wird das Linux-Dateisystem Ext4 eingeführt. Ext4 ist die aktuelle Version der ExtX Dateisystem-Familie und damit Nachfolger von Ext3. Bei den meisten Linux-Distributionen wird Ext4 standardmäßig verwendet und ist damit das verbreitetste Dateisystem in der Linux-Umgebung. Seit Version 2.3 (Codename „Gingerbread“) des Betriebssystems Android wird es ebenfalls für mobile Geräte eingesetzt (vgl. [Tso10]).

Als Nachfolger von Ext3 bringt es zwei wesentliche Änderungen mit sich: Die Blocknummern wurden auf 48 Bit erweitert und eine Verwendung von indirekter Blockadressierung, wurde durch sogenannte Extents ersetzt (vgl. [Die09]). Tabelle 4.1 zeigt den dadurch gewonnen Speichervorteil gegenüber dem Vorgänger Ext3.

Dateisystem	Blocknummern	max. Dateigröße	max. Dateisystemgröße
Ext3	32 Bit	2 TByte	16 TByte
Ext4	48 Bit	16 TByte	1024 PByte

Tabelle 4.1.: Vergleich Datengrößen Ext3 und Ext4

Ext4 arbeitet mit Blocknummern einer Länge von 48 Bit. Bei einer Standard-Blockgröße von 4096 Byte erlaubt dies eine maximale Dateisystemgröße von bis zu $2^{48} * 4096\ Byte$, also einem Exabyte (1024 PByte). Die maximale Dateigröße ergibt sich durch den Aufbau der Extents, der in einem späteren Abschnitt detailliert beschrieben werden.

4.1. Verbreitung

Linux

Mit der Ablösung von Ext3 wird Ext4 standardmäßig als Dateisystem für die meisten Linux-Distributionen eingesetzt. Damit ist es, zumindest in der Linux-Welt, das am häufigsten eingesetzte Betriebssystem. Im Vergleich zu anderen Betriebssystemen wie Windows oder Macintosh hat Linux jedoch einen relativ kleinen Anteil von lediglich einem Prozent für Endbenutzer, siehe Abbildung 4.1 (vgl. [Net13]).

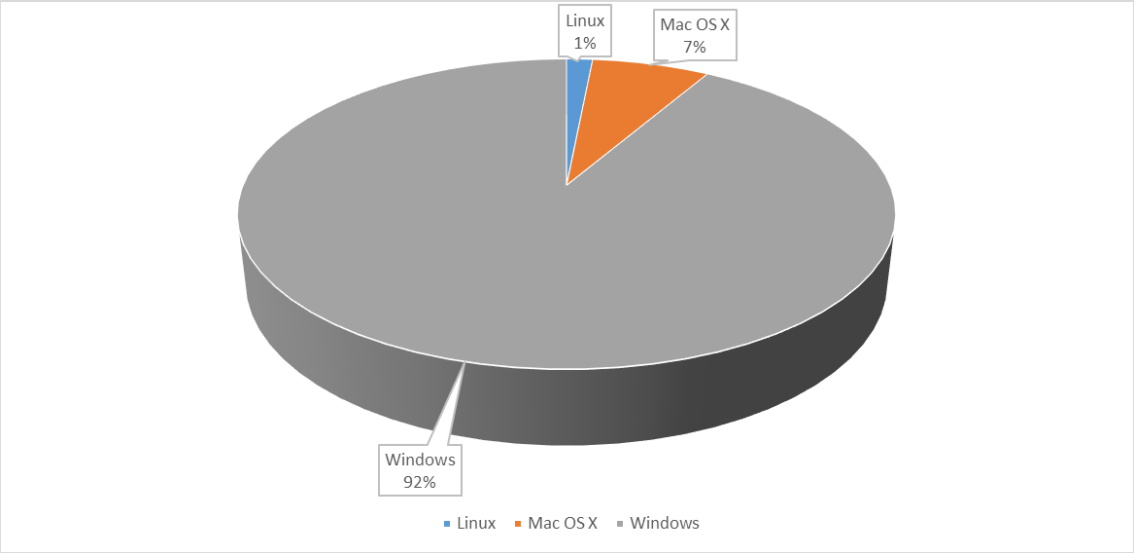


Abbildung 4.1.: Desktop Marktanteile nach Betriebssystem

Android

Weitaus verbreiteter ist der Einsatz von Ext4 bei mobilen Geräten. Während der Desktopbereich von Linux nie erobert werden konnte, hat sich mit Android ein auf dem Linux-Kernel basierendes Betriebssystem etabliert. Seit 2010 wird Ext4 als Dateisystem in Android eingesetzt (vgl. [Tso10]) und ist somit auf fast jedem Android-Gerät zu finden. Laut aktuellen Zahlen von [Net15] besitzt Android mit 46,97% den zur Zeit größten Marktanteil, siehe Abbildung 4.2. Folglich kann behauptet werden, dass Ext4 das am weitesten verbreitete Dateisystem für mobile Geräte ist und damit eine große Motivation für eine forensische Analyse darstellt.

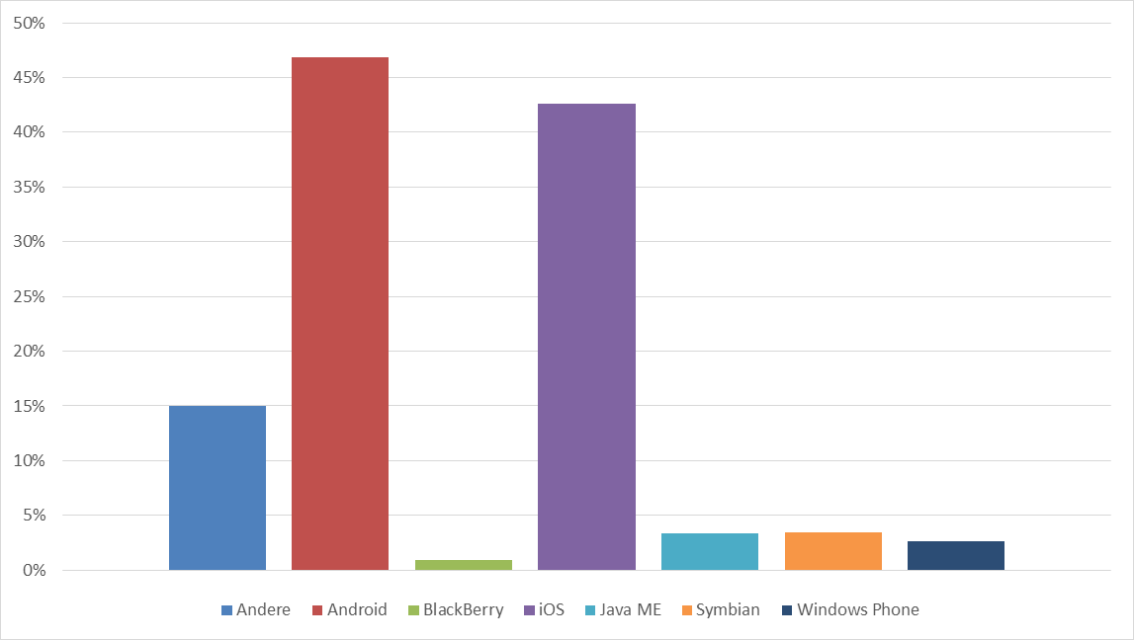


Abbildung 4.2.: Smartphone Marktanteile nach Betriebssystem

4.2. Layout

Das Dateisystem-Layout der Ext4 Partition beginnt mit der *Reserved Area*. Diese besteht aus den ersten 1024 Bytes des ersten Datenblocks und bietet bei Bedarf Platz für den Bootsektor. Dessen ungeachtet befindet sich ab Byte 1024 des ersten Datenblocks der *Superblock* des Dateisystems, der alle grundlegenden Informationen der Partition beinhaltet. Die typische Datenblockgröße beträgt heute 4096 Bytes. Falls aus irgendeinem Grund die minimale Blockgröße von 1024 Bytes verwendet wird, ist der erste Datenblock für den Bootsektor bestimmt und der *Superblock* wandert in den zweiten Datenblock. Dies ist jedoch die Ausnahme für den Anfang der Ext4 Partition. Im Normalfall beginnt eine Blockgruppe direkt mit dem *Superblock*. Anschließend folgen die einzelnen *Group Descriptors*, die zusammen die *Group Descriptor Table (GDT)* bilden. Diese belegen in der Regel ebenfalls einen Datenblock. Zudem werden weitere Datenblöcke beim Formatieren des Dateisystems reserviert, um zukünftige Erweiterungen des Dateisystems zu gewährleisten (*Reserved GDT Blocks*). Daraufhin schließen sich jeweils ein Datenblock für die *Data Bitmap* und einer für die *Inode Bitmap* an. Die darauf folgende *Inode Table* besteht aus mehreren aufeinanderfolgenden Datenblöcken. Die restlichen Datenblöcke einer Blockgruppe sind die *Data Blocks*, die den individuell nutzbaren Speicherbereich des Dateisystems bieten (vgl. [Lin15]). Abbildung 4.3 stellt das soeben beschriebene Layout grafisch dar.

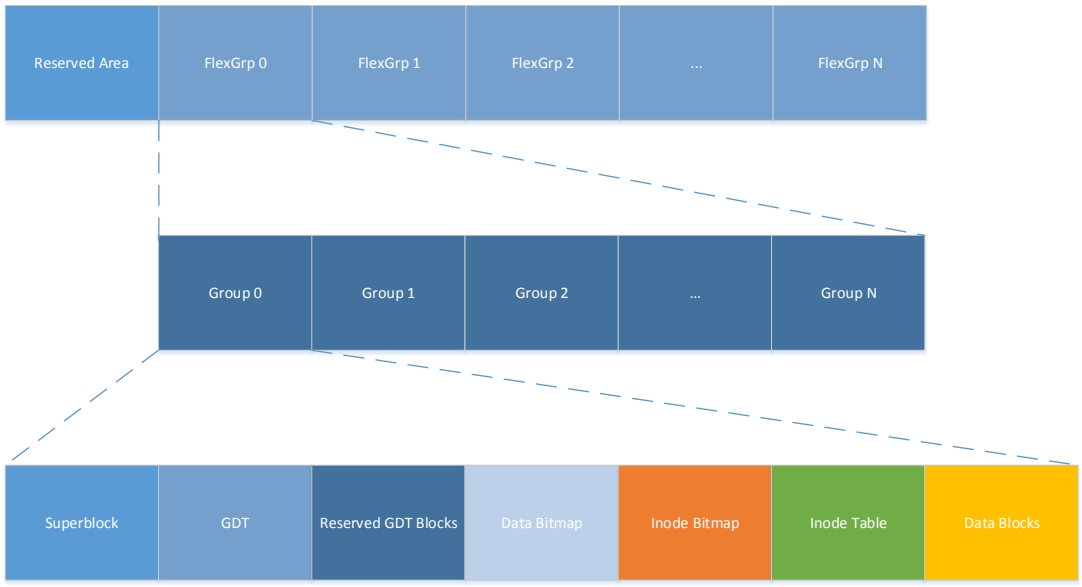


Abbildung 4.3.: Ext4 Dateisystem Layout

Eine Neuerung, die mit Ext4 hinzu kam, sind die *Flexible Block Groups*. Diese binden mehrere Blockgruppen als eine logische Blockgruppe zusammen. Bei Verwendung der *Flexible Block Groups* werden die Speicherbereiche der Bitmaps und Inode Table in der ersten Blockgruppe erweitert, um die Bitmaps und Inode Table aller dazugehörigen Blockgruppen einzufügen. Ist die Größe der *FlexGrp* beispielsweise 4, enthält Blockgruppe 0 den Superblock, die GDT, die Data- und Inode Bitmaps der Gruppen 0-3 sowie die Inode Table der Gruppen 0-3. Hieraus entsteht der Vorteil, dass Metadaten nahe beieinander liegen und damit schneller geladen werden können, sowie größeren zusammenhängenden Speicherbereich, um Fragmentierung entgegenzuwirken (vgl. [Lin15]).

4.2.1. Superblock

Der Ext4 Superblock enthält verschiedene Informationen über das gesamte Dateisystem, wie zum Beispiel die Anzahl der Datenblöcke und verfügbaren Inodes, unterstützte Funktionen oder Wartungsinformationen. Der original Superblock befindet sich innerhalb des ersten Datenblocks des Dateisystems, direkt hinter dem Bootsektor. Dieser ist auch der Einzige, der vom Dateisystem aktiv verwendet wird. Aus Sicherheitsgründen existieren jedoch Kopien, verteilt auf dem ganzen Dateisystem. Durch das Flag *sparse_super* innerhalb des Superblocks wird bestimmt in welchen Blockgruppen sich die redundanten Kopien des Superblocks und der GDT befinden. Ist dieses Flag gesetzt, werden die Kopien

nur in Gruppen der Potenzen von 3, 5 und 7 abgelegt, anderenfalls in jeder Blockgruppe des Dateisystems. Mit Hilfe dieser Sicherheitskopien kann der Superblock im Fehlerfall wiederhergestellt werden. Dies ist unbedingt notwendig, da der Superblock für die Verwendung des Dateisystems unerlässlich ist. Die wesentlichen Informationen, die der Superblock enthält, sind folgende:

- Anzahl der verfügbaren Datenblöcke und Inodes
- Lage des ersten Datenblocks
- Größe eines Datenblocks
- Datenblöcke und Inodes pro Gruppe
- Zeitstempel und Zähler bzgl. des Mounten
- Inode-Nummer des Journals
- Flags und weitere Dateisystem Informationen

Die Gesamtgröße des Superblocks beläuft sich auf 1024 Bytes (vgl. [Lin15]).

4.2.2. Group Descriptor Table

Jede Blockgruppe innerhalb des Dateisystems wird durch einen Group Descriptor beschrieben. Zusammen bilden diese die Group Descriptor Table (GDT). Wie in Abbildung 4.3 gezeigt, befindet sich die GDT unmittelbar nach dem Superblock am Anfang des Dateisystems. Ebenso wie bei dem Superblock werden redundante Kopien der GDT aus Sicherheitsgründen in den Blockgruppen abgelegt, welches sich durch das Flag *sparse_super* beeinflussen lässt. Verwendet wird jedoch ausschließlich die GDT in Blockgruppe 0.

Innerhalb einer Blockgruppe sind der Superblock und die GDT die einzigen Strukturen mit festgelegter Position. Die Lage und Größe der anderen Strukturen, wie den Bitmaps oder der Inode Table können variieren. Aus diesem Grund sind diese innerhalb der GDT für jede Gruppe beschrieben. Ein einzelner Group Descriptor enthält dabei folgende Informationen:

- Adresse der Block Bitmap
- Adresse der Inode Bitmap
- Adresse der Inode Table

- Zähler für freie Blöcke, Inodes und Verzeichnisse
- Blockgruppen Flags
- Prüfsummen

Die Größe eines Eintrags beträgt je nach verwendeter Architektur 32 beziehungsweise 64 Byte. Die tatsächlich verwendete Größe steht jedoch stets im Superblock. Wie bereits erwähnt, existiert für jede Blockgruppe ein Group Descriptor. Alle Einträge zusammen ergeben die Group Descriptor Table (GDT), die je nach Dateisystemgröße aus einem oder mehreren Datenblöcken besteht (vgl. [Lin15]).

4.2.3. Data- und Inode Bitmaps

Die Bitmap ist ein Abbild der freien und belegten Blöcke in der Blockgruppe. Jedes Bit in der Data Bitmap repräsentiert einen Datenblock. Die Data Bitmap ist genau einen Datenblock groß. Bei einer Größe von 4096 Byte sind das 32768 Datenblöcke, die innerhalb dieser Bitmap abgebildet werden können. Dadurch ergibt sich die Größe einer Blockgruppe von 128 MiB. Ein Bit der Inode Bitmap repräsentiert eine Inode. Anhand dieser Bitmaps lässt sich der Belegt-Status von Datenblöcken bzw. Inodes effizient ermitteln (vgl. [Lin15]).

4.2.4. Inode Table

In einem regulären UNIX-Dateisystem speichert eine Inode alle zu einer Datei gehörenden Metadaten, inklusive Zeitstempel, Zugriffsrechten und die Position der Datei. Um die Informationen einer Datei zu finden, müssen die Verzeichnisdateien durchquert und der entsprechende Verzeichniseintrag gefunden werden. Dieser enthält die Inode-Nummer, die eine Assoziation mit der Inode ermöglicht.

Die Inode Table ist nicht ein Eintrag, sondern enthält alle Inodes einer Blockgruppe. Die Größe der Tabelle berechnet sich durch die Informationen aus dem Superblock: *Größe einer Inode * Inodes pro Gruppe*.

Während die übliche Größe einer Inode in Ext2 und Ext3 auf 128 Byte festgelegt war, wurde sie in Ext4 auf 256 Byte erweitert (vgl. [Lin15]).

Aufbau eines Inode-Eintrags

Nachfolgende Tabelle 4.2 zeigt die Struktur eines Inode-Eintrags. Die darin enthaltenden Informationen sind essentiell für die Abbildung einer Datei.

Offset	Länge	Name	Beschreibung
0x00	16 Bit	i_mode	Dateityp und Zugriffsrechte
0x02	16 Bit	i_uid	UID des Dateibesitzers
0x04	32 Bit	i_size_lo	Untere 32 Bits für die Dateigröße
0x08	32 Bit	i_atime	Letzter Lesezugriff
0x0C	32 Bit	i_ctime	Letzte Änderung der Inode
0x10	32 Bit	i_mtime	Letzte Dateiänderung
0x14	32 Bit	i_dtime	Zeitpunkt des Löschens
0x18	16 Bit	i_gid	Besitzer der Gruppe (GID)
0x1A	16 Bit	i_links_count	Hardlink Zähler
0x1C	32 Bit	i_blocks_lo	Anzahl Datenblöcke
0x20	32 Bit	i_flags	Spezielle Inode-Flags
0x24	32 Bit	osd1	Abhängig vom Erzeuger des Dateisystems
0x28	60 Byte	i_block	Extents
0x64	32 Bit	i_generation	Datei Version
0x68	32 Bit	i_file_acl_lo	Erweiterte Attribute
0x6C	32 Bit	i_size_high	Obere 32 Bits für die Dateigröße
0x70	32 Bit	i_obso_faddr	Obsolet
0x74	12 Byte	osd2	Abhängig vom Erzeuger des Dateisystems
0x80	16 Bit	i_extra_isize	Größe der Inode
0x82	16 Bit	i_checksum_hi	Untere 16 Bit der Inode-Prüfsumme
0x84	32 Bit	i_ctime_extra	Zusätzliche Genauigkeit für die CTIME
0x88	32 Bit	i_mtime_extra	Zusätzliche Genauigkeit für die MTIME
0x8C	32 Bit	i_atime_extra	Zusätzliche Genauigkeit für die ATIME
0x90	32 Bit	i_crtime	Zeitpunkt der Dateierstellung
0x94	32 Bit	i_crtime_extra	Zusätzliche Genauigkeit für die CRTIME
0x98	32 Bit	i_version_hi	Obere 32 Bit für Versionsnummer

Tabelle 4.2.: Aufbau einer Inode [Lin15]

Spezielle Inodes

Die ersten zehn Inodes des Dateisystems können nicht vergeben werden, da sie vom System reserviert sind. Inode 0 wird nicht verwendet, die weiteren Inodes sind für spezielle Systemdateien reserviert. Tabelle 4.3 listet alle reservierten Inodes mit einem kurzen Verwendungszweck auf. Inode 11 ist demnach die erste freie Inode, wird jedoch üblicherweise für das *„lost+found“* Verzeichnis verwendet (vgl. [Lin15]).

Inode	Beschreibung
0	Existiert nicht; es gibt keine Inode 0
1	Liste der fehlerhaften Blöcke
2	Stammverzeichnis (Root directory)
3	Benutzer Quota
4	Gruppen Quota
5	Bootloader
6	Wiederherstellungsverzeichnis
7	Reservierte *Group Descriptor* Inode („resize inode“)
8	Journal Inode
9	„Exclude“ Inode
10	„Replica“ Inode
11	Inode des „lost+found“ Verzeichnisses.

Tabelle 4.3.: Reservierte Inodes [Lin15]

4.3. Verzeichniseinträge

In Ext4 gibt es keine feste Zuordnung zwischen Dateinamen und Datei. Beim Erstellen einer neuen Datei wird zunächst eine Inode-Nummer als Referenz benutzt und als nächstes ein Verzeichniseintrag mit dem Dateinamen erzeugt, der auf diese Inode verweist. Ein Verzeichniseintrag ermöglicht demgemäß die Zuordnung zwischen Dateiname und Datei. Dabei können beliebig viele Verzeichniseinträge auf dieselbe Datei verweisen. Der Dateiname darf maximal 255 Zeichen enthalten. Folglich ist ein Verzeichniseintrag höchstens 263 Bytes lang. Die exakte Größe lässt sich nur durch das Attribut *rec_len* ermitteln (vgl. [Lin15]). Tabelle 4.4 zeigt den Aufbau eines Verzeichniseintrags.

Offset	Länge	Name	Beschreibung
0x00	32 Bit	inode	Inode-Nummer der Datei
0x04	16 Bit	rec_len	Länge dieses Verzeichniseintrags
0x06	8 Bit	name_len	Länge des Dateinamens
0x07	8 Bit	file_type	Kennzeichen für eins der folgenden: 0x0: Unknown 0x1: Regular File 0x2: Directory 0x3: Character device file 0x4: Block device file 0x5: FIFO 0x6: Socket 0x7: Symbolic link
0x08	name_len	name	Dateiname

Tabelle 4.4.: Aufbau eines Verzeichniseintrags [Lin15]

4.4. Extents

Eine wesentliche Neuerung in Ext4 ist die Verwendung von Extents anstelle der in Ext3 verwendeten indirekten Blockadressierung. Extents adressieren keine Datenblöcke, sondern mappen stattdessen einen möglichst großen Bereich einer Datei. Dazu werden drei Werte benötigt: Die Anfangsadresse, eine Länge und ein Offset. Die Anfangsadresse gibt die Adresse des ersten Datenblocks an, die Länge ist die Anzahl der folgenden dazugehörigen Datenblöcke und der Offset gibt an, welche Position der aktuelle Eintrag in der Datei hat (vgl. [Die09, S. 181], [Lin15]).

Offset	Länge	Name	Beschreibung
0x00	32 Bit	ee_block	Erste Dateiblocknummer, die dieses Extent abdeckt
0x04	16 Bit	ee_len	Anzahl Blöcke des Datenbereichs
0x06	16 Bit	ee_start_hi	Obere 16 Bit der Blockadresse
0x08	32 Bit	ee_start_lo	Untere 32 Bit der Blockadresse

Tabelle 4.5.: Aufbau eines Extents [Lin15]

Ext4 benutzt die 60 Byte innerhalb der Inode (siehe Tabelle 4.2), die Ext3 zum Speichern von 15 32 Bit großen Blocknummern verwendet, um dort einen Extent Header (Tabelle

4.6) und vier Extents (Tabelle 4.5) von jeweils 12 Byte Länge abzulegen. Dadurch können Daten bis 512 MByte direkt aus dem Inode verwaltet werden (vgl. [Die09, Lin15]).

Offset	Länge	Name	Beschreibung
0x00	16 Bit	eh_magic	Magic Number, 0xF30A
0x02	16 Bit	eh_entries	Anzahl der gültigen Einträge nach dem Header
0x04	16 Bit	eh_max	Max. Anzahl der Einträge
0x06	16 Bit	eh_depth	Tiefe des Extent-Knoten
0x08	32 Bit	eh_generation	Derzeit ungenutzt

Tabelle 4.6.: Aufbau des Extent Headers [Lin15]

Überschreitet eine Datei diese Grenze, baut Ext4 einen Baum aus Extents auf. Für diesen Zweck kommt eine weitere Datenstruktur, der Extent Index (Tabelle 4.7), zum Einsatz.

Offset	Länge	Name	Beschreibung
0x00	32 Bit	ei_block	Erste Dateiblocknummer, die dieses Extent abdeckt
0x04	32 Bit	ei_leaf_lo	Untere 32 Bit der Blocknummer des nächsten Knotens. Dieser kann entweder ein weiterer interner Knoten oder Extents enthalten
0x08	16 Bit	ei_leaf_hi	Obere 16 Bit des vorherigen Feldes
0x0A	16 Bit	ei_unused	Unbenutzt

Tabelle 4.7.: Aufbau des Extent Index [Lin15]

Der Extent Index enthält lediglich die Startposition des Extents in der Datei und die Blocknummer auf der Festplatte. In diesem Datenblock können entweder Extents stehen, die auf die Daten verweisen, oder ein weiterer Extent Index, wobei jeder Block mit einem Extent Header beginnt.

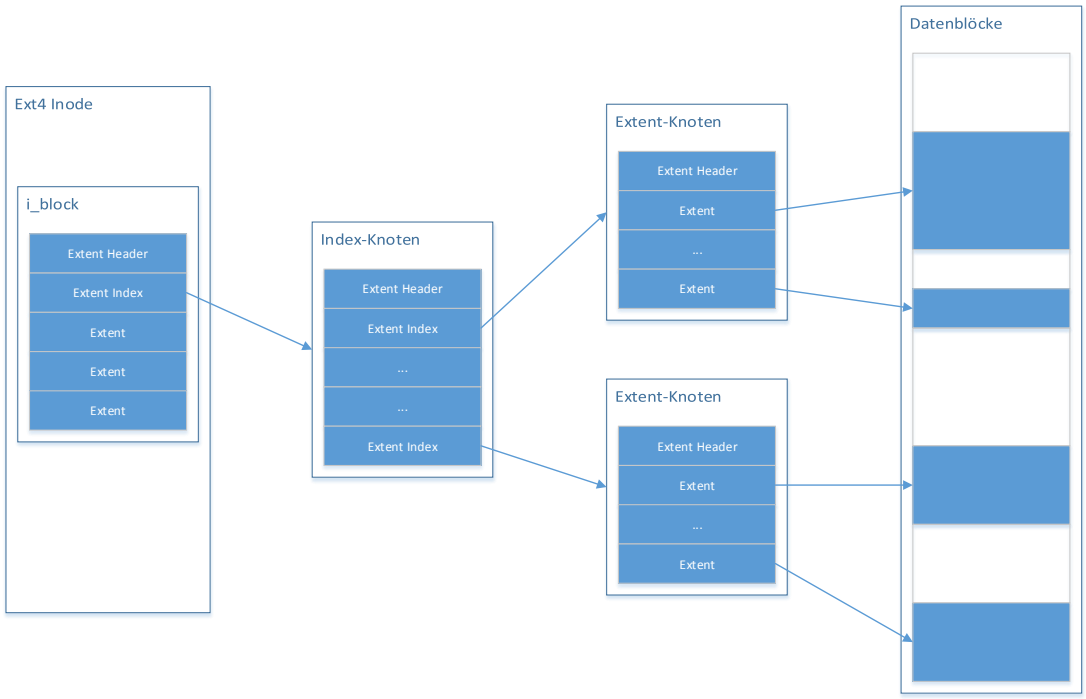


Abbildung 4.4.: Ext4 Extent-Baum

Abbildung 4.4 visualisiert das Prinzip eines Extent-Baumes, der bei großen, beziehungsweise fragmentierten Dateien aufgebaut wird. Innerhalb der Inode befindet sich anstelle von Extents, die direkt auf einen Datenbereich verweisen, ein Extent Index. Dieser verweist auf einen Datenblock, der wiederum Extent-Indizes enthält. Erneut wird auf Datenblöcke verwiesen, welche dieses Mal Extents enthalten. Jeder Extent mappt nun einen entsprechenden Bereich der Datei. Anhand der Nummerierung innerhalb der Extents, kann dabei die Reihenfolge berücksichtigt werden.

4.5. Zeitstempel

Ursprünglich beinhaltete die Inode in älteren Versionen vier Zeitstempel: Die Zugriffszeit (ATIME), die Änderungszeit der Inode (CTIME), die Änderungszeit der Daten (MTIME) und die Löschzeit (DTIME). Diese vier Attribute werden in 32 Bit Integer-Attributen gespeichert, die die Sekunden seit Beginn der Unix-Zeit, also die vergangenen Sekunden seit dem 1. Januar 1970 Uhr repräsentiert. Mit Ext4 kam ein weiterer Zeitstempel hinzu: die Erstellungszeit (CRTIME). Des Weiteren wurden die [MAC]- und der CR-Zeitstempel um ein extra 32 Bit Integer-Attribut erweitert. Dieses erweitert den ursprünglichen Zeit-

stempel von 32 auf 34 Bit und nutzt die restlichen 30 Bits, um die Nanosekunden zu zählen. Dadurch kann zum einen ein längerer Zeitraum abgebildet und zum anderen eine höhere Genauigkeit erzielt werden (vgl. [Lin15, KL14]).

ATIME Der Access Time Zeitstempel gibt an, wann eine Datei das letzte Mal gelesen wurde. Dies kann auch ein Lesen der Metadaten sein.

CTIME Die Change Time wird verändert, wenn die Metadaten einer Datei und nicht ihr eigentlicher Inhalt geändert wird.

MTIME Die Modification Time gibt an, wann der Inhalt einer Datei zuletzt geändert wurde.

DTIME Deletion Time wird gesetzt, wenn eine Datei gelöscht wird.

CRTIME Die Creation Time gibt an, wann die Datei bzw. die Inode erstellt wurde.

KAPITEL 5

EXT4 JOURNAL

Das Ext4 Dateisystem verwendet ein Journal, um das Dateisystem bei Systemabstürzen vor Datenkorruption zu schützen. Ein kleiner Speicherbereich (typischerweise 128 MiB) auf der Partition ist reserviert, um wichtige Schreiboperationen durchzuführen. Aus Performancegründen werden Daten nicht sofort auf den Datenträger geschrieben, sondern zunächst im Arbeitsspeicher gehalten. Erst nach einer zeitlichen Verzögerung werden sie auf den Datenträger geschrieben. Dieses Verfahren wird als Caching bezeichnet und ermöglicht ein wesentlich schnelleres Arbeiten. Kommt es währenddessen zu einem Systemabsturz, ist es unsicher, ob die Daten bereits auf den Datenträger geschrieben wurden. Um diesen ungewissen Zustand zu vermeiden, werden Journale verwendet. Wird beispielsweise eine neue Datei angelegt, werden die Änderungen zunächst in das Journal geschrieben. Erst wenn dieser Vorgang vollständig abgeschlossen ist, werden die Daten auf die Festplatte geschrieben. Kommt es in diesem Fall zu einem Systemabsturz, kann zunächst im Journal geprüft werden, ob der letzte Vorgang erfolgreich abgeschlossen wurde. Fehlt ein entsprechender Eintrag im Journal, werden die Änderungen nicht auf die Festplatte geschrieben, sondern verworfen (vgl. [FB07, Lin15]).

Da dieser Vorgang immer ein zweifaches Schreiben der Daten (einmal in das Journal und ein weiteres Mal auf den Datenträger) benötigen würde, werden im Standardmodus von Ext4 nur Metadaten in das Journal geschrieben. Diese Einstellung kann jedoch durch die Mount-Optionen beeinflusst werden, siehe dazu Abschnitt 5.2.

Das Journal selbst ist eine normale, jedoch versteckte Datei auf dem Dateisystem. Die dazugehörige Inode wird zuvor reserviert, siehe Abschnitt 4.2.4. Da hiervon dennoch abgewichen werden kann, ist die Inode des tatsächlich verwendeten Journals im Super-

block des Dateisystems zu finden. Wie bereits erwähnt beträgt die übliche Größe des Journals 128 MiB, was genau der Größe einer Blockgruppe entspricht. Ein wesentlicher Unterschied im Vergleich zu „normalen“ Dateien im Dateisystem ist, dass sämtliche Journal-Strukturen in Big-Endian-Reihenfolge auf den Datenträger geschrieben werden (vgl. [Lin15]).

5.1. Layout

Das Journal hat einen strukturierten Aufbau. Nach einem eigenen Superblock, der sich immer im ersten Block des Journals befindet, folgen die einzelnen Schreibvorgänge in Form von Transaktionen. Jede Transaktion beginnt mit dem Descriptor Block und endet im Regelfall mit einem Commit Block. Zwischen diesen Blöcken befinden sich die Datenblöcke, die durch diese Transaktion geschrieben werden (vgl. [Lin15]).

Generell hat das Journal des Ext4-Dateisystems folgendes Format:

Abbildung 5.1.: Aufbau des Journals

Unter Ext4 ist es zusätzlich möglich das Journal auszugliedern und auf ein externes Gerät zu verlagern. In diesem Fall steht vor dem Journal Superblock noch der Ext4 Superblock.

5.1.1. Block Header

Jeder Block des Journals, auch der Superblock, beginnt mit einem 12 Byte langen Header und gibt an, welche Daten sich innerhalb dieses Blocks befinden, siehe Tabelle 5.1.

Dabei kennzeichnet *h_magic* den Beginn des Block Headers mit einer eindeutigen Magic Number. Anschließend gibt ein Wert im Attribut *h_blocktype* an, welcher der Typen in diesem Block enthalten ist. Die *h_sequence* ist eine innerhalb des Journals eindeutige Identifikationsnummer der Transaktion.

Offset	Länge	Name	Beschreibung
0x00	32 Bit	h_magic	Magic Number: 0xC03B3998
0x04	32 Bit	h_blocktype	Kennzeichnet den Inhalt des Blocks: 0x1: Descriptor 0x2: Commit 0x3: Superblock Version 1 0x4: Superblock Version 2 0x5: Revocation
0x08	32 Bit	h_sequence	Transaktionsnummer dieses Blocks

Tabelle 5.1.: Aufbau des Block Headers [Lin15]

5.1.2. Superblock

Das Journal besitzt einen eigenen Superblock, der verglichen mit dem Ext4 Superblock wesentlich simpler aufgebaut ist. Er befindet sich immer am Anfang des Journals in Journalblock 0.

Der Superblock beinhaltet Informationen wie Größe eines Journalblocks und Anzahl der Blöcke des Journals. Des Weiteren enthält er dynamische Werte, die das weitere Beschreiben beeinflussen. Die *s_sequence* ist eine fortlaufende Nummer, die immer die nächste zu verwendende Transaktionsnummer enthält. Das Attribut *s_start* enthält die Blocknummer innerhalb des Journals, auf die als nächstes geschrieben wird und *s_errno* enthält einen Wert, der im Falle eines Fehlers gesetzt wird.

Beim Mounten des Dateisystems ist der Fehlerwert im Regelfall 0 und die nächste Transaktion wird beginnend ab Journalblock 1 geschrieben. Wird das Dateisystem anschließend fehlerfrei ausgehängt, werden *s_errno* und *s_start* zurück auf 0 bzw. 1 gesetzt. Folglich werden nach jedem Mounten des Dateisystems alte Transaktionen überschrieben. Die *s_sequence* wird im Gegensatz dazu fortlaufend inkrementiert und nicht zurückgesetzt. Tabelle 5.2 zeigt den Aufbau des Superblocks. Die Gesamtgröße, inklusive Block Header, beträgt 1024 Byte (vgl. [Lin15]).

Offset	Länge	Name	Beschreibung
0x000	12 Byte	s_header	Block Header
Statische Informationen, die das Journal beschreiben:			
0x00C	32 Bit	s_blocksize	Blockgröße des Journals
0x010	32 Bit	s_maxlen	Anzahl Blöcke in diesem Journal
0x014	32 Bit	s_first	Erster Block des Journals
Dynamische Informationen, die den aktuellen Zustand beschreiben:			
0x018	32 Bit	s_sequence	Erste ID innerhalb des Protokolls
0x01C	32 Bit	s_start	Blocknummer des ersten Eintrags
0x020	32 Bit	s_errno	Fehlerwert
Zusätzliche Felder, nur für Version 2 (seit Ext4):			
0x024	32 Bit	s_feature_compat	Flags für kompatible Funktionen
0x028	32 Bit	s_feature_incompat	Flags für inkompatible Funktionen
0x02C	32 Bit	s_feature_ro_compat	Unbenutzt
0x030	16 Byte	s_uuid[16]	16 Byte lange UUID des Journals
0x040	32 Bit	s_nr_users	Anzahl Dateisysteme
0x044	32 Bit	s_dynsuper	Position der Superblock Kopie
0x048	32 Bit	s_max_transaction	Max. Anzahl Journalblöcke pro Trans.
0x04C	32 Bit	s_max_trans_data	Max. Anzahl Datenblöcke pro Trans.
0x050	8 Bit	s_checksum_type	Verwendeter Prüfsummen-Algorithmus
0x051	24 Bit	s_padding2	
0x054	168 Byte	s_padding[42]	
0x0FC	32 Bit	s_checksum	Prüfsumme des Superblocks
0x100	768 Byte	s_users[16*48]	UUID aller Dateisysteme

Tabelle 5.2.: Aufbau des Journal Superblocks [Lin15]

5.1.3. Descriptor Block

Jede Transaktion des Journals startet mit einem Descriptor Block. Dieser beschreibt, welche Datenblöcke auf dem Datenträger verändert werden und besteht im Grunde aus einer Liste von Blockadressen. Ohne den Descriptor Block ist es nicht möglich, die in der Transaktion enthaltenden Datenblöcke zuzuordnen.

Offset	Länge	Name	Beschreibung
0x00	12 Byte	t_header	Block Header
Liste von Blockadressen:			
0x0C	4 Byte	t_blocknr	Adresse des Datenblocks im Dateisystem
0x10	4 Byte	t_flags	Eines der folgenden Kennzeichen: 0x1: Gleiche vier Bytes wie Magic Number 0x2: Gleiche UUID wie zuvor 0x4: Block wurde durch diese Transaktion gelöscht 0x8: Letzter Eintrag, wenn nur ein Eintrag existiert 0xA: Letzer Eintrag, wenn mehr als ein Eintrag exisitiert
0x14	16 Byte	t_uuid	UUID (existiert nicht, wenn das Flag 0x2 gesetzt ist)

Tabelle 5.3.: Aufbau des Descriptor Blocks [Lin15]

Dabei ist zu beachten, dass das *t_uuid* Attribut nur verwendet wird, falls das Flag `0x02` nicht gesetzt wird. Typischerweise wird dieses Attribut nur im ersten Eintrag verwendet und in allen folgenden weggelassen.

Abbildung 5.2 soll den Aufbau des Descriptor Blocks verdeutlichen. Nach dem generellen Block Header folgt der erste Eintrag des Descriptor Blocks. Dieser besteht aus einer Blockadresse und dem Flag-Attribut. In der Regel ist dies im ersten Eintrag nicht gesetzt und dementsprechend folgt die 16 Byte lange UUID. Anschließend folgt der zweite Eintrag mit Blockadresse und Flag, welcher hier `0x02` entspricht. Damit entfällt das *t_uuid* Attribut dieses Eintrags, da es redundant zum ersten Eintrag ist. Gleichermaßen besteht der dritte Eintrag aus Blockadresse und Flag bis schließlich der vierte Eintrag mit Blockadresse und dem Flag `0x0A` folgt, der für das Ende des Descriptor Blocks steht. Dadurch sind die vier, auf den Descriptor Block folgende, Datenblöcke der Transaktion beschrieben und können ihrer ursprünglichen Blockadresse zugeordnet werden.

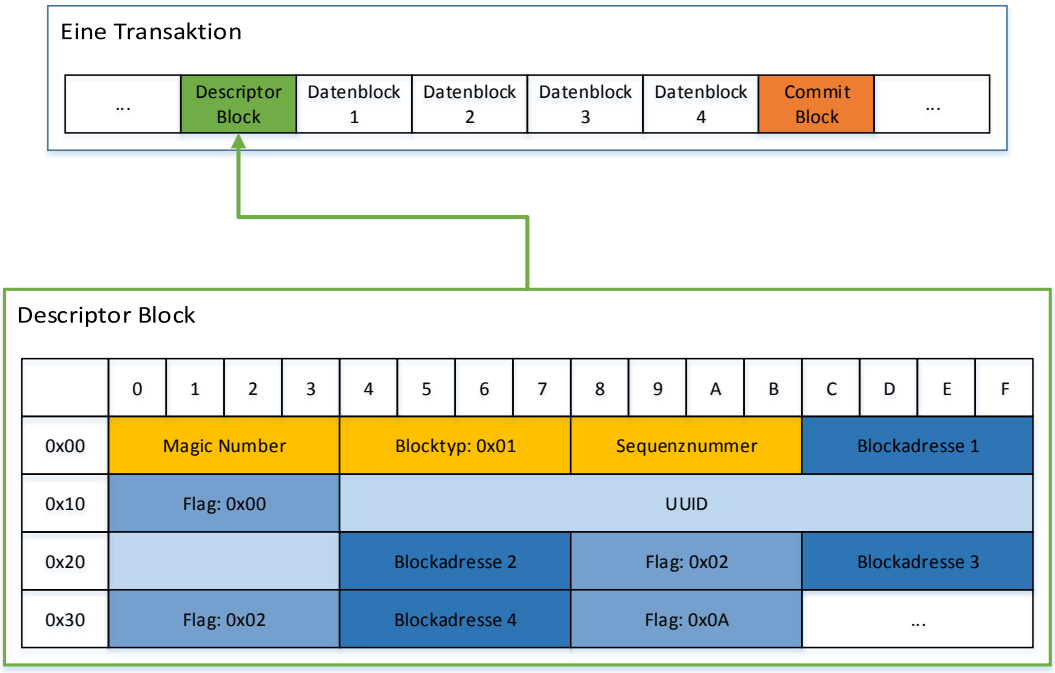


Abbildung 5.2.: Aufbau des Descriptor Blocks

5.1.4. Data Block

Nach dem Descriptor Block folgen alle Datenblöcke, die durch das Journal geschrieben werden. Im Standardmodus des Journals, siehe Abschnitt 5.2, werden nur Metadaten protokolliert. Demnach schließen sich dem Descriptor Block eine Reihe von Datenblöcken an, die unter anderem die Bitmaps und Inode Tables des Dateisystems enthalten.

Hierbei werden alle Datenblöcke unverändert in das Journal geschrieben. Die einzige Ausnahme geschieht, falls die ersten 4 Bytes zufällig der Magic Number des Journals `0xC03B3998` entsprechen. Dann wird das Flag `0x01` im Descriptor Block gesetzt und diese 4 Bytes werden durch Nullen ersetzt (vgl. [Lin15]).

5.1.5. Revocation Block

Ähnlich wie der Descriptor Block, besitzt der Revocation Block eine Sequenznummer und eine Liste von Blockadressen. Sein Zweck ist jedoch ungewollte Änderungen bei einem Wiederherstellungsprozess zu verhindern.

Wird eine Transaktion nicht erfolgreich abgeschlossen, so besitzt sie keinen Commit

Block. Beim erneuten mounten des Dateisystems wird dies festgestellt und ein Revocation Block erstellt. Dadurch wird gewährleistet, dass diese Datenblöcke während einer Wiederherstellung nicht erneut auf die Festplatte geschrieben werden und somit eine Korruption des Dateisystems verursachen (vgl. [Fai12]).

Offset	Länge	Name	Beschreibung
0x00	12 Byte	r_header	Block Header
0x0C	32 Bit	r_count	Anzahl Bytes, die dieser Block benutzt
0x10	32/64 Bit	blocks[0]	Datenblöcke, die nicht wiederhergestellt werden

Tabelle 5.4.: Aufbau des Revocation Blocks [Lin15]

5.1.6. Commit Block

Der Commit Block kennzeichnet das Ende einer Transaktion und gibt dadurch an, dass die Transaktion beendet ist. Anschließend können alle zur Transaktion gehörenden Daten auf ihre endgültige Position auf der Festplatte geschrieben werden.

Offset	Länge	Name	Beschreibung
0x00	12 Byte	header	Block Header
0x0C	1 Byte	h_chksum_type	Verwendeter Prüfsummen-Algorithmus
0x0D	1 Byte	h_chksum_size	Größe der Prüfsumme
0x0E	2 Byte	h_padding	
0x10	32 Byte	h_chksum	Prüfsumme
0x30	8 Byte	h_commit_sec	UNIX-Zeitstempel des Transaktionsendes
0x38	4 Byte	h_commit_nsec	Nanosekunden-Komponente des Zeitstempels

Tabelle 5.5.: Aufbau des Commit Blocks [Lin15]

Der Block Header des Commit Blocks enthält die gleiche Sequenznummer wie der dazugehörige Descriptor Block. Damit wird gekennzeichnet, dass der Descriptor Block und der Commit Block zu derselben Transaktion gehören.

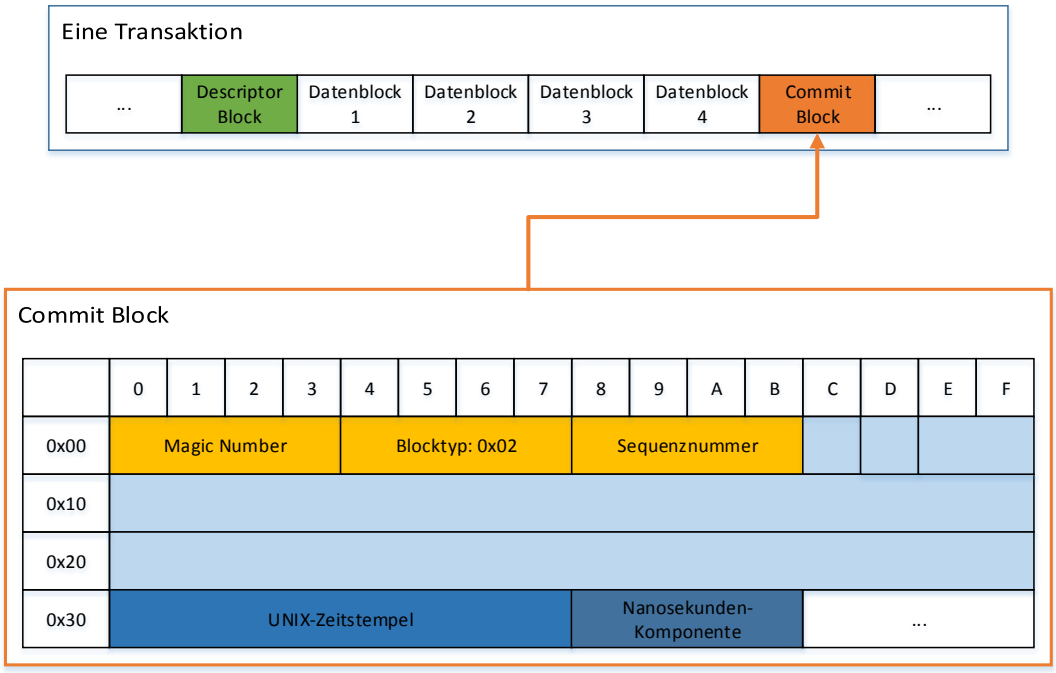


Abbildung 5.3.: Aufbau des Commit Blocks

Abbildung 5.3 verdeutlicht den Aufbau und des Commit Blocks. Eine Transaktion wird durch den Descriptor Block eingeleitet. Dieser enthält die Blockadressen der Datenblöcke, die durch diese Transaktion geschrieben werden. Daraufhin folgen alle Datenblöcke bis der Commit Block das Ende der Transaktion kennzeichnet.

5.2. Konfigurationsmodi

Wie bereits erwähnt kann das Journal in verschiedenen Modi betrieben werden. Beeinflusst wird dies durch die Optionen, die beim Mounten des Dateisystems angegeben werden können. Wird der Modus nicht explizit angegeben, wird das Journal standardmäßig im *ordered mode* betrieben (vgl. [Ker13]). Zur Auswahl stehen die drei folgenden Modi:

writeback mode In diesem Modus werden nur Metadaten protokolliert. Inhaltsdaten werden erst hinterher auf die Festplatte geschrieben. Bei einem Systemabsturz kann das Dateisystem durch die protokollierten Metadaten wiederhergestellt werden, allerdings können die Inhaltsdaten korrupt sein. Dieser Fall tritt ein, wenn die Metadaten protokolliert werden und das System abstürzt, bevor die Inhaltsdaten voll-

ständig auf die Festplatte geschrieben wurden. Dieser Modus bietet keine hohe Sicherheit, jedoch die beste Performance für das Dateisystem (vgl. [Bus13, Ker13]).

ordered mode Hier ist die Reihenfolge umgekehrt zum *writeback mode*. Die Daten werden direkt auf die Festplatte geschrieben, bevor die Metadaten in das Journal übertragen werden. Dadurch können das Dateisystem und die Daten unbeschädigt sein, falls ein Systemabsturz vor erfolgreichem Schreiben des Journals auftritt. Verglichen mit dem *writeback* Modus ist dieser etwas langsamer, jedoch deutlich schneller als der *journal mode* (vgl. [Bus13, Ker13]).

journal mode Sowohl Metadaten als auch Inhaltsdaten werden in diesem Modus zuerst in das Journal geschrieben, bevor sie an ihrer endgültigen Position auf dem Datenträger landen. Dieser ist demnach der einzige Modus, der ein Full-Journaling anwendet. Dadurch kann bei einem Systemabsturz die Konsistenz aller Daten sichergestellt werden. Die Zuverlässigkeit wird damit erhöht, ist jedoch vergleichsweise langsam beim Schreiben, da alle Daten zweifach auf den Datenträger geschrieben werden müssen (vgl. [Bus13, Ker13]).

5.3. Lebenszyklus

Das Journal hat eine, bei der Formatierung festgelegte, Gesamtkapazität von typischerweise 128 MiB. Bei einer Blockgröße von 4096 Bytes entspricht das 32768 Datenblöcken. Abzüglich des ersten Blocks für den Superblock des Journals, bleiben noch 32767 Journalblöcke, die für Transaktionen verwendet werden können. Die erste Transaktion einer Sitzung wird ab Journalblock 1 in das Journal geschrieben. Alle darauffolgenden Transaktionen werden dahinter geschrieben. Sind alle Journalblöcke verbraucht, kommt es zu einem Überlauf. Ausgenommen davon ist der Superblock, der sich immer in Journalblock 0 befindet. Alle restlichen Journalblöcke können demnach als eine Art Ringspeicher angesehen werden, der sich zyklisch selbst überschreibt. Wird eine laufende Sitzung zum Beispiel durch einen Absturz oder simplen Neustart des Systems beendet, wird die nächste Transaktion ab Beginn des Journals, also erneut in Journalblock 1 geschrieben.

Teil (1) der Abbildung 5.4 zeigt zwei Transaktionen (seq: 1 und seq: 2), die hintereinander in das Journal geschrieben wurden. In Teil (2) ist der Beginn einer Transaktion (seq: 8) am Ende des Journals dargestellt. Da das Journal nach dem ersten Datenblock (JBlk: 32767) voll ist, werden die zwei folgenden Datenblöcke in Journalblock 1 und 2 geschrieben. Hier findet also ein Überlauf statt und Teile der ersten Transaktion werden überschrieben.

Abbildung 5.4.: Journal Lebenszyklus

Teil (3) zeigt das Journal nach einem Systemneustart. Die erste Transaktion dieser neuen Sitzung beginnt ab Journalblock 1 und überschreibt damit ebenfalls Blöcke der vorherigen Transaktionen.

Das Journal wird jedes Mal, wenn das Dateisystem ausgehängt und wieder gemounted wird, neu gestartet. Gleichermaßen wird, wenn das Journal voll ist, begonnen die ersten Journalblöcke zu überschreiben. Im Falle einer anstehenden Untersuchung ist aus diesem Verhalten ratsam, das Journal umgehend zu sichern, bevor mögliche Spuren im Journal überschrieben werden.

KAPITEL 6

ANALYSE

Neben den drei Konfigurationsmodi des Journals, ist es in Ext4 erstmals möglich das Journaling vollständig zu deaktivieren. Der Overhead an Daten durch ein aktiviertes Journal ist laut Theodore Ts'o jedoch relativ gering: Bei einer normalen Auslastung beträgt dieser nur etwa 4 bis 12%, je nach Dateioperation (vgl. [Tso09a]). Der durch eine Deaktivierung des Journals gewonnene Geschwindigkeitszuwachs steht demnach in keinem Verhältnis zum Nutzen.

Eine triviale Voraussetzung für die Extraktion von Informationen aus dem Journal ist ein aktiviertes Journal. Der *writeback mode* und der *ordered mode* des Journals unterscheiden sich in der Reihenfolge der Schreibvorgänge. Im Bezug auf die Daten innerhalb des Journals sind diese jedoch identisch: Beide Modi schreiben nur Metadaten. Es liegt auf der Hand, dass der *journal mode* potenziell mehr relevante Informationen bietet, da in diesem Modus auch Inhaltsdaten in das Journal geschrieben werden. Eine Änderung an dem verwendeten Modus muss allerdings zusätzlich konfiguriert werden. Da dies die wenigstens Anwender tun, wird der Standard verwendet. Gleichzeitig bilden die dadurch protokollierten Metadaten auch die Schnittmenge aller Modi. Aufgrund dessen wird in diesem und folgenden Kapiteln von einem Journal im Standardmodus *ordered mode* ausgegangen.

6.1. Hintergrundinformationen

Es ist nötig zu verstehen, wie eine Datei innerhalb des Ext4 Dateisystems gebildet wird, um den Herausforderungen einer Dateiwiederherstellung zu begegnen. Aus diesem Grund

wird zunächst beschrieben, inwiefern Verzeichniseinträge, Inodes und die Daten zusammenhängen.

Jede Datei besteht aus folgenden drei Komponenten:

- Ein **Verzeichniseintrag** ist ein Eintrag in einer Verzeichnisdatei. Dieser bestimmt die Lage der Datei innerhalb des Dateisystems und beinhaltet den Dateinamen sowie eine eindeutige Inode-Nummer.
- In der **Inode** einer Datei sind sämtliche Eigenschaften der Datei hinterlegt. Darunter fallen Informationen, wie die verschiedenen Zeitstempel, Zugriffsrechte und Dateigröße. Insbesondere enthält sie auch die Extents der Datei, mit dessen Hilfe die Datenblöcke der Datei abgebildet werden.
- Die tatsächlichen Inhaltsdaten werden in **Datenblöcken** gespeichert. Je größer die Datei, desto mehr Datenblöcke werden benötigt.

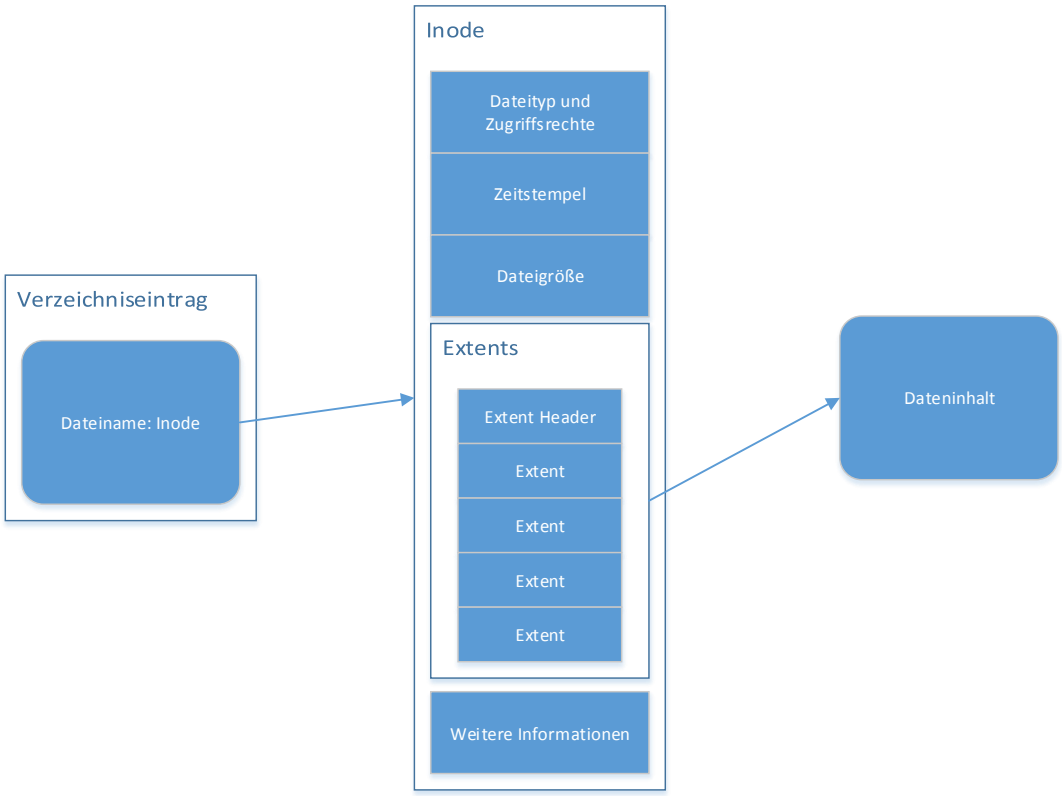


Abbildung 6.1.: Abbildung einer Datei im Dateisystem

Die Beziehung dieser drei Punkte wird in Abbildung 6.1 gezeigt. Um die Darstellung zu vereinfachen, sind die Inhalte der einzelnen Komponenten auf das Wesentliche reduziert.

Anhand des Dateinamens kann der Verzeichniseintrag einer Datei identifiziert werden. Dieser Verzeichniseintrag enthält eine eindeutige Inode-Nummer, die auf die dazugehörige Inode weist. Die Inode ist Dreh- und Angelpunkt jedes Dateizugriffs. Sie verwaltet die Zugriffsrechte, enthält alle benötigten Metadaten und stellt die Verknüpfungen zu den Datenblöcken her.

6.2. Dateilöschung

In älteren Dateisystemen wie Ext2 ist die Wiederherstellung einer gelöschten Datei möglich. Früher konnten diese Dateien gut rekonstruiert werden, da die Dateisysteme nur den Eintrag im Verzeichnis löschten. Die Metadaten, die den Ort der Datenblöcke auf dem Datenträger beschreiben, blieben erhalten. Solange die Inode nicht wiederverwendet oder die Datenblöcke überschrieben wurden, ist eine Rekonstruktion der Datei durchführbar. Mit Ext3 wurde eine andere Vorgehensweise eingeführt, die in Ext4 beibehalten wurde: Informationen über die verwendeten Datenblöcke werden überschrieben.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  A481 0000 0700 0000 774D E354 2D4D E354  ........wM.T-M.T
00000010  2D4D E354 0000 0000 0000 0100 0800 0000  -M.T............
00000020  0000 0800 0100 0000 0AF3 0100 0400 0000  ................
00000030  0000 0000 0000 0000 0100 0000 0084 0000  ................
00000040  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060  0000 0000 8270 7177 0000 0000 0000 0000  ....,pqw........
00000070  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080  1C00 0000 103F A53A 103F A53A 180A C732  .....?.:.?.:...2
00000090  2D4D E354 103F A53A 0000 0000 0000 0000  -M.T.?.:........
```

Listing 6.1: Inode vor dem Löschen

In Fairbanks et al. [FLO10] wird gezeigt, dass die Extents innerhalb der Inode nach dem Löschen mit Nullen überschrieben werden. Um diese Aussage zu verifizieren, ist in Listing 6.1 eine Inode mit einem integrierten Extent-Eintrag hexadezimal dargestellt. Der Extent Header (blau) und der Extent-Eintrag (grün) sind farblich markiert.

Ein anschließendes Löschen der dazugehörigen Datei zeigt, welche Änderungen sich dadurch ergeben. Nachfolgendes Listing 6.2 zeigt dieselbe Inode nach dem Absetzen des

„*rm*“ Befehls. Alle Änderungen, die sich durch das Löschen ergeben haben, sind hier farblich gekennzeichnet. Die Farbe Orange markiert dabei Änderungen der Werte, Rot kennzeichnet überschriebene Informationen.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  A481 0000 0000 0000 774D E354 C74D E354  ........wM.T.M.T
00000010  C74D E354 C74D E354 0000 0000 0000 0000  .M.T.M.T........
00000020  0000 0800 0100 0000 0AF3 0000 0400 0000  ................
00000030  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000040  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000050  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060  0000 0000 8270 7177 0000 0000 0000 0000  ....,pqw........
00000070  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080  1C00 0000 B831 437A B831 437A 180A C732  .....1Cz.1Cz...2
00000090  2D4D E354 103F A53A 0000 0000 0000 0000  -M.T.?.:........
```

Listing 6.2: Inode nach dem Löschen

Der Zeitstempel *DTIME* (`0x14-0x17`) wird durch das Löschen erstmalig gesetzt. Auch die Zeitstempel *CTIME* und *MTIME* (`0x0C-0x0F und 0x10-0x13`), inklusive ihrer Zusätze (`0x84-0x8B`), werden auf den Zeitpunkt des Löschens aktualisiert. Dem hingegen bleibt der Zeitstempel *ATIME* unverändert.

Die Dateigröße (`0x04-0x07`) sowie die Zähler für *Hard Links* (`0x1A-0x1B`) und Datenblöcke (`0x1C-0x1F`) werden nach dem Löschen mit Nullen überschrieben. Ebenso wird der Extent Header auf seinen Ursprung zurückgesetzt. Das heißt, dass die Magic Number und die maximale Anzahl erlaubter Extents bestehen bleiben, die Anzahl der gültigen Extents (`0x2A-0x2B`) jedoch gelöscht wird. Der Extent-Eintrag (`0x34-0x3F`) hingegen wird vollständig überschrieben und lässt somit keine Rückschlüsse über die assoziierten Datenbereiche zu. Die für eine Rekonstruktion relevanten Informationen sind damit vollständig entfernt. Weder ist erkennbar, wie Groß die Datei ursprünglich war, noch wie viele Datenblöcke bzw. Extent-Einträge verwendet wurden.

Extent-Bäume

Wenn vier Extent-Einträge nicht ausreichen, um alle referenzierten Datenbereiche zu beschreiben, wird ein Extent-Baum aufgebaut. Nachfolgende Analyse untersucht das Verhalten des Dateisystems bei dem Löschen dieser Dateien und vergleicht dazu die betrof-

fenen Metadaten. Für diesen Zweck ist in Listing 6.3 zunächst eine Inode dargestellt, die auf einen Extent-Knoten verweist.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  A481 0000 00CA 9A3B 845D E354 885D E354  .......;.].T.].T
00000010  885D E354 0000 0000 0000 0100 70CD 1D00  .].T........p...
00000020  0000 0800 0100 0000 0AF3 0100 0400 0100  ................
00000030  0000 0000 0000 0000 8182 0000 0000 0000  ........,.......
00000040  0080 0000 0078 0000 0008 0100 00F8 0000  .....x..........
00000050  0080 0000 0088 0100 0078 0100 0078 0000  .........x...x..
00000060  0008 0200 CCFB 7B44 0000 0000 0000 0000  ......{D........
00000070  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080  1C00 0000 900A 392C 900A 392C 5C76 46D8  ......9,..9,\vF.
00000090  845D E354 5C76 46D8 0000 0000 0000 0000  .].T\vF.........
```

Listing 6.3: Inode mit Extent-Baum vor dem Löschen

Die Extents in der Inode verweisen nicht direkt auf Datenbereiche der Datei, sondern auf einen Datenblock mit weiteren Extent-Einträgen. Dies ist aus der Interpretation des Extent Headers ersichtlich. Die relevanten Informationen des Headers sind in Listing 6.3 blau markiert: Die Anzahl der gültigen Einträge, die dem Header folgen, beträgt eins und dieser verweist auf einen Extent-Index (*depth = 1*). Obwohl nur der erste Extent-Eintrag (grün) berücksichtigt wird, sind die weiteren Einträge nicht leer, sondern enthalten den zweiten bis vierten Extent des Extent-Knotens.

Der dazugehörige Extent-Knoten ist in nachfolgendem Listing 6.4 abgebildet. Er beinhaltet einen Extent Header, gefolgt von acht gültigen Extent-Einträgen (blau). Diese verweisen nun auf den Datenbereich (*depth = 0*, grün gekennzeichnet) der Datei.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0AF3 0800 5401 0000 0000 0000 0000 0000  ....T...........
00000010  0080 0000 0088 0000 0080 0000 0078 0000  .............x..
00000020  0008 0100 00F8 0000 0080 0000 0088 0100  ................
00000030  0078 0100 0078 0000 0008 0200 00F0 0100  .x...x..........
00000040  0080 0000 0088 0200 0070 0200 0078 0000  .........p...x..
00000050  0008 0300 00E8 0200 0080 0000 0088 0300  ................
00000060  0068 0300 AD51 0000 0008 0400 0000 0000  .h...Q..........
```

Listing 6.4: Extent-Knoten vor dem Löschen

In diesem Beispiel wird untersucht, welche Änderungen sich nach dem Löschen in der Inode und in dem Extent-Knoten ergeben. Hierfür wird die dazugehörige Datei, wie zuvor, mit dem *„rm“* Befehl gelöscht. Änderungen an den Zeitstempeln, der Dateigröße, den Hard Links und der Anzahl verwendeter Datenblöcke werden dieses Mal außer Betracht gelassen, da sich im Vergleich zu Listing 6.2 nichts geändert hat. Interessant sind die Änderungen innerhalb der Extents sowie des Extent-Knotens.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  A481 0000 0000 0000 845D E354 F75D E354  .........].T.].T
00000010  F75D E354 F75D E354 0000 0000 0000 0000  .].T.].T........
00000020  0000 0800 0100 0000 0AF3 0000 0400 0000  ................
00000030  0000 0000 0000 0000 8182 0000 0000 0000  .........,......
00000040  0080 0000 0078 0000 0008 0100 00F8 0000  .....x..........
00000050  0080 0000 0088 0100 0078 0100 0078 0000  .........x...x..
00000060  0008 0200 CCFB 7B44 0000 0000 0000 0000  ......{D........
00000070  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080  1C00 0000 108D 4656 108D 4656 5C76 46D8  ......FV..FV\vF.
00000090  845D E354 5C76 46D8 0000 0000 0000 0000  .].T\vF.........
```

Listing 6.5: Inode mit Extent-Baum nach dem Löschen

Wie zuvor wird der Extent Header zurückgesetzt. Die Anzahl der gültigen Einträge wird demnach gelöscht. Das Attribut *eh_depth* (`0x2E-0x2F`) des Headers, das im ersten Beispiel ohnehin 0 war und daher keine Änderung feststellen ließ, wurde ebenfalls auf Null gesetzt. Im Gegensatz zu dem vorherigen Fall bleiben die integrierten Extent-Einträge jedoch unverändert (in Listing 6.5 in grün gekennzeichnet).

Als nächstes wird der Extent-Knoten auf Änderungen untersucht. Listing 6.6 zeigt diesen nach dem Löschen der Datei.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0AF3 0000 5401 0000 0000 0000 0000 0000  ....T...........
00000010  0000 0000 0000 0000 0080 0000 0000 0000  ................
00000020  0000 0000 00F8 0000 0000 0000 0000 0000  ................
00000030  0078 0100 0000 0000 0000 0000 00F0 0100  .x..............
00000040  0000 0000 0000 0000 0070 0200 0000 0000  .........p......
00000050  0000 0000 00E8 0200 0000 0000 0000 0000  ................
00000060  0068 0300 0000 0000 0000 0000 0000 0000  .h..............
```

Listing 6.6: Extent-Knoten nach dem Löschen

Wie in den anderen Fällen wird der Header zurückgesetzt und lässt daher nicht mehr erkennen, wie viele gültige Einträge folgen. Allerdings werden die hier enthaltenden Extent-Einträge nicht vollständig überschrieben. Die ersten 4 Byte eines jeden Eintrags, die sozusagen die Nummerierung der Datenbereiche enthalten, bleiben vorhanden. Die restlichen 8 Byte der Extent-Einträge werden überschrieben.

Erkenntnisse

Die ersten zwei Analysen zeigen die Änderungen, die sich nach dem Löschen einer Datei auf die Metadaten der Inode ergeben. Hier sind zwei Fälle zu unterscheiden: Eine Inode mit integrierten Extents wird anders behandelt als eine Inode mit einem Extent-Baum. Während die Extents im ersten Fall vollständig überschrieben werden, bleiben diese im zweiten Fall unverändert.

Dies öffnet im zweiten Fall Möglichkeiten zur Interpretation. Sind die Extents nicht überschrieben, kann davon ausgegangen werden, dass hier vor dem Löschen auf einen Extent-Knoten verwiesen wurde. Eine Untersuchung der Extent-Einträge könnte demnach zu dem Extent-Knoten führen. Wie in Listing 6.6 gezeigt, werden hier jedoch die für eine Rekonstruktion relevanten Informationen überschrieben. Die verbleibenden Block-Nummerierungen innerhalb des Knotens bieten lediglich einen Hinweis auf die Anzahl verwendeter Datenblöcke, mit dessen Hilfe die Größe der Datei abgeschätzt werden kann.

Zusammengefasst ist eine Rekonstruktion gelöschter Dateien unter Ext4 ohne weitere Informationsquellen im Endeffekt nicht möglich.

6.3. Journal Analyse

Die Untersuchung der gelöschten Dateien hat ergeben, dass weitere Informationen benötigt werden, um eine Datei wiederherzustellen. Wie in Kapitel 2 aufgeführt, wurde das Journal bereits als potenzielle Quelle für gelöschte Informationen identifiziert. Der nächste Schritt besteht somit aus einer Analyse des Journals und der darin enthaltenen Daten.

6.3.1. Dateierstellung

Bei dem Erstellen einer simplen Datei werden Abbildung 6.1 zufolge mindestens drei Datenblöcke verändert: Der Verzeichniseintrag, die Inode Table und der Inhalt der Datei.

Wird ein neuer Datenblock belegt bzw. eine neue Inode verwendet, werden die entsprechenden Bitmaps aktualisiert. Zusätzlich existieren innerhalb des Superblocks und der GDT Zähler über freie Datenblöcke und Inodes.

Anhand eines minimalen Beispiels soll zunächst Untersucht werden, welche Informationen sich tatsächlich im Journal auffinden lassen. Zu diesem Zweck wird mit Hilfe der Kommandozeile eine Textdatei auf einer Ext4 Partition erstellt:

```
# echo "Das ist eine Testdatei" > /media/ext4/testdatei.txt
```

Listing 6.7: Erstellen einer Textdatei

Um darzustellen was diese Schreiboperation im Journal bewirkt hat, kann das TSK-Tool *jls* genutzt werden. *Jls* listet den Inhalt des Journals in einem menschenlesbaren Format auf. Listing 6.8 zeigt die Ausgabe des Tools nach Erstellung der Textdatei.

```
# jls /dev/sdb

JBlk    Description
8:      Allocated Descriptor Block (seq: 4)
9:      Allocated FS Block 657
10:     Allocated FS Block 1
11:     Allocated FS Block 673
12:     Allocated FS Block 8865
13:     Allocated FS Block 0
14:     Allocated Commit Block (seq: 4, sec: 1417686660.3725720832)
15:     Allocated Descriptor Block (seq: 5)
16:     Allocated FS Block 642
17:     Allocated FS Block 1
18:     Allocated FS Block 673
19:     Allocated Commit Block (seq: 5, sec: 1417686690.2021293824)
20:     Allocated FS Block Unknown
```

Listing 6.8: Journal nach Erstellen einer Datei

Durch das Erstellen der Textdatei aus Listing 6.7 wurden zwei neue Transaktionen in das Journal geschrieben. Die erste Transaktion (seq: 4) wird in Journalblock 8 beschrieben und beinhaltet fünf Datenblöcke. Die zweite Transaktion (seq: 5) beginnt bei Journalblock 15 und enthält drei Datenblöcke. Beide Transaktionen werden jeweils mit einem Commit Block abgeschlossen.

Wie in Kapitel 5 beschrieben enthält der Descriptor Block alle Blockadressen, die im Rahmen einer Transaktion durch das Journal verändert werden. Geht der Descriptor Block einer Transaktion verloren oder wird überschrieben, können die Datenblöcke ihrer Position auf dem Datenträger nicht mehr zugeordnet werden. *Jls* nutzt diese Informationen ebenfalls, um Journalblöcke ihrer Blockadressen zuzuordnen. Ungeachtet dessen zeigt Listing 6.11 den Inhalt des ersten Desciptor Blocks in Form eines Hexdumps. Durch die Beschreibung der Struktur des Descriptor Blocks in Abschnitt 5.1.3 kann dieser interpretiert werden.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00008000  C03B 3998 0000 0001 0000 0004 0000 0291  .;9.............
00008010  0000 0000 0000 0000 0000 0000 0000 0000  ................
00008020  0000 0000 0000 0001 0000 0002 0000 02A1  ................
00008030  0000 0002 0000 22A1 0000 0002 0000 0000  ......".........
00008040  0000 000A 0000 0000 0000 0000 0000 0000  ................
00008050  0000 0000 0000 0000 0000 0000 0000 0000  ................
[Entfernt]
```

Listing 6.9: Hexdump eines Descriptor Blocks

Nach dem 12 Byte langen Block Header folgt eine Liste von Blockadressen und Flags. Die Blockadressen dieser Einträge sind im obigen Listing fett markiert. Folglich stimmt dies mit der Ausgabe von *fls* überein und die Transaktion beinhaltet fünf Datenblöcke, die unmittelbar nach dem Descriptor Block folgen.

JBlk	Hexdump	Blockadresse	Beschreibung
9	0x0291	657	Inode Bitmap
10	0x0001	1	Group Descriptor Table
11	0x02A1	673	Inode Table
12	0x22A1	8865	Datenblock
13	0x0000	0	Superblock

Tabelle 6.1.: Übersicht - Transaktion 4

Eine einfache Möglichkeit die Zugehörigkeit der Blockadressen zu ermitteln bietet das Tool *fsstat*, das ebenfalls im TSK enthalten ist. Dieses Tool listet Dateisystem Informationen auf, indem es den Superblock und die GDT ausliest, siehe Anhang A.

Der erste in das Journal geschriebene Datenblock ist die Inode Bitmap. Ein Vergleich mit der Inode Bitmap vor dem Erstellen der Testdatei zeigt, dass sich genau ein Byte verändert hat, nämlich durch das Belegen einer neuen Inode. Aus dem gleichen Grund wird die GDT im zweiten Block dieser Transaktion geändert: Sie enthält Zähler für den Status der verwendeten Inodes jeder Blockgruppe. Wesentlich interessanter sind die Änderungen in der Inode Table. Da die Textdatei im Wurzelverzeichnis des Dateisystems erstellt wurde, finden sich die ersten Änderungen in Inode 2. Durch das Angelegen eines neuen Verzeichniseintrags im Wurzelverzeichnis wurden die Zeitstempel der Inode 2 aktualisiert und die Inode-Version inkrementiert. Anschließend wurde in der Inode Table eine neue Inode angelegt: Inode 12. Diese ist nach Inode 11, welche für das *„lost+found“* Verzeichnis vergeben wurde, die erste verfügbare Inode.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000B00  A481 0000 0000 0000 7E2E 8054 7E2E 8054  ........~..T~..T
00000B10  7E2E 8054 0000 0000 0000 0100 0000 0000  ~..T............
00000B20  0000 0800 0100 0000 0AF3 0000 0400 0000  ................
00000B30  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B40  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B50  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B60  0000 0000 5A00 AE61 0000 0000 0000 0000  ....Z..a........
00000B70  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B80  1C00 0000 F8EE 55E5 F8EE 55E5 F8EE 55E5  ......U...U...U.
00000B90  7E2E 8054 F8EE 55E5 0000 0000 0000 0000  ~..T..U.........
00000BA0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BB0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BC0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BD0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BE0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BF0  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Listing 6.10: Inode 12 innerhalb Journalblock 11

Die Größe einer Inode beträgt 256 Byte. Aus diesem Grund ist Inode 12 ab Offset `0xB00` $(11 * 256)$ des aktuellen Journalblocks zu finden.

Die ersten 2 Byte der Inode enthalten einen codieren Dateimodus. `0x81A4` steht für *„Regular File“*, mit Lese- und Schreibrechten für den Besitzer, Leserechte für die Gruppe und Leserechte für Andere (vgl. [Lin15]). Aus Performancegründen werden die Datenblöcke in Ext4 verzögert alloziert, infolgedessen ist die Dateigröße zu diesem Zeitpunkt noch

Null. Ebenso folgen dem Extent Header noch keine Extent-Einträge (vgl. [KCSD08]). Ansonsten enthält die Inode hauptsächlich Zeitstempel und zusätzliche Kennzeichen, die für eine forensische Untersuchung von geringem Interesse sind.

Der nächste Datenblock dieser Transaktion beinhaltet den Verzeichniseintrag. Anhand der Definition eines Verzeichniseintrags in Abschnitt 4.3 kann der folgende Hexdump ebenfalls gedeutet werden.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0200 0000 0C00 0102 2E00 0000 0200 0000  ................
00000010  0C00 0202 2E2E 0000 0B00 0000 1400 0A02  ................
00000020  6C6F 7374 2B66 6F75 6E64 0000 0C00 0000  lost+found......
00000030  D40F 0D01 7465 7374 6461 7465 692E 7478  ....testdatei.tx
00000040  7400 0000 0000 0000 0000 0000 0000 0000  t...............
00000050  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000060  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000070  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000080  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090  0000 0000 0000 0000 0000 0000 0000 0000  ................
[Entfernt]
```

Listing 6.11: Verzeichniseintrag nach Dateierstellung

Die ersten zwei Einträge in einem Verzeichnis sind immer „.“ (aktuelles Verzeichnis) und „..“ (übergeordnetes Verzeichnis). Dem folgt der Eintrag für das *„lost+found“* Verzeichnis mit einer Länge von `0x14` Bytes, das beim Erstellen des Dateisystems angelegt wurde. Der Verzeichniseintrag für die Testdatei beginnt demnach bei Offset `0x0C` + `0x0C` + `0x14` = `0x2C` dieses Datenblocks, der in Tabelle 6.2 gedeutet wird.

Offset	Hexdump	Wert	Beschreibung
0x2C	0C00 0000	12	Inode 12
0x30	D40F	4052	Nächster Eintrag bei 0x2C + **0xFD4** = 0x1000 (letzter Eintrag in diesem Datenblock)
0x32	0D	13	Dateiname hat 13 Zeichen
0x33	01	1	„Regular File“
0x34	7465...74	testdatei.txt	Dateiname

Tabelle 6.2.: Verzeichniseintrag der Testdatei

Der fünfte und letzte Datenblock innerhalb dieser Transaktion beinhaltet Änderungen des

Superblocks, die bezüglich der Testdatei jedoch nicht von weiterem Interesse sind.

Die zweite Transaktion mit der Sequenznummer 5 wurde ebenfalls zur Erstellung der Testdatei erzeugt. Hierbei werden drei Datenblöcke geschrieben:

JBlk	Blockadresse	Beschreibung
16	642	Data Bitmap
17	1	Group Descriptor Table
18	673	Inode Table

Tabelle 6.3.: Übersicht - Transaktion 5

Für den Inhalt der Testdatei wird ein Datenblock benötigt. Blockadresse 642 enthält die Data Bitmap der zweiten Blockgruppe des Dateisystems, der durch das Allozieren eines Datenblocks verändert wurde. Dies erklärt auch die erneuten Änderungen innerhalb der GDT. Die GDT beinhaltet neben den Zählern für die Inodes auch welche für freie Datenblöcke. Der dritte Datenblock beinhaltet erneut die Inode Table, der die Inode der Testdatei enthält. In Listing 6.12 sind die Änderungen gegenüber Listing 6.10 hervorgehoben, die nun durch die neue Transaktion geschrieben wurden.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000B00  A481 0000 1700 0000 7E2E 8054 7E2E 8054  ........~..T~..T
00000B10  7E2E 8054 0000 0000 0000 0100 0800 0000  ~..T............
00000B20  0000 0800 0100 0000 0AF3 0100 0400 0000  ................
00000B30  0000 0000 0000 0000 0100 0000 8182 0000  ..............,..
00000B40  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B50  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B60  0000 0000 5A00 AE61 0000 0000 0000 0000  ....Z..a........
00000B70  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B80  1C00 0000 F8EE 55E5 F8EE 55E5 F8EE 55E5  ......U...U...U.
00000B90  7E2E 8054 F8EE 55E5 0000 0000 0000 0000  ~..T..U.........
00000BA0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BB0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BC0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BD0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BE0  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000BF0  0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Listing 6.12: Inode 12 innerhalb Journalblock 18

Die tatsächliche Größe der Textdatei ist jetzt in Byte `0xB04-0xB07` eingetragen. Darüber hinaus steht in Byte `0xB1C-0xB1F` die Anzahl benötigter 512 Byte großen Blöcke. Demnach $8 * 512 = 4096$ Byte. In Byte `0xB28-0xB64` der Inode befinden sich die Extents mit einer Gesamtgröße von 60 Bytes. Um hierauf noch einmal verdeutlicht einzugehen zeigt Abbildung 6.2 den Aufbau und Inhalt dieses speziellen Extents.

Extent

	0	1	2	3	4	5	6	7	8	9	A	B
Header	0xF30A		1 Eintrag		Max. 4 Einträge		Zeigt auf Datenblock		Ungenutzt			
Extent 1	Nummer des ersten Blocks: 0				Anzahl Blöcke: 1		Blockadresse: 33409					
Extent 2												
Extent 3												
Extent 4												

Abbildung 6.2.: Extents der Inode 12

Die ersten 12 Bytes beinhalten den Header, der die Anzahl benutzter Einträge enthält und angibt, ob die Extents auf einen weiteren Extent oder direkt auf Datenblöcke zeigen. Anschließend folgen bis zu vier Extents mit einer Größe von jeweils 12 Byte, wovon in diesem Fall nur einer verwendet wird. Der Extent selbst enthält die Anzahl der Blöcke, die durch diesen Extent abgedeckt werden. Hinzu kommt die Nummer und die Anfangsadresse des verlinkten Datenbereichs.

6.3.2. Dateiänderung

Nachdem die Testdatei erstellt wurde und die damit verbundenen Auswirkungen auf das Journal gezeigt wurden, zeigt dieser Abschnitt welche Metadaten durch eine Änderung der Testdatei im Journal erfasst werden. Die Dateiänderung umfasst hierbei eine Umbenennung des Dateinamens, die durch folgenden Befehl realisiert wird:

```
# mv /media/ext4/testdatei.txt /media/ext4/umbenannt.txt
```

Listing 6.13: Umbenennung der Testdatei

Wie schon bei der Dateierstellung ist *jls* das geeignete Tool, um die Änderungen des Journals aufzuzeigen:

```
# jls /dev/sdb

JBlk    Description
20:     Allocated Descriptor Block (seq: 6)
21:     Allocated FS Block 673
22:     Allocated FS Block 8865
23:     Allocated Commit Block (seq: 6, sec: 1417687429.4107850752)
```

Listing 6.14: Journal nach einer Dateiänderung

Die Ausgabe des Tools *jls* (Listing 6.14) zeigt die durch die Umbenennung erzeugte Transaktion (seq: 6) innerhalb des Journals.

JBlk	Blockadresse	Beschreibung
21	673	Inode Table
22	8865	Datenblock

Tabelle 6.4.: Übersicht - Transaktion 6

Die exakten Änderungen, die sich durch diese Transaktion ergeben haben, lassen sich durch einen Vergleich der Journalblöcke 18 und 21 evaluieren. Zur Erinnerung: Journalblock 18 enthält ebenfalls Datenblock 673, siehe Tabelle 6.3. Es wird somit die aktuelle Inode Table mit der Inode Table vor der Umbenennung verglichen. Dieser Vergleich zeigt, dass zwei Änderungen in der Inode Table stattgefunden haben. Die erste Änderung betrifft die Inode des Wurzelverzeichnisses. Durch das Umbenennen der Testdatei wurden die *CTIME* und *MTIME* sowie die Version der Inode aktualisiert.

Die zweite Änderung betrifft die Inode der Testdatei. Listing 6.15 zeigt Inode 12 nach der Umbenennung. Alle Änderungen, im Vergleich zur Inode 12 in Listing 6.12, sind fett gedruckt. Da sich an den Inhaltsdaten nichts geändert hat und nur der Dateiname verändert wurde, wurde in der Inode nur der Zeitstempel *CTIME* aktualisiert.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000B00  A481 0000 1700 0000 7E2E 8054 8031 8054  ........~..T.1.T
00000B10  7E2E 8054 0000 0000 0000 0100 0800 0000  ~..T............
00000B20  0000 0800 0100 0000 0AF3 0100 0400 0000  ................
00000B30  0000 0000 0000 0000 0100 0000 8182 0000  ..............,..
00000B40  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B50  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B60  0000 0000 5A00 AE61 0000 0000 0000 0000  ....Z..a........
00000B70  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B80  1C00 0000 149D 8D29 F8EE 55E5 F8EE 55E5  .......)..U...U.
00000B90  7E2E 8054 F8EE 55E5 0000 0000 0000 0000  ~..T..U.........
[Entfernt]
```

Listing 6.15: Inode 12 innerhalb Journalblock 21

Der andere Datenblock, der von der Dateiänderung betroffen ist, beinhaltet den Verzeichniseintrag der Testdatei. In Byte `0x1C-0x1D` wurde die Länge des *„lost+found“* Verzeichniseintrags auf 44 Byte erhöht, der somit den alten Verzeichniseintrag der Testdatei bedeutungslos macht. Anschließend folgt ein neuer Verzeichniseintrag für Inode 12, der den neuen umbenannten Dateinamen enthält.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0200 0000 0C00 0102 2E00 0000 0200 0000  ................
00000010  0C00 0202 2E2E 0000 0B00 0000 2C00 0A02  ............,...
00000020  6C6F 7374 2B66 6F75 6E64 0000 0C00 0000  lost+found......
00000030  1800 0D01 7465 7374 6461 7465 692E 7478  ....testdatei.tx
00000040  7400 0000 0C00 0000 BC0F 0D01 756D 6265  t...........umbe
00000050  6E61 6E6E 742E 7478 7400 0000 0000 0000  nannt.txt.......
[Entfernt]
```

Listing 6.16: Verzeichniseintrag nach Dateiänderung

Anhand dieses Beispiels wird gezeigt, dass sich nach minimalen Änderungen mehrere Kopien einer Inode im Journal befinden. Nach dem aktuellen Stand enthält das Journal Inode 12 dreimal: Zwei Kopien durch die Erstellung und eine Kopie der Inode nach der Änderung.

6.3.3. Dateilöschung

Dieser Abschnitt zeigt, welche Informationen sich nach dem Löschen der Testdatei im Journal auffinden lassen. Zur Löschung dient der Standard Linux Befehl *„rm“*:

```
# rm /media/ext4/umbenannt.txt
```

Listing 6.17: Löschen der Testdatei

Anschließend wird abermals mit *jls* festgestellt, welche Änderungen durch das Journal geschrieben wurden.

```
# jls /dev/sdb

JBlk    Description
24:     Allocated Descriptor Block (seq: 7)
25:     Allocated FS Block 8865
26:     Allocated FS Block 673
27:     Allocated FS Block 0
28:     Allocated FS Block 642
29:     Allocated FS Block 1
30:     Allocated FS Block 657
31:     Allocated Commit Block (seq: 7, sec: 1417688239.1326519040)
```

Listing 6.18: Journal nach der Dateilöschung

Durch das Löschen der Datei werden alle Datenblöcke verändert, die von dem Erstellen und Verändern der Testdatei betroffen waren. Alle Metainformationen, die zuvor geschrieben wurden, werden dementsprechend angepasst.

JBlk	Blockadresse	Beschreibung
25	8865	Datenblock
26	673	Inode Table
27	0	Superblock
28	642	Data Bitmap
29	1	Group Descriptor Table
30	657	Inode Bitmap

Tabelle 6.5.: Übersicht - Transaktion 7

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0200 0000 0C00 0102 2E00 0000 0200 0000  ................
00000010  0C00 0202 2E2E 0000 0B00 0000 E80F 0A02  ................
00000020  6C6F 7374 2B66 6F75 6E64 0000 0C00 0000  lost+found......
00000030  1800 0D01 7465 7374 6461 7465 692E 7478  ....testdatei.tx
00000040  7400 0000 0C00 0000 BC0F 0D01 756D 6265  t...........umbe
00000050  6E61 6E6E 742E 7478 7400 0000 0000 0000  nannt.txt.......
```

Listing 6.19: Verzeichniseintrag nach der Dateilöschung

Ähnlich wie bei der Umbenennung wird die Länge des vorherigen Verzeichniseintrags verändert und auf `0xFE8` gesetzt. Damit ist der Verzeichniseintrag des *„lost+found“* Verzeichnisses der letzte gültige in diesem Datenblock. Im nächsten Datenblock, der die Inode Table enthält, wird wie bei der Erstellung und der Dateiänderung die Inode des Wurzelverzeichnisses aktualisiert. Dazu kommen die Änderungen an der Inode 12, die in Listing 6.20 farblich markiert sind.

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000B00  A481 0000 0000 0000 7E2E 8054 AA34 8054  ........~..T.4.T
00000B10  AA34 8054 AA34 8054 0000 0000 0000 0000  .4.T.4.T........
00000B20  0000 0800 0100 0000 0AF3 0000 0400 0000  ................
00000B30  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B40  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B50  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B60  0000 0000 5A00 AE61 0000 0000 0000 0000  ....Z..a........
00000B70  0000 0000 0000 0000 0000 0000 0000 0000  ................
00000B80  1C00 0000 54C4 2689 54C4 2689 F8EE 55E5  ....T.&.T.&...U.
00000B90  7E2E 8054 F8EE 55E5 0000 0000 0000 0000  ~..T..U.........
[Entfernt]
```

Listing 6.20: Inode 12 innerhalb Journalblock 26

Die Änderungen, die sich an der Inode durch das Löschen der Datei ergeben, sind identisch mit den gezeigten Änderungen in Abschnitt 6.2. Dies sollte lediglich verdeutlichen, dass sämtliche Änderungen an Metadaten durch das Journal geschrieben werden und die Inode in jeder Form im Journal zu finden ist.

Die weiteren vier Datenblöcke, die durch diese Transaktion geändert werden, sind der Superblock, die GDT sowie die Data- und Inode Bitmap. Die Inode 12 sowie die verwen-

deten Datenblöcke werden nach der Löschung freigegeben. Diese Informationen spiegeln sich in den Zählern des Superblocks und der GDT wieder.

6.3.4. Zwischenstand

Durch das relativ ausführliche Beispiel anhand einer simplen Textdatei wurde gezeigt, dass eine Datei nicht zwangsläufig in einem Vorgang auf den Datenträger geschrieben wird. Das Erstellen der Datei in diesem Beispiel benötigte zwei Transaktionen innerhalb des Journals. Auch nach einer Dateiänderung oder Löschung werden alle geänderten Metadaten in das Journal geschrieben. Dies hat zur Folge, dass die Metadaten einer Datei mehrfach im Journal zu finden sind.

Die Inode ist dabei von besonderem Interesse, da sie Informationen bezüglich der verwendeten Datenblöcke enthält und diese für eine Rekonstruktion unabdingbar sind.

6.3.5. Fragmentierung

Die ersten Analysen zeigen die Transaktionen des Journals nach dem Erstellen, Ändern und Löschen einer einfachen Datei. Dabei bestand die Datei aus einen Datenblock für die Inhaltsdaten, somit wäre bei einem anderen Datentyp, zum Beispiel einer Bilddatei, eine Wiederherstellung durch File Carving ohne Weiteres möglich.

Ein interessantes Szenario ist daher die Erstellung einer fragmentierten Datei und die damit verbundenen Auswirkungen auf das Journal.

Listing 6.21 zeigt zwei Transaktionen im Journal, die durch die Erstellung einer fragmentierten Datei geschrieben wurden.

```
# jls /dev/sdb

JBlk    Description
32:     Allocated Descriptor Block (seq: 8)
33:     Allocated FS Block 657
34:     Allocated FS Block 1
35:     Allocated FS Block 673
36:     Allocated FS Block 8865
37:     Allocated FS Block 642
38:     Allocated FS Block 643
39:     Allocated FS Block 644
```

```
40:     Allocated FS Block 645
41:     Allocated FS Block 646
42:     Allocated FS Block 33879
43:     Allocated FS Block 647
44:     Allocated FS Block 648
45:     Allocated Commit Block (seq: 8, sec: 1419777122.887152640)
46:     Allocated Descriptor Block (seq: 9)
47:     Allocated FS Block 649
48:     Allocated FS Block 1
49:     Allocated FS Block 33879
50:     Allocated FS Block 673
51:     Allocated FS Block 650
52:     Allocated Commit Block (seq: 9, sec: 1419777130.2805975296)
```

Listing 6.21: Journal nach Erstellung einer fragmentierten Datei

Wie bereits in Abschnitt 6.2 gezeigt wird bei einer hinreichend fragmentierten Datei ein Extent-Baum erstellt. Werden mehr als vier Extents benötigt, um alle Datenbereiche zu verweisen, wird mindestens ein zusätzlicher Datenblock verwendet, der einen Extent-Knoten enthält. Nach dem Löschen einer solchen Datei werden die Informationen des Extent-Knotens weitestgehend überschrieben. Aus diesem Grund wird für eine erfolgreiche Rekonstruktion, neben der Inode, auch eine alte Kopie des Extent-Knotens benötigt.

JBlk	Blockadresse	Beschreibung
33	657	Inode Bitmap
34	1	Group Descriptor Table
35	673	Inode Table
36	8865	Datenblock
37	642	Data Bitmap
38	643	Data Bitmap
39	644	Data Bitmap
40	645	Data Bitmap
41	646	Data Bitmap
42	33879	Datenblock
43	647	Data Bitmap
44	648	Data Bitmap

Tabelle 6.6.: Übersicht - Transaktion 8

Erwartungsgemäß enthält Transaktion 8 alle Metadaten, die durch das Erstellen der frag-

mentierten Datei geändert wurden. Neben den Bitmaps, der GDT und der Inode Table sind in der ersten Transaktion zwei Datenblöcke enthalten (Jblk 36 und JBlk 42). Ein Blick in die Extents der dazugehörigen Inode, siehe Listing 6.22, zeigt, dass Datenblock 33879 den Extent-Knoten enthält (0x8457 = 33879).

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0AF3 0100 0400 0100 0000 0000 0000 0000  ................
00000010  5784 0000 0000 0000 0080 0000 0048 0000  W............H..
00000020  0038 0100 00C8 0000 0080 0000 0088 0100  .8..............
00000030  0048 0100 0078 0000 0008 0200
```

Listing 6.22: Extents der fragmentierten Datei

Neben der Inode wird demnach also auch der dazugehörige Extent-Knoten (Jblk 42) durch das Journal geschrieben. Damit ist gezeigt, dass alle Informationen, die für eine Wiederherstellung benötigt werden, im Journal zu finden sind.

JBlk	Blockadresse	Beschreibung
47	649	Data Bitmap
48	1	Group Descriptor Table
49	33879	Data Block
50	673	Inode Table
51	650	Data Bitmap

Tabelle 6.7.: Übersicht - Transaktion 9

Mit Transaktion 9 wird die fragmentierte Datei vollständig auf den Datenträger geschrieben. Ein Vergleich der beiden Journalblöcke 42 und 49 zeigt, dass die Datei in zwei Durchläufen auf den Datenträger geschrieben wird. Dadurch enthält der erste Extent-Knoten nicht die gesamten Extents der Datei. Diese werden erst in der zweiten Transaktion vervollständigt (siehe Extent Header in Listing 6.23).

```
Offset(h) 0001 0203 0405 0607 0809 0A0B 0C0D 0E0F

00000000  0AF3 0700 5401 0000 0000 0000 0000 0000  ....T...........
----------------------------------------------------------------------
00000000  0AF3 0A00 5401 0000 0000 0000 0000 0000  ....T...........
```

Listing 6.23: Vergleich der Extent Header in JBlk 42 und Jblk 49

KAPITEL 7

IMPLEMENTIERUNG

Das in diesem Kapitel erarbeitete Konzept basiert auf den Erkenntnissen, die durch das vorherige Kapitel gewonnen wurden. Des Weiteren beschreibt das Konzept die Umsetzung eines kommandozeilenbasierten Programms, das diese Schritte automatisiert. In der abschließenden Evaluierung wird letzten Endes der Mehrwert gegenüber dem File Carving gezeigt.

7.1. Konzeption

Neben der Hauptaufgabe des Journals, das Dateisystem in einem konsistenten Zustand zu halten, dient das Journal als Informationsquelle für vergangene Daten. Alle Änderungen am Dateisystem werden in Form von Transaktionen in das Journal geschrieben und bleiben darin bestehen bis sie überschrieben werden. Anhand der durch die Analyse gewonnen Erkenntnissen wird ein Konzept erarbeitet, das die Rekonstruktion gelöschter Dateien mit Hilfe extrahierter Informationen aus dem Dateisystem Journal ermöglicht.

Der Grundgedanke hierbei ist folgender: Wird eine Datei erstellt, werden die dazugehörigen Metadaten durch das Journal geschrieben. Nach dem Löschen der Datei werden die Metadaten auf dem Dateisystem weitestgehend unbrauchbar gemacht, sodass eine Rekonstruktion ohne weitere Informationen nicht möglich ist. Sofern jedoch noch eine alte Kopie dieser Metadaten im Journal existiert, können diese extrahiert und für die Wiederherstellung verwendet werden.

Alle Daten, die für diesen Zweck benötigt werden, sind die Extents der Datei, die sich

entweder direkt in der Inode oder in einem Extent-Knoten befinden. Die Analyse hat gezeigt, dass in beiden Fällen alle relevanten Datenblöcke durch das Journal geschrieben werden. Da es sich ausschließlich um Metadaten handelt, ist das Konzept somit bei allen drei Konfigurationsmodi (siehe Abschnitt 5.2) des Journals anwendbar.

Um eine gelöschte Datei auf diese Weise zu rekonstruieren, besteht der erste Schritt in dem Ermitteln der Inode-Nummer. Dazu können existierende Anwendungen verwendet werden. Mittels *fls* des TSK werden die Verzeichniseinträge des Dateisystems durchlaufen und sämtliche Einträge, auch diejenigen einer gelöschten Datei, dargestellt. Sofern der entsprechende Verzeichniseintrag nicht überschrieben wurde, kann *fls* die Inode-Nummer der Datei also unproblematisch feststellen.

Im zweiten Schritt wird die ermittelte Inode-Nummer genutzt, um die Adresse des Datenblocks zu berechnen, welcher die referenzierte Inode enthält. Nach dieser Blockadresse kann das Journal anschließend durchsucht werden, um alle Kopien der Inode zu finden. Die Berechnung der Blockadresse ist dabei wie folgt:

1. Da die Anzahl der verfügbaren Inodes bei der Formatierung des Dateisystems festgelegt wird und diese gleichmäßig über alle Blockgruppen verteilt sind, wird zunächst die Blockgruppe der Inode ermittelt. Die benötigen Informationen dazu sind die vorhandene Inode-Nummer und die Anzahl Inodes pro Gruppe, die sich im Superblock befindet.

   $$Blockgruppe = (inode\_num - 1) \ / \ inodes\_per\_group$$

2. Anschließend wird aus der absoluten Inode-Nummer die relative Inode-Nummer innerhalb dieser Blockgruppe berechnet. So ist beispielsweise Inode 8542 bei 8192 Inodes pro Gruppe die 349. Inode der zweiten Blockgruppe.

   $$Index = (inode\_num - 1) \ \% \ inodes\_per\_group$$

3. Zu der ermittelten Blockgruppe wird als nächstes die Anfangsadresse der Inode Table aus der GDT ausgelesen. Mit der relativen Inode-Nummer kann nun der Versatz innerhalb der Inode Table berechnet werden. Dies ist nötig, da sie aus mehreren Datenblöcken besteht. Die Größe der Inode und die Größe eines Datenblocks stehen dabei ebenfalls im Superblock und können bei Bedarf ausgelesen werden.

   $$Versatz = (Index * inode\_size) \ / \ block\_size$$

4. Der letzte Schritt, um die Blockadresse zu ermitteln, ist die Anfangsadresse der Inode Table (Basisadresse) und den Versatz innerhalb dieser zu addieren.

$$Blockadresse = Basisadresse + Versatz$$

Im dritten Schritt des Konzepts werden alle Transaktionen des Journals nach der errechneten Blockadresse durchsucht. Dazu ist es nötig alle Descriptor Blöcke des Journals sequentiell zu durchlaufen, da die Blockadressen nur in diesem enthalten sind und nicht direkt mit einem Journalblock verknüpft sind.

Eine essenzielle Erkenntnis, die aus Kapitel 6 hervorgeht, ist, dass die Journalblöcke Daten von jedem Zeitpunkt enthalten. Auch wenn mit einem Journalblock die richtige Datenquelle gefunden wurde, heißt das nicht, dass diese eine geeignete Kopie der Inode enthält. Es kann demnach sein, dass der identifizierte Journalblock von einem Zeitpunkt vor der Erstellung oder nach der Löschung stammt. Aus diesem Grund ist es nicht ausreichend einen Journalblock mit dem gesuchten Datenblock zu finden, sondern es müssen alle Journalblöcke ermittelt werden, die als potenzielle Quelle für eine Kopie dieser Inode dienen.

Sind alle Journalblöcke ermittelt, die eine Kopie der Inode enthalten, besteht der vierte Schritt aus der Identifizierung der, für die Rekonstruktion am besten geeignetsten Inode. Zu diesem Zweck wird die gesuchte Inode aus jedem gefundenen Journalblock extrahiert und miteinander verglichen. Dabei gilt es eine Inode zu finden, die:

- nicht gelöscht: *DTIME* = *0*,
- möglichst neu: > *CTIME*,
- möglichst groß: > *SIZE* ist.

Der Zeitstempel *DTIME* ist die effektivste Möglichkeit festzustellen, ob eine Inode gelöscht ist oder nicht, da *DTIME* erst gesetzt wird, wenn die Datei gelöscht wird. Eine Kopie der Inode zu finden, die durch das Löschen der Datei stammt, bringt keinen Mehrwert. *CTIME* enthält den Zeitstempel der letzten Änderung der Inode. Existieren mehrere Kopien der Inode im Journal, kann somit die letzte Änderung berücksichtigt werden. Des Weiteren wurde in der Analyse gezeigt, dass es vorkommen kann, dass eine Datei erstellt und in mehreren Schreibvorgängen auf das Dateisystem geschrieben wird. In so einem Fall wird die Inode mehrfach mit den gleichen Zeitstempeln in das Journal geschrieben, die Dateigröße und die Extents sind jedoch erst im letzten Schreibvorgang vollständig. Aus diesem Grund wird bei dem Vergleich der Inodes bei gleichen Zeitstempeln auf die Dateigröße geachtet.

Nach erfolgreicher Auswahl einer geeigneten Inode-Kopie ist der fünfte Schritt die Interpretation der Extents. Hierbei ist eine Fallunterscheidung gefordert:

a) Sind alle Extent-Einträge in der Inode integriert, reicht die gefundene Kopie der Inode aus, um die Datei vollständig zu rekonstruieren.

b) Wird ein Extent-Baum in der Inode verwendet, muss das Journal ein zweites Mal durchsucht werden, um einen Journalblock mit dem Extent-Knoten zu finden. Auch hierbei muss das gesamte Journal durchlaufen werden, um einen geeigneten Journalblock zu finden. Werden mehrere Kopien des Extent-Knotens gefunden, wird derjenige mit den meisten gültigen Einträgen verwendet. Wird kein Journalblock mit einer Kopie der Extent-Knoten gefunden, ist eine vollständige Wiederherstellung unmöglich.

Der sechste und letzte Schritt besteht aus dem herausschreiben aller Datenbereiche, die durch die Extents beschrieben sind. Dabei werden diese der Reihe nach abgearbeitet und alle Datenblöcke in eine neue Datei zusammengestellt. Ist die Größe der rekonstruierten Datei identisch mit der Dateigröße, die in der Inode angegeben ist, so ist die Rekonstruktion aller Wahrscheinlichkeit nach erfolgreich.

Abbildung 7.1 stellt die sechs beschriebenen Schritte des Konzepts in Form eines Programmablaufplan (PAP) dar. Anhand dieser Visualisierung soll die Vorgehensweise dargestellt werden und der grobe Ablauf des entwickelten Prototyps verdeutlicht werden.

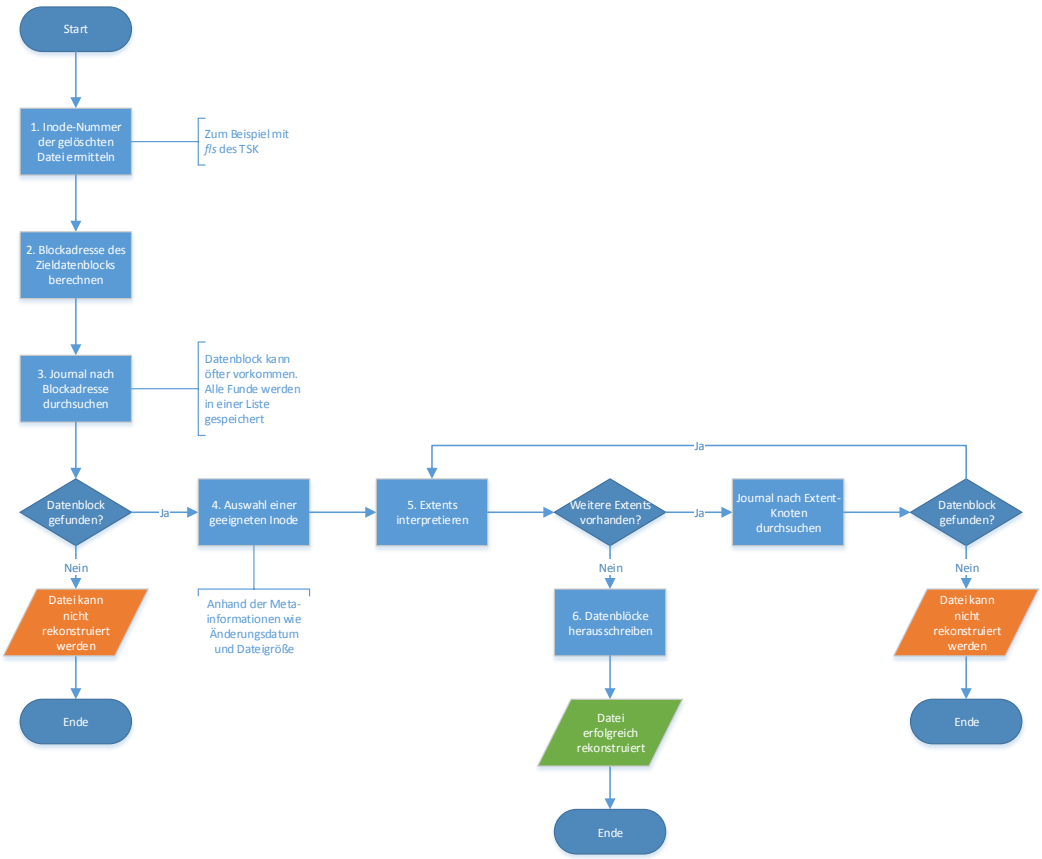


Abbildung 7.1.: Programmablaufplan

7.2. Umsetzung

Die Umsetzung des Konzepts erfolgt dabei vollständig in C++. Unter anderem wurde auch auf darauf geachtet, ausschließlich Standard-Bibliothek zu verwende. Dies ermöglicht eine Kompilierung und anschließende Nutzung des entwickelten Prototyps unter Windows und Linux. In Anlehnung an die Namenskonvention des TSK, heißt das entworfene Programm *JRec*, was für Journal Recovery steht.

Der Aufbau des Programms ist in Abbildung 7.2 in Form eines Klassendiagramms dargestellt. Dieses zeigt die drei Klassen *JRec*, *Filesystem* und *Journal* sowie deren Beziehungen zueinander, die im Abschnitt 7.2.1 näher erläutert werden.

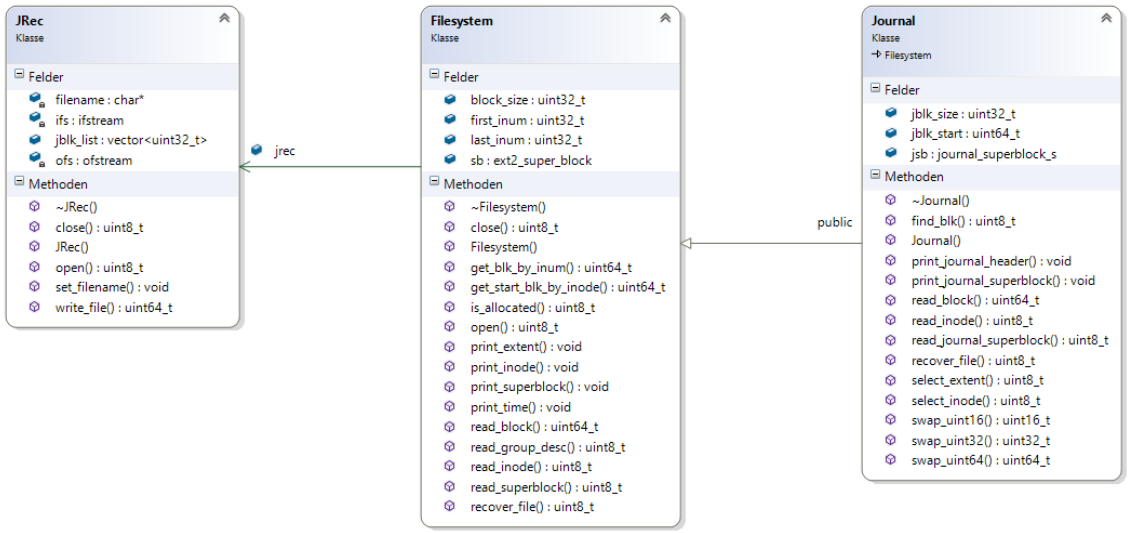


Abbildung 7.2.: Klassendiagramm

7.2.1. Klassen

JRec

Die Klasse *JRec* dient dem speichern globaler Variablen und enthält unter anderem die Ein- und Ausgabestream-Objekte. Das Eingabestream-Objekt ist die Ext4 Partition die im Rahmen dieser Untersuchung analysiert wird. Dabei wird diese nur zum Lesen geöffnet und es können keine Schreibvorgänge oder Änderungen an der Datei durchgeführt werden. Das Ausgabestream-Objekt dient dabei ausschließlich dem Wiederherstellen der gelöschten Datei.

Filesystem

Die Klasse *Filesystem* stellt Methoden zur Verfügung, die zum Auslesen von Daten aus dem Ext4 Dateisystem benötigt werden. Beispielsweise existieren Methoden zum Lesen des Superblocks und der GDT. Diese werden benötigt, um zum Beispiel die Größe und Position einer Inode zu berechnen. Des Weiteren beinhaltet diese Klasse noch Funktionen zum Lesen der Inode und der dazugehörigen Extents. Die vorhandenen Print-Funktionen geben die Inhalte in einem menschenlesbaren Format auf dem Bildschirm aus.

Journal

Journal ist eine abgeleitete Klasse von *Filesystem*. Sämtliche Funktionen von *Filesystem* können auf diese Weise übernommen werden. Spezifische Funktionen werden hier jedoch verändert. So unterscheidet sich die Methode *read_inode* innerhalb beider Klassen. Während im Dateisystem nur eine Inode mit derselben Nummer existieren kann, können sich innerhalb des Journals mehrere Kopien derselben Inode befinden. Die Methode *read_inode* in der Klasse *Journal* benötigt daher zwei Eingabeparameter, den Journalblock und die Inode-Nummer, um eine Inode zu lesen. Weitere erwähnenswerte Methoden sind hier *select_inode* und *select_extent*, welche für die Auwahl der geeigneten Inodes und Extents innerhalb des Journals zuständig sind.

7.3. Evaluation

Die Evaluierung des Konzeptes wird mit Hilfe der entwickelten Anwendung *JRec* und einer vorbereiteten Ext4 Partition durchgeführt. Um den forensischen Mehrwert des Journals aufzuzeigen, werden die Ergebnisse mit den File Carving Tools *Scalpel* und *PhotoRec* verglichen. Obgleich die Motivation für das Dateisystem Ext4 vor allem der verbreitete Einsatz in Android Geräten ist, besteht die Testumgebung aus einer Ubuntu 14.04 LTS 64-Bit-Maschine mit dem 3.13 Linux Kernel. Dies ermöglicht eine flexiblere Vorbereitung des Evaluations-Datenträgers.

7.3.1. Vorbereitung des Datenträgers

Wie bereits erwähnt ist File Carving eine bewährte Methode zur Wiederherstellung gelöschter Dateien. Eine Herausforderung ist jedoch die Rekonstruktion fragmentierter Da-

teien. Aus diesen Grund wird eine Evaluationsumgebung bestehend aus einer Ext4 Partition mit überwiegend fragmentierten Dateien erzeugt. Ausgangspunkt für diese Partition ist eine 4 GB große Datei, die zuvor mit der Anwendung *dd* erstellt wurde. Diese wird mit dem Befehl *mke2fs* und den folgenden Optionen formatiert: „-F -t ext“. Dies erzeugt eine Ext4 Partition mit Standardwerten und einer Blockgröße von 4096 Byte. Um den Datenträger anschließend mit fragmentierten Dateien zu beschreiben, muss im Vorfeld geklärt werden, wie eine Fragmentierung von Dateien in Ext4 erzwungen werden kann.

1. Am Anfang jeder Blockgruppe befindet sich ein reservierter Bereich für Systemdateien wie zum Beispiel für die Kopien des Superblocks und der GDT. Dadurch existiert in jeder Blockgruppe nur begrenzt zusammenhängender Speicherbereich von üblicherweise 128 MiB. Überschreitet eine Datei diese Größe, wird sie zwangsläufig fragmentiert.

2. Fragmentierung geschieht, wenn das Dateisystem nicht genügend freien Speicherplatz hat und gezwungen ist nicht-zusammenhängende Datenblöcke zu verwenden. Dieses Verhalten kann ausgenutzt werden, indem das Dateisystem mit vielen kleinen Dateien beschrieben wird bis es voll ist. Anschließend werden einzelne Dateien gelöscht, sodass vereinzelt freie Datenblöcke auf dem Dateisystem entstehen. Im weiteren Verlauf kann die gewünschte Datei erstellt werden, welche die Lücken füllt und somit fragmentiert ist.

Da übliche Dateien, wie Dokumente und Bilder, eher kleine Dateigrößen haben, ist hier die zweite Variante für die Evaluierung geeignet. Darüber hinaus wird nur auf diese Weise sichergestellt, fragmentierte Dateien mit einer kleinen Dateigröße reproduzierbar zu erstellen. Demzufolge wird die Partition zunächst eingehängt, um sie anschließend mit 4096 256 kB großen Dateien zu beschreiben. Diese belegen zusammen exakt einen Gigabyte des vorhandenen Speicherplatzes und werden im Weiteren als Fülldaten bezeichnet. Im nächsten Schritt wird eine in etwa 3 GB große Datei auf die Partition geschrieben bis der gesamte Speicherplatz verbraucht ist und die Meldung „Auf dem Gerät ist kein Speicherplatz mehr verfügbar“ ausgegeben wird. Damit ist die Partition vollständig belegt und es können keine weiteren Dateien angelegt werden. Abbildung 7.3 stellt diese Belegung vereinfacht dar.

Soll nun eine neue Datei *a* auf der Partition erstellt werden, muss zunächst Speicherplatz freigegeben werden. Dazu wird berechnet wie viele Fülldaten entfernt werden müssen, um ausreichend freien Speicher zu erwirken. So erfordert eine 2 MB große Datei beispielsweise die Entfernung von acht Fülldaten ($2048\ kB / 256\ kB = 8$). Im Anschluss wird

Abbildung 7.3.: Beschreibung des Datenträgers

zufällig diese Anzahl von Fülldaten ausgewählt und von der Partition gelöscht. Durch diesen Vorgang wird nicht-zusammenhängender freier Speicherplatz geschaffen, der in Abbildung 7.4 veranschaulicht ist.

Abbildung 7.4.: Löschen zufälliger Fragmente

Wird die Datei *a* daraufhin auf den Datenträger kopiert, wird diese zwangsläufig fragmentiert, da nicht genügend zusammenhängender Speicherbereich zur Verfügung steht. Beispielhaft veranschaulichen soll dies Abbildung 7.5, die zeigt, wie Datei *a* in fünf Fragmente aufgeteilt und die frei gewordenen Speicherbereiche des Datenträgers füllt.

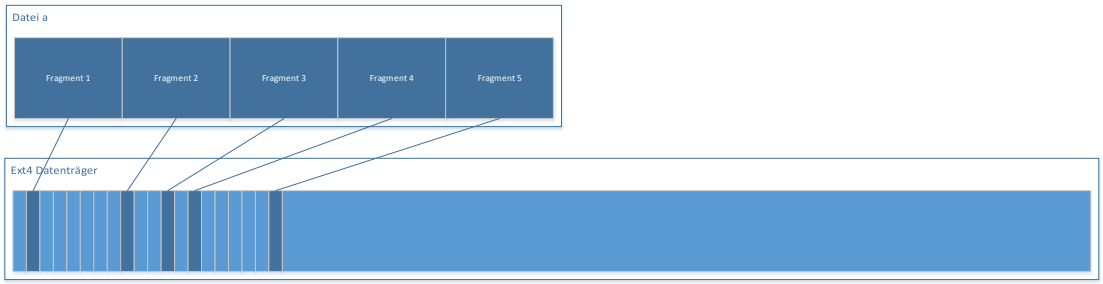


Abbildung 7.5.: Einfügen einer fragmentierten Datei

Das zufällige Löschen der Fülldaten und anschließende Kopieren der Testdatei wird für jedes Bild der Tabelle 7.1 sequenziell durchgeführt. Diese Schritte erlauben eine reproduzierbare Fragmentierung der Testdaten und verteilen die Datenblöcke der Bilder auf verschiedene Blockgruppen des Dateisystems. Die Durchführung erfolgt dabei automatisiert, unter Zuhilfenahme von Skripten, welche im Anhang B zu finden sind.

7.3.2. Testdaten

Zur besseren Veranschaulichung der Ergebnisse handelt es sich bei den Testdaten um Bilder. Dadurch können nicht nur vollständige, sondern teilweise auch Fragmente einer rekonstruierten Datei abgebildet werden.

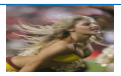
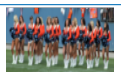
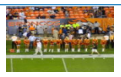
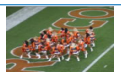
Name	Größe	SHA-256	Thumbnail
01.jpg	3086779 Bytes (2 MB)	`533A6D723E...ACEB90C41D`	
02.jpg	3070563 Bytes (2 MB)	`D5BD3C18BA...BDC8D53FA0`	
03.jpg	2729190 Bytes (2 MB)	`954FAA9046...07D90FA085`	
04.jpg	3056790 Bytes (2 MB)	`72DE4A3BC8...2C6334B0B1`	
05.jpg	1215533 Bytes (1 MB)	`546643ACBD...E4FAF05F7B`	
06.jpg	3378669 Bytes (3 MB)	`E491309DEE...BFEF86CEBA`	
07.jpg	1369559 Bytes (1 MB)	`BAAC73D461...5A86C6AF41`	
08.jpg	8993096 Bytes (8 MB)	`A741DE14ED...27209DA3BD`	
09.jpg	9103266 Bytes (8 MB)	`CDC4A4E1F5...DB9494FD87`	
10.jpg	8984977 Bytes (8 MB)	`BE8E1E2A32...71D89E8DA1`	
11.jpg	6328129 Bytes (6 MB)	`A10BA341D8...391BFD72EB`	
12.jpg	4180920 Bytes (3 MB)	`7289ED00F1...56E1CD8A99`	
13.jpg	3778516 Bytes (3 MB)	`B8116B6F78...9C32670585`	
14.jpg	3547686 Bytes (3 MB)	`7C5C5F3D99...D442DE2550`	
15.jpg	7000635 Bytes (6 MB)	`CFADB29410...9E17771C14`	

Tabelle 7.1.: Testbilder mit Prüfsumme

Tabelle 7.1 enthält eine Übersicht über die in der Evaluation verwendeten Originalbilder inklusive ihrer Dateigröße, der *SHA-256*-Prüfsumme und eines Vorschaubildes. Die

Prüfsumme kann nach der Durchführung verwendet werden, um die Integrität einer Datei festzustellen. Durch Anwendung der zuvor beschriebenen Schritte sind die Testbilder vorwiegend fragmentiert auf der Partition abgelegt. Dass nicht alle Bilder fragmentiert sind, erklärt sich folgendermaßen: 5% des Speicherplatzes werden beim Erstellen der Ext4 Partition für das System reserviert. Unter anderem auch, um Fragmentierung zu vermeiden (vgl. [Tso09b]). Nachdem die Fülldaten für das erste zu kopierende Bild gelöscht wurden ist wieder mehr Speicherplatz verfügbar, die 95%-Grenze ist nicht erreicht und das Dateisystem findet genügend zusammenhängende Datenblöcke, um das Bild nicht fragmentiert abzulegen. Dies gelingt bei den ersten fünf Bildern noch erfolgreich, danach werden jedoch keine zusammenhängenden Datenblöcke gefunden und eine Fragmentierung der restlichen Bilder ist nicht zu vermeiden.

Um den Grad der Fragmentierung festzustellen, kann das Tool *Filefrag* genutzt werden. *Filefrag* ist Bestandteil der *Ext2/3/4 Filesystem Utilities (e2fsprogs)*, welches Auskunft über die Fragmentierung von Dateien gibt (vgl. [Tso11]). Listing 7.1 zeigt dessen Ausgabe für die Testbilder nach dem Kopieren auf den Datenträger.

```
# filefrag *.jpg

/media/ext4/01.jpg: 1 extent found
/media/ext4/02.jpg: 1 extent found
/media/ext4/03.jpg: 1 extent found
/media/ext4/04.jpg: 1 extent found
/media/ext4/05.jpg: 1 extent found
/media/ext4/06.jpg: 8 extents found
/media/ext4/07.jpg: 5 extents found
/media/ext4/08.jpg: 35 extents found
/media/ext4/09.jpg: 36 extents found
/media/ext4/10.jpg: 35 extents found
/media/ext4/11.jpg: 25 extents found
/media/ext4/12.jpg: 16 extents found
/media/ext4/13.jpg: 13 extents found
/media/ext4/14.jpg: 14 extents found
/media/ext4/15.jpg: 27 extents found
```

Listing 7.1: Anzahl der Extents pro Testdatei

7.3.3. File Carving

Die herkömmliche Methode gelöschte Dateien wiederherzustellen ist die Verwendung von File Carving Tools. Dabei wird der gesamte Datenträger, unabhängig vom Dateisystem, einschließlich dem nicht-allozierten Speicherbereich, auf bekannte Dateimuster untersucht.

Scalpel

Da es sich bei den Testdateien ausschließlich um Bilder handelt, kann die Konfiguration des File Carving Tools minimiert werden, sodass dieser nur nach JPG-Dateien sucht. Die Konfiguration des Tools *Scalpel*[5] sieht dementsprechend folgendermaßen aus:

```
extension       cs      size            header          footer
jpg             y       200000000       \xff\xd8\xff    \xff\xd9
```

Listing 7.2: Ausschnitt aus scalpel.conf

Dadurch sucht *Scalpel* ausschließlich nach der hexadezimalen Zeichenfolge `0xffd8ff` (Header). Wird eine solche Folge erkannt, werden alle folgenden Zeichen bis zu der Zeichenfolge `0xffd9` (Footer) herausgeschrieben. Falls kein Footer gefunden wird, wird nach maximal 200000000 Zeichen (ca. 200 MB) abgebrochen.

Das Ergebnis nach Ausführung von *Scalpel* ist in Abbildung 7.6 abgebildet. Insgesamt konnten 31 Bilder gefunden werden. Von diesen 31 Bildern können nur 15 ordnungsgemäß angezeigt werden, wobei es sich ausnahmslos um Thumbnails handelt, also um eingebettete Miniaturbilder. Bei den restlichen Bildern handelt es sich um Bildfragmente, die nur teilweise bzw. gar nicht dargestellt werden können (siehe dazu Anhang C).

[5] https://github.com/sleuthkit/scalpel

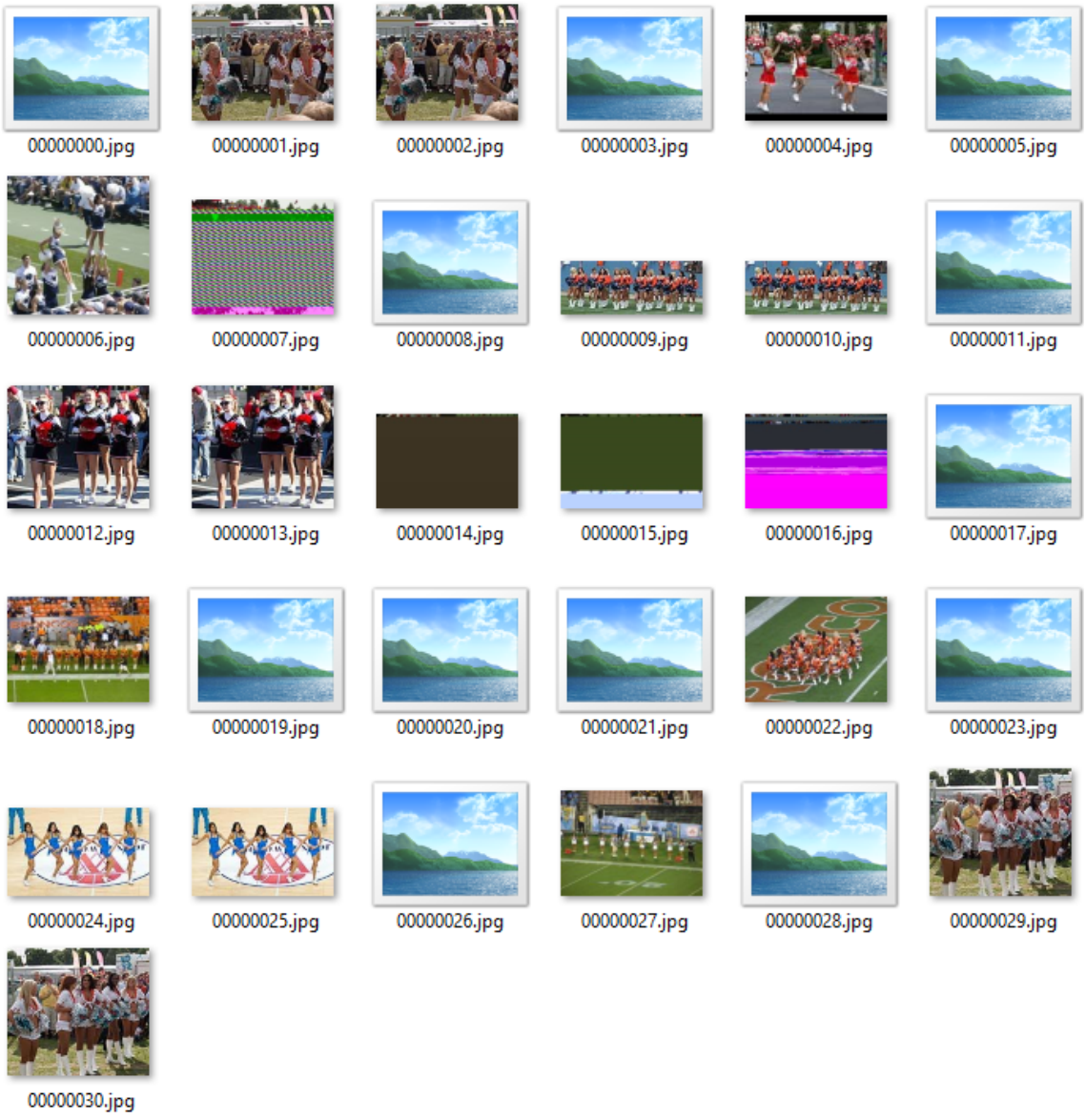


Abbildung 7.6.: Wiederhergestellte Dateien Scalpel, Teil 1

Da *Scalpel* beim ersten Durchlauf hauptsächlich eingebettete Thumbnails gefunden hat, wird *Scalpel* ein zweites Mal mit der Option *-e* ausgeführt. Diese Option verspricht einen besseren Umgang mit eingebetteten Dateien des gleichen Typs. Nach Entfernung der redundanten Bildern aus dem ersten *Scalpel* Vorgang, bleiben folgende vier Bilder übrig:

Abbildung 7.7.: Wiederhergestellte Dateien Scalpel, Teil 2

Dieses Mal konnten vier Bilder vollständig rekonstruiert werden. Die Sicherstellung, dass die gefundenen Bilder und Originale identisch sind, kann durch die Berechnung von Prüfsummen gewährleistet werden.

```
sha256sum *.jpg

954faa90 4698c1df 24d619ae ... b3d28415 b1a21f07 d90fa085 00000000.jpg
546643ac bdde7933 12e609e0 ... b36dc183 05e179e4 faf05f7b 00000005.jpg
533a6d72 3edb48df 8bd13fba ... d1acf842 07c6cfac eb90c41d 00000019.jpg
d5bd3c18 ba44f6fa a2598a0a ... 8276f96f cb8687bd c8d53fa0 00000021.jpg
```

Listing 7.3: Prüfsummen der von Scalpel rekonstruierten Bilder

Ein Vergleich der berechneten Prüfsummen (Listing 7.3) mit den Originalen aus Tabelle 7.1 zeigt, dass es sich bei den vier Bildern um *01.jpg, 02.jpg, 03.jpg und 05.jpg* handelt. Die vollständigen Audit-Dateien beider Scalpel Durchläufe sind im Anhang C zu finden.

PhotoRec

Als zweites File Carving Tool kommt *PhotoRec*[6] zum Einsatz. Wie aus dem Namen hervorgeht, wurde das Tool ursprünglich entworfen, um Bilder zu rekonstruieren, unterstützt inzwischen jedoch weit mehr Dateiformate. Im Unterschied zu *Scalpel* wird die Konfiguration des Tools vorwiegend über eine geführte Oberfläche vorgenommen.

[6] http://www.cgsecurity.org/wiki/PhotoRec

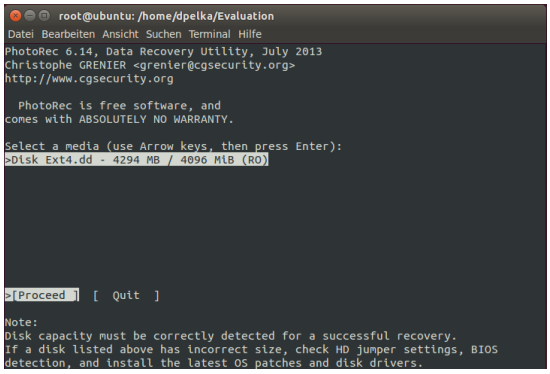


Abbildung 7.8.: Auswahl des Datenträgers

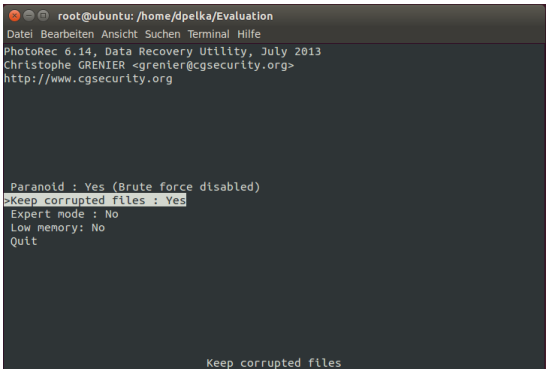


Abbildung 7.9.: Fragemente beibehalten

Abbildung 7.8 und 7.9 zeigen die Auswahl des Datenträgers sowie die Option, dass beschädigte Daten beibehalten werden. Durch diese Einstellung werden die Fragmente der Bilder nicht verworfen.

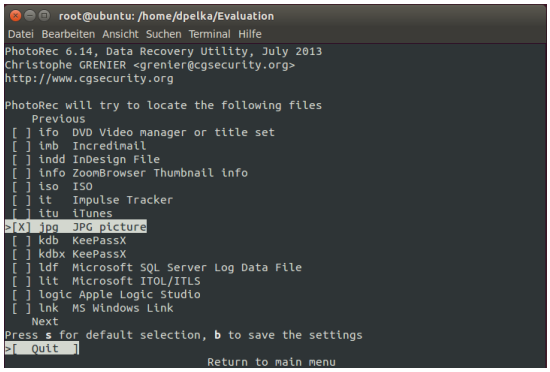


Abbildung 7.10.: Nur JPG-Dateien suchen

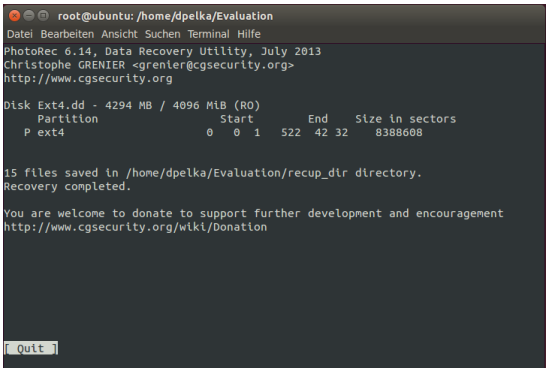


Abbildung 7.11.: Vorgang abgeschlossen

Abbildung 7.10 zeigt die Datenformate nach denen gesucht werden soll. Ähnlich wie *Scalpel* kann hier die Einstellung ebenfalls auf „jpg“ gesetzt werden, um die Laufzeit zu minimieren. Das Ergebnis von *PhotoRec* wird in Abbildung 7.11 gezeigt. Insgesamt stellt *PhotoRec* fünfzehn Dateien her, was der Anzahl der Testbilder entspricht. Bei genauer Betrachtung der rekonstruierten Bilder (siehe Abbildung 7.12) fällt auf, dass es sich bei den fünfzehn Bildern um fünf vollständige Bilder und zehn Fragmente bzw. korrupte Bilder handelt. Eine erneute Berechnung der Prüfsummen (siehe Listing 7.4) bestätigt, dass die fünf vollständig rekonstruierten Bilder die ersten fünf nicht-fragmentierten Bilder der Originaldateien sind.

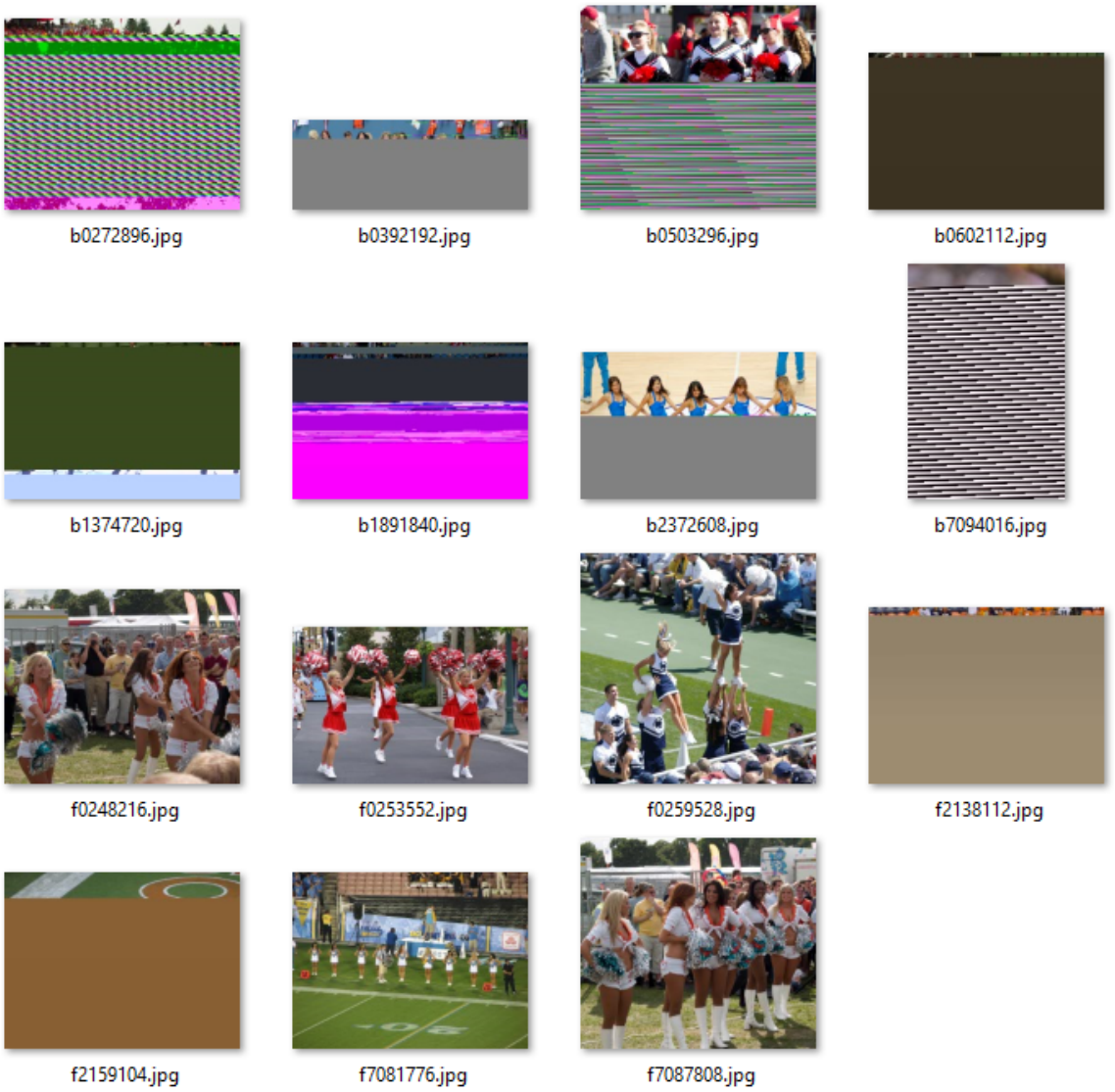


Abbildung 7.12.: Wiederhergestellte Dateien PhotoRec

```
sha256sum *.jpg

954faa904698c1df24d619ae...b3d28415b1a21f07d90fa085 f0248216.jpg
72de4a3bc8eb2744d8de8269...2e0f8678daed722c6334b0b1 f0253552.jpg
546643acbdde793312e609e0...b36dc18305e179e4faf05f7b f0259528.jpg
533a6d723edb48df8bd13fba...d1acf84207c6cfaceb90c41d f7081776.jpg
d5bd3c18ba44f6faa2598a0a...8276f96fcb8687bdc8d53fa0 f7087808.jpg
```

Listing 7.4: Prüfsummen der von PhotoRec rekonstruierten Bilder

7.3.4. Journal Recovery

Gemäß dem Konzept aus Abschnitt 7.1 nutzt *JRec* das Dateisystem Journal, um alte Kopien der Inode zu finden. Anhand diesen Informationen kann der ursprüngliche Speicherort auf dem Dateisystem ermittelt werden. Um *JRec* jedoch einsetzen zu können, ist es zunächst erforderlich die Inode-Nummer der gelöschten Bilder zu identifizieren. Für diesen Zweck wird das Tool *fls* des TSK eingesetzt. Mit den Optionen *-r* und *-d* wird das Dateisystem rekursiv nach gelöschten Verzeichniseinträgen durchsucht. Ein *„grep -i .jpg“* filtert die Ausgabe auf alle JPG-Dateien.

```
# fls -rd Ext4.dd | grep -i .jpg

r/r * 122:      01.jpg
r/r * 136:      02.jpg
 ...
r/r * 147:      14.jpg
r/r * 148:      15.jpg
```

Listing 7.5: Inode-Nummer und Dateiname der gelöschten Bilder

Die Ausgabe des Tools zeigt alle fünfzehn Bilddateien inklusive ihrer Inode-Nummer. Diese Informationen können an *JRec* als Parameter in folgender Form übergeben werden: *jrec image inode -o name*. Am Beispiel des ersten Bildes würde der Befehl zum Rekonstruieren der Datei dementsprechend: *„jrec Ext4.dd 122 -o 01.jpg“* lauten.

Auf diese Weise kann *JRec* für alle weiteren Daten eingesetzt werden. Um diesen Vorgang jedoch nicht manuell fünfzehn Mal ausführen zu müssen, kann dieser Vorgang unter Zuhilfenahme eines Skripts automatisiert werden.

```
# /bin/bash/

image="/home/dpelka/Evaluation/Ext4.dd"

outputdir="jrec_dir"
mkdir $outputdir

fls -d -r -p $image | grep -i ".jpg" > $outputdir/filelist.txt

for i in $(grep "r/r" $outputdir/filelist.txt
  | sed 's/[[:space:]]//g'|sed 's/r\/r//') do
  inode=$(echo $i | awk -F ":" '{print $1}'| sed 's/*//')
```

```
  name="$(echo $i | awk -F ":" '{print $2}'| sed 's/\//SLASH/g')"
  jrec $image $inode -o $outputdir/$name
done
```

Listing 7.6: Bash-Skript zur Rekonstruktion aller JPG-Dateien mittels JRec

Das in Listing 7.6 dargestellte Skript automatisiert diesen Vorgang, indem es die Ausgabe von *fls* weiterverarbeitet und die passenden Parameter an *jrec* übergibt. Dabei würde das Script nicht nur die fünfzehn Bilder, sondern alle JPG-Dateien rekonstruieren, die *fls* ausgibt. In dieser Form dient es demnach dem gleichen Zweck wie die Konfiguration der File Carver *Scalpel* und *PhotoRec*. Die mit *JRec* und dem Skript rekonstruierten Bilder sind in Abbildung 7.13 dargestellt. Alle fünfzehn Bilder konnten erfolgreich wiederherstellt werden. Dies zeigt auch die Prüfsummenberechnung (siehe Listing 7.7). Mit dessen Hilfe kann belegt werden, dass es sich bei den rekonstruierten Bilder um die Originaldaten handelt.

```
# sha256sum *.jpg

533a6d723edb48df8bd13fba...d1acf84207c6cfaceb90c41d 01.jpg
d5bd3c18ba44f6faa2598a0a...8276f96fcb8687bdc8d53fa0 02.jpg
954faa904698c1df24d619ae...b3d28415b1a21f07d90fa085 03.jpg
72de4a3bc8eb2744d8de8269...2e0f8678daed722c6334b0b1 04.jpg
546643acbdde793312e609e0...b36dc18305e179e4faf05f7b 05.jpg
e491309dee9fa5ebccd36068...dd638079e1dc76bfef86ceba 06.jpg
baac73d46139093ceff0ebfd...067affda65ffd65a86c6af41 07.jpg
a741de14ed504df77834a938...3c5375796d1ac427209da3bd 08.jpg
cdc4a4e1f5264106dd19d992...fb99d4925d1153db9494fd87 09.jpg
be8e1e2a32ab0f4ea0e61a80...6d26ca2e5ca49d71d89e8da1 10.jpg
a10ba341d881b30911a142ed...02c85d66210406391bfd72eb 11.jpg
7289ed00f1eb9543666da669...9fca212e3a753856e1cd8a99 12.jpg
b8116b6f7867da622c3d53f2...a7be31ab1a844f9c32670585 13.jpg
7c5c5f3d99712b8a7014204a...648d00028c1037d442de2550 14.jpg
cfadb29410de6c0084442abd...ab0c90401dee359e17771c14 15.jpg
```

Listing 7.7: Prüfsummen der von JRec rekonstruierten Bilder

Abbildung 7.13.: Wiederhergestellte Dateien JRec

7.3.5. Ergebnis

Die Evaluierung im Vergleich zu File Carving Tools zeigt, dass File Carver tatsächlich sehr gut mit nicht-fragmentierten Dateien (Bild 1-5, siehe Listing 7.1) umgehen können. Wie erwartet konnte jedoch kein fragmentiertes Bild wiederhergestellt werden. Eine nähere Betrachtung des vierten Bildes zeigt, dass die Zeichenfolge des JPG-Footers (`0xFFD9`) dreimal im Bild enthalten ist. Aus diesem Grund war *Scalpel* nicht in der Lage das vierte Bild *04.jpg* erfolgreich zu rekonstruieren. *PhotoRec* hingegen ging mit diesem Phänomen erfolgreicher um.

Mit *JRec* ist es dagegen möglich alle fünfzehn Bilder wiederherzustellen. Sowohl die nicht-fragmentierten als auch die fragmentierten Bilder konnten eins-zu-eins rekonstruiert werden, was durch den Vergleich der Prüfsummen erwiesen wurde.

Name	**Rekonstruierte Bilder**	**Prozent**	**Laufzeit**
Scalpel	4/15	27%	34 Sek.
PhotoRec	5/15	33%	20 Sek.
JRec	15/15	100%	3 Sek.

Tabelle 7.2.: Vergleich File Carving mit JRec

Dem ist allerdings hinzuzufügen, dass die Werte in Tabelle 7.2 nicht repräsentativ für eine allgemeine Bewertung der verglichenen Methoden sind. Die Tabelle soll lediglich die Anzahl rekonstruierter Bilder und die Laufzeit der Anwendung innerhalb dieser Evaluation zusammenfassen.

KAPITEL 8

FAZIT

In diesem Kapitel findet die Schlussbetrachtung über die vorliegende Arbeit statt. Es werden die Ergebnisse diskutiert und eine Zusammenfassung wiedergegeben. Überdies wird ein Ausblick gegeben, um zu zeigen an welcher Stelle an diese Arbeit angeknüpft werden kann.

8.1. Ergebnisdiskussion

Grundsätzlich wurde gezeigt, dass mit Hilfe des Journals gelöschte Dateien innerhalb einer Ext4 Partition rekonstruiert werden können. Dies betrifft nicht nur Dateien, die nicht fragmentiert sind. Prinzipiell ist es möglich, stark fragmentierte Dateien vollständig wiederherzustellen. Durch den Einsatz des erarbeiteten Konzepts und des im Rahmen dieser Arbeit entwickelten Tools *JRec*, lassen sich gelöschte fragmentierte Dateien auf Basis extrahierter Informationen aus dem Journal rekonstruieren. Dies stellt gegenüber herkömmlicher Methoden, wie dem File Carving, einen erheblichen Mehrwert dar. Hierbei ist anzumerken, dass File Carving dadurch nicht ersetzt werden kann. Die Unabhängigkeit gegenüber der zugrundeliegenden Dateisysteme machen File Carver unentbehrlich. Allerdings wird mit Tools wie *JRec* eine weitere Möglichkeit aufgezeigt, die unter entsprechenden Umständen bessere Ergebnisse liefert.

Eine erfolgreiche Rekonstruktion aus dem Dateisystem Journal ist jedoch abhängig von drei nicht beeinflussbaren Faktoren.

Der erste Faktor ist die Ermittlung der Inode-Nummer der gelöschten Datei. Da der Ver-

zeichniseintrag gelöschter Dateien nicht entfernt wird, ist dies durch Tools wie *fls* möglich. Wird eine neue Datei im gleichen Verzeichnis wie der gelöschten Datei erstellt, kann dieser Eintrag überschrieben werden.

Der zweite Faktor ist der Inhalt des Journals. Das Journal hat zu jedem Zeitpunkt eine definierte Größe. Wird diese Größe erreicht oder wird das System neu gestartet, überschreibt das Journal alte Inhalte. Das Journal dient daher nur für einen begrenzten Zeitraum als Informationsquelle für gelöschte Daten.

Der letzte Faktor ist der Inhalt der Datenblöcke. Selbst wenn eine alte Inode gefunden wird, die alle nötigen Informationen enthält, kann die entsprechende Datei nicht wiederhergestellt werden, wenn die ursprünglich verwendeten Datenblöcke überschrieben sind.

8.2. Zusammenfassung

In dieser Arbeit wurde ein ausführlicher Einblick in den Aufbau des Ext4 Dateisystems gegeben. Dies umfasste insbesondere die Strukturen von Inodes und Extents, die erforderlich sind, um Dateien zu verwalten. Ebenso wurde der Aufbau des Ext4 Dateisystem Journal detailliert beschrieben und der Lebenszyklus des Journals aufgezeigt. Anhand der Erkenntnisse, die sich durch die umfangreiche Analyse von Dateioperationen, wie dem Erstellen, Ändern und Löschen von Dateien, herauskristallisiert haben, konnte ein forensischer Nutzen des Journals gezeigt werden. Auf Basis dieser Erkenntnisse wurde ein Konzept entwickelt, das ein allgemeines Vorgehen zur Wiedererstellung von gelöschten Dateien anhand Informationen aus dem Journal beschreibt. Dieses diente zugleich auch als Grundlage des entwickelten Prototyps. In der anschließenden Evaluation wurde die prinzipielle Eignung dieser Vorgehensweise gezeigt. Durch die Wiederherstellung fragmentierter Dateien mit Hilfe des entwickelten Tools konnte zugleich ein Mehrwert gegenüber dem File Carving gezeigt werden.

8.3. Ausblick

Bei der Durchführung dieser Arbeit haben sich mehrere Ansatzmöglichkeiten ergeben, die einen forensischen Nutzen des Ext4 Dateisystem Journals aufzeigen. Das prototypisch entwickelte Tool *JRec* rekonstruiert Dateien anhand ihrer Inode-Nummer.

Wurde eine Datei jedoch gelöscht und der Verzeichniseintrag bereits durch neue Einträge überschrieben, lässt sich die Inode-Nummer der gelöschten Datei nicht mehr identifizieren. Des Weiteren ist es in diesem Fall schwer zu sagen, welche Dateien eigentlich gelöscht wurden. Auch wenn sich im Dateisystem keine Informationen mehr über die besagte Datei auffinden lassen, ist es dennoch möglich, dass sich im Journal alte Metadaten befinden und die dazugehörigen Datenblöcke noch nicht überschrieben wurden. Ohne eine Inode-Nummer ist es jedoch nicht möglich, gezielt Informationen aus dem Journal zu extrahieren. Ein anderer Ansatz bestünde folglich darin, das Journal Block für Block zu durchlaufen und jede Inode zu extrahieren, die im Dateisystem nicht alloziert ist. Anhand dieser Inodes könnten gegebenenfalls alte Daten rekonstruiert werden.

Unabhängig davon kann das Journal außerhalb des Standards betrieben werden. Zwei durchaus betrachtenswerte Szenarien sind der Betrieb des Journals auf einem externen Gerät sowie der Konfigurationsmodus *journal mode*, der das Full-Journaling aktiviert. Hier könnte ein Ansatz darin geknüpft werden, dass Daten vollständig aus dem Journal zu rekonstruiert werden, ungeachtet dem Fall, dass die inhaltlichen Daten bereits überschrieben wurden.

ANHANG A

DATEISYSTEM INFORMATIONEN

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext4
Volume Name:
Volume ID: 17970911983c31abd0421c9cc072271f

Last Written at: 2014-11-28 11:42:48 (CET)
Last Checked at: 2014-11-28 11:39:15 (CET)

Last Mounted at: 2014-11-28 11:39:38 (CET)
Unmounted properly
Last mounted on: /mnt

Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode, Dir Index
InCompat Features: Filetype, Extents, Flexible Block Groups,
Read Only Compat Features: Sparse Super, Large File, Huge File,
                           Extra Inode Size

Journal ID: 00
Journal Inode: 8


METADATA INFORMATION
--------------------------------------------
Inode Range: 1 - 655361
```

```
Root Directory: 2
Free Inodes: 655349
Inode Size: 256

CONTENT INFORMATION
-------------------------------------------
Block Groups Per Flex Group: 16
Block Range: 0 - 2621439
Block Size: 4096
Free Blocks: 2541777

BLOCK GROUP INFORMATION
-------------------------------------------
Number of Block Groups: 80
Inodes per group: 8192
Blocks per group: 32768

Group: 0:
Block Group Flags: [INODE_ZEROED, ]
Inode Range: 1 - 8192
Block Range: 0 - 32767
Layout:
Super Block: 0 - 0
Group Descriptor Table: 1 - 1
Group Descriptor Growth Blocks: 2 - 640
Data bitmap: 641 - 641
Inode bitmap: 657 - 657
Inode Table: 673 - 1184
Data Blocks: 8865 - 32767
Free Inodes: 8181 (99%)
Free Blocks: 23897 (72%)
Total Directories: 2
Stored Checksum: 0x764C

Group: 1:
Block Group Flags: [INODE_UNINIT, INODE_ZEROED, ]
Inode Range: 8193 - 16384
Block Range: 32768 - 65535
Layout:
Super Block: 32768 - 32768
Group Descriptor Table: 32769 - 32769
Group Descriptor Growth Blocks: 32770 - 33408
Data bitmap: 642 - 642
```

```
Inode bitmap: 658 - 658
Inode Table: 1185 - 1696
Data Blocks: 33409 - 65535
Free Inodes: 8192 (100%)
Free Blocks: 32127 (98%)
Total Directories: 0
Stored Checksum: 0x12D1

Group: 2:
Block Group Flags: [INODE_UNINIT, BLOCK_UNINIT, INODE_ZEROED, ]
Inode Range: 16385 - 24576
Block Range: 65536 - 98303
Layout:
Data bitmap: 643 - 643
Inode bitmap: 659 - 659
Inode Table: 1697 - 2208
Data Blocks: 65536 - 98303
Free Inodes: 8192 (100%)
Free Blocks: 32768 (100%)
Total Directories: 0
Stored Checksum: 0x830B

Group: 3:
Block Group Flags: [INODE_UNINIT, BLOCK_UNINIT, INODE_ZEROED, ]
Inode Range: 24577 - 32768
Block Range: 98304 - 131071
Layout:
Super Block: 98304 - 98304
Group Descriptor Table: 98305 - 98305
Group Descriptor Growth Blocks: 98306 - 98944
Data bitmap: 644 - 644
Inode bitmap: 660 - 660
Inode Table: 2209 - 2720
Data Blocks: 98945 - 131071
Free Inodes: 8192 (100%)
Free Blocks: 32127 (98%)
Total Directories: 0
Stored Checksum: 0xA3F6

[Entfernt]
```

ANHANG B

SKRIPTE ZUR ERSTELLUNG DER EVALUATIONSUMGEBUNG

```
#!/bin/bash

declare -i i=0
declare -i d=0

# create 256 kB files and group them into directories
for i in {0..4096} do
  if (( $i % 500 == 0 )) then
    d=$d+1
    mkdir $d
  fi

  echo "file_$i" > $d/$i
  fallocate -l 256K $d/$i
  i=$i+1
done

# cause disk full error
dd if=/dev/zero of=file bs=1M count=4000
```

Listing B.1: Bash-Skript zur Beschreibung des Datenträgers

```
#!/bin/bash

typeset -i size
typeset -i blocks

# for every picture in folder
for f in *.jpg do
  # calculate needed blocks
  size=$(stat -c%s "$f")
  blocks=$size/262144
  blocks=$blocks+1

  # delete random blocks
  find /ext4 -type f | shuf -n$blocks | while read ln; do rm $ln; done
  # copy files to filesystem
  cp $f /ext4/$f
done
```

Listing B.2: Bash-Skript zum Kopieren der Testbilder

ANHANG C

SCALPEL AUDIT-DATEIEN

```
Scalpel version 2.1 audit file
Started at Thu Mar 12 17:09:19 2015
Command line:
scalpel -c scalpel.conf Ext4.dd -o scalpel

Output directory: scalpel
Configuration file: /home/dpelka/Evaluation/scalpel.conf

------ BEGIN COPY OF CONFIG FILE USED ------
jpg y   200000000   \xff\xd8\xff    \xff\xd9
------ END COPY OF CONFIG FILE USED ------


Opening target "/home/dpelka/Evaluation/Ext4.dd"

The following files were carved:
File          Start         Chop        Length      Extracted From
00000006.jpg    132878922       NO             9392     Ext4.dd
00000005.jpg    132878336       NO             9978     Ext4.dd
00000004.jpg    129847828       NO             5121     Ext4.dd
00000003.jpg    129818624       NO            13812     Ext4.dd
00000002.jpg    127101864       NO             9687     Ext4.dd
00000001.jpg    127091306       NO             9071     Ext4.dd
00000000.jpg    127086592       NO            13785     Ext4.dd
00000010.jpg    200813738       NO             5845     Ext4.dd
00000009.jpg    200807054       NO             5305     Ext4.dd
00000008.jpg    200802304       NO            10055     Ext4.dd
```

```
00000007.jpg    139722752       NO          61089607     Ext4.dd
00000013.jpg    257704380       NO             13930     Ext4.dd
00000012.jpg    257688430       NO             13930     Ext4.dd
00000011.jpg    257687552       NO             14808     Ext4.dd
00000014.jpg    308281344       NO         113712056     Ext4.dd
00000015.jpg    703856640       NO          82421716     Ext4.dd
00000016.jpg    968622080       NO          93770299     Ext4.dd
00000018.jpg   1094731062       NO              7553     Ext4.dd
00000017.jpg   1094713344       NO              6264     Ext4.dd
00000022.jpg   1105478798       NO              6360     Ext4.dd
00000021.jpg   1105472812       NO             12346     Ext4.dd
00000020.jpg   1105472742       NO             12416     Ext4.dd
00000019.jpg   1105461248       NO             23910     Ext4.dd
00000025.jpg   1214788066       NO              7240     Ext4.dd
00000024.jpg   1214780034       NO              6645     Ext4.dd
00000023.jpg   1214775296       NO             11383     Ext4.dd
00000027.jpg   3625876252       NO              5844     Ext4.dd
00000026.jpg   3625869312       NO             12784     Ext4.dd
00000030.jpg   3628973996       NO             10891     Ext4.dd
00000029.jpg   3628962410       NO             10099     Ext4.dd
00000028.jpg   3628957696       NO             14813     Ext4.dd


Completed at Thu Mar 12 17:10:00 2015
```

Listing C.1: Scalpel Audit-Datei nach erster Ausführung

```
Scalpel version 2.1 audit file
Started at Thu Mar 12 17:08:16 2015
Command line:
scalpel -c scalpel.conf -e Ext4.dd -o scalpel

Output directory: scalpel
Configuration file: /home/dpelka/Evaluation/scalpel.conf

------ BEGIN COPY OF CONFIG FILE USED ------
jpg y   200000000   \xff\xd8\xff    \xff\xd9
------ END COPY OF CONFIG FILE USED ------


Opening target "/home/dpelka/Evaluation/Ext4.dd"

The following files were carved:
```

```
File            Start          Chop        Length      Extracted From
00000006.jpg    132878922       NO             9392     Ext4.dd
00000005.jpg    132878336       NO          1215533     Ext4.dd
00000004.jpg    129847828       NO             5121     Ext4.dd
00000003.jpg    129818624       NO            13812     Ext4.dd
00000002.jpg    127101864       NO             9687     Ext4.dd
00000001.jpg    127091306       NO             9071     Ext4.dd
00000000.jpg    127086592       NO          2729190     Ext4.dd
00000008.jpg    200813738       NO             5845     Ext4.dd
00000007.jpg    200807054       NO             5305     Ext4.dd
00000010.jpg    257704380       NO            13930     Ext4.dd
00000009.jpg    257688430       NO            13930     Ext4.dd
00000011.jpg    308281344       NO        113712056     Ext4.dd
00000012.jpg    703856640       NO         82421716     Ext4.dd
00000013.jpg    968622080       NO         93770299     Ext4.dd
00000015.jpg   1094731062       NO             7553     Ext4.dd
00000014.jpg   1094713344       NO             6264     Ext4.dd
00000016.jpg   1105478798       NO             6360     Ext4.dd
00000018.jpg   1214788066       NO             7240     Ext4.dd
00000017.jpg   1214780034       NO             6645     Ext4.dd
00000020.jpg   3625876252       NO             5844     Ext4.dd
00000023.jpg   3628973996       NO            10891     Ext4.dd
00000022.jpg   3628962410       NO            10099     Ext4.dd
00000021.jpg   3628957696       NO          3070563     Ext4.dd
00000019.jpg   3625869312       NO          3086779     Ext4.dd


Completed at Thu Mar 12 17:08:55 2015
```

Listing C.2: Scalpel Audit-Datei nach dem zweiten Durchlauf

ANHANG D

INHALT DER BEIGELEGTEN CD-ROM

Die dieser Arbeit beigelegte CD-ROM enthält vier Verzeichnisse mit folgendem Inhalt:

Abbildungen Dieses Verzeichnis enthält die in der Arbeit verwendeten Abbildungen in einer höheren Auflösung.

Implementierung Enthält Dateien die im Rahmen der vorliegenden Arbeit entwickelt wurden.

- **JRec** Enthält den Quellcode und das übersetzte Programm.
- **Skripte** Beinhaltet die für die Evaluierung erstellten Bash-Skripte.

Literatur In diesem Verzeichnis befinden sich die verwendeten Internet-Referenzen.

Masterarbeit Enthält die Masterarbeit in elektronischer Form (PDF-Format).

Abkürzungsverzeichnis

BSI	Bundesamt für Sicherheit in der Informationstechnik
Ext2	Second Extended Filesystem
Ext3	Third Extended Filesystem
Ext4	Fourth Extended Filesystem
FAT	File Allocation Table
GDT	Group Descriptor Table
HFS	Hierarchical File System
NTFS	New Technology File System
PAP	Programmablaufplan
RAM	Random-Access Memory
S-A-P	Secure-Analyse-Present
TSK	The Sleuth Kit
UFS	Unix File System
UUID	Universally Unique Identifier
YAFFS	Yet Another Flash File System

Literaturverzeichnis

[Bai14] BAIER, Harald: *Computerforensik - Kapitel 2: Grundlagen der digitalen Forensik*. Hochschule Darmstadt - University of Applied Sciences, 10.04.2014

[Bal14] BALLENTHIN, William: *The Sleuthkit and Ext4*. http://williballenthin.com/ext4/index.html. Erstellt am 02.04.2014, zuletzt besucht am 17.11.2014

[Bib14] BIBLIOGRAPHISCHES INSTITUT GMBH - DUDEN VERLAG: *Duden | Dateisystem | Rechtschreibung, Bedeutung, Definition*. http://www.duden.de/rechtschreibung/Dateisystem. Erstellt am 17.11.2014, zuletzt besucht am 17.11.2014

[Bun11] BUNDESAMT FÜR SICHERHEIT IN DER INFORMATIONSTECHNIK: *Leitfaden „IT-Forensik“*. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Internetsicherheit/Leitfaden_IT-Forensik_pdf.pdf?__blob=publicationFile. Version: 1.0.1, März 2011, zuletzt besucht am 21.10.2014

[Bus13] BUSE, Jarret W.: *Journal File System | Linux.org*. http://www.linux.org/threads/journal-file-system.4136/. Erstellt am 09.07.2013, zuletzt besucht am 14.11.2014

[Car05] CARRIER, Brian: *File System Forensic Analysis*. 1. Aufl. Boston, Mass. : Addison-Wesley, 2005

[Car14a] CARRIER, Brian: *The Sleuth Kit: File- and Volume System Analysis*. http://www.sleuthkit.org/sleuthkit/desc.php. Erstellt am 07.11.2014, zuletzt besucht am 07.11.2014

[Car14b] CARRIER, Brian: *TSK Tool Overview - SleuthKitWiki*. http://wiki.sleuthkit.org/index.php?title=TSK_Tool_Overview. Erstellt am 13.01.2014, zuletzt besucht am 07.11.2014

[Car14c] CARRIER, Brian: *The Sleuth Kit: History*. http://www.sleuthkit.org/sleuthkit/history.php. Erstellt am 25.01.2014, zuletzt besucht am 17.11.2014

[Die09] DIEDRICH, Oliver: Schneller, größer, reifer - Das Linux-Dateisystem Ext4. In: *c't* 2009, Heft 10 (2009), S. 180–183

[Eck04] ECKSTEIN, Knut: Forensics for advanced UNIX file systems. In: *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, 2004, S. 377–385

[Eck05] ECKSTEIN, Knut: Forensische Analyse komplexer Unix-Dateisysteme. In: *GI Jahrestagung (2)*, 2005, S. 658–662

[Fai12] FAIRBANKS, Kevin D.: An analysis of Ext4 for digital forensics. In: *Digital Investigation* 9, Supplement (2012), Nr. 0, S. 118–130. – The Proceedings of the Twelfth Annual {DFRWS} Conference 12th Annual Digital Forensics Research Conference

[FB07] FRANK, Florian ; BRUNS, Jörn: *Journaling Dateisysteme*. http://www.selflinux.org/selflinux/html/dateisysteme_journaling01.html. Erstellt am 09.10.2007, zuletzt besucht am 18.03.2015

[FLO10] FAIRBANKS, Kevin D. ; LEE, Christopher P. ; OWEN, Henry L.: Forensic Implications of Ext4. In: *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. New York, NY, USA : ACM, 2010 (CSIIRW '10), S. 22:1–22:4

[Ges12] GESCHONNECK, Alexander: *Computer-Forensik (iX Edition) - Computerstraftaten erkennen, ermitteln, aufklären*. 5. Aufl. Heidelberg : dpunkt.verlag GmbH, 2012

[KCSD08] KUMAR, Aneesh ; CAO, Mingming ; SANTOS, Jose R. ; DILGER, Andreas: Ext4 block and inode allocator improvements. In: *Proceedings of the Linux Symposium* Bd. 1, 2008, S. 263–274

[Ker13] KERNEL.ORG: *Ext4 Filesystem*. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt. Erstellt am 16.08.2013, zuletzt besucht am 14.11.2014

[KL14] KAART, Marnix ; LARAGHY, S.: Android forensics: Interpretation of timestamps. In: *Digital Investigation* 11 (2014), Nr. 3, S. 234–248. – Special Issue: Embedded Forensics

[KPLL12] KIM, Dohyun ; PARK, Jungheum ; LEE, Keun gi ; LEE, Sangjin: Forensic analysis of android phone using ext4 file system journal log. In: *Future Information Technology, Application, and Service* Bd. 164. Springer, 2012, S. 435–446

[Lin15] LINUX KERNEL ARCHIVES: *Ext4 Disk Layout - Ext4*. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout. Erstellt am 16.03.2015, zuletzt besucht am 18.03.2015.

[Loc30] LOCARD, Edmond: *Die Kriminaluntersuchung und ihre wissenschaftlichen Methoden*. 2. Aufl. Berlin : Kameradschaft Verlagsges., 1930

[Net13] NET APPLICATIONS: *Desktop Operating System Market Share*. http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0. Erstellt am 01.09.2013, zuletzt besucht am 17.03.2015

[Net15] NET APPLICATIONS: *Mobile/Tablet Operating System Market Share*. http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8&qpcustomd=1. Erstellt am 01.02.2015, zuletzt besucht am 17.03.2015

[Spe08] SPENNEBERG, Ralf: Carving-Tools spüren Files auf, ohne das Dateisystem zu kennen kennen. In: *Linux Magazin* 06 (2008)

[SPS07] SWENSON, Christopher ; PHILLIPS, Raquel ; SHENOI, Sujeet: File System Journal Forensics. In: *IFIP International Conference on Digital Forensics*, 2007, S. 231–244

[Ste14] STEINEBACH, Martin: *File Carving*. https://www.sit.fraunhofer.de/de/angebote/kompetenzfelder/itforensics/file-carving/. Erstellt am 2014, zuletzt besucht am 22.12.2014

[Tso09a] TSO, Theodore: *SSD's, Journaling, and noatime/relatime*. http://thunk.org/tytso/blog/2009/03/01/ssds-journaling-and-noatimerelatime/. Erstellt am 01.03.2009, zuletzt besucht am 13.10.2014

[Tso09b] TSO, Theodore: *Re: Reserved block count for Large Filesystem*. http://www.redhat.com/archives/ext3-users/2009-January/msg00026.html. Erstellt am 23.01.2009, zuletzt besucht am 02.03.2015

[Tso10] TSO, Theodore: *Android will be using ext4 starting with Gingerbread*. http://thunk.org/tytso/blog/2010/12/12/android-will-be-using-ext4-starting-with-gingerbread/. Erstellt am 12.12.2010, zuletzt besucht am 21.10.2014

[Tso11] TSO, Theodore: *filefrag(8) - Linux man page*. http://linux.die.net/man/8/filefrag. Erstellt am 01.11.2011, zuletzt besucht am 04.03.2015

[Wik15] WIKIPEDIA: *Big Four (Wirtschaftsprüfungsgesellschaften) — Wikipedia, Die freie Enzyklopädie*. https://de.wikipedia.org/wiki/Big_Four_%28Wirtschaftspr%C3%BCfungsgesellschaften%29. Erstellt am 04.01.2015, zuletzt besucht am 05.02.2015