Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Masterarbeit

Implementierung eines forensischen Ext4-Inode-Carvers

Sabine Seufert

Erlangen, 1. September 2015

Prüfer:	Prof. Dr. Felix Freiling
Betreuer:	Dr. Andreas Dewald

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, 1. September 2015

Sabine Seufert

Zusammenfassung

In der digitalen Forensik, vor allem im Bereich der Datenträgerforensik, spielt die Rekonstruktion von Dateien eine große Rolle. Insbesondere gelöschte oder verloren gegangenen Dateien können essentielle Spuren für forensische Untersuchungen enthalten. Für die Rekonstruktion von Dateien von Datenträgern müssen spezifische Gegebenheiten beachtet werden, wie etwa das verwendete Datei- oder Betriebssystem. Ein Beispiel für ein in Linux-Systemen häufig anzutreffendes Dateisystem stellt das Ext4-Dateisystem dar. Für die Interpretation des Ext4-Dateisystems ist die Integrität des sogenannten Superblocks essentiell, weshalb seine Verfügbarkeit durch auf der Festplatte verteilte, redundante Kopien gesichert wird. Neben dem Superblock spielt die Gruppen-Deskriptor-Tabelle eine wichtige Rolle, da in ihr der grobgranulare Aufbau des Dateisystems enthalten und sie somit unerlässlich für die Rekonstruktion von Inhaltsdaten ist. In dieser Arbeit wird ein Ansatz präsentiert, der mittels verschiedener Suchmuster Dateien in einem Ext4-Dateisystem mittels File-Carving-Methoden auffindet und diese mit einer Metadatenanalyse rekonstruiert. Zusätzlich wird ein forensisches Werkzeug vorgestellt, welches allein auf Basis Ext4-spezifischer Parameter Dateien rekonstruieren kann. Bei diesem Ansatz wird das Auslesen des Superblocks und der Gruppen-Deskriptor-Tabelle gänzlich vermieden, wodurch selbst von korrumpierten oder überschriebenen Ext4-Dateisystemen Dateien rekonstruiert werden können.

Abstract

In digital forensics, especially in the field of storage volume forensics, the reconstruction of files plays a major role. More specifically, deleted or lost files can contain traces that are essential to forensic investigations. For the reconstruction of files from storage volumes, specifics like the underlying file or operating system have to be considered. An example for a file system common to Linux operating systems is the Ext4 file system. For the interpretation of the Ext4 file system, the integrity of the so-called superblock is essential, which is why its availability is secured by redundant copies spread over the hard drive. Apart from the superblock, the group descriptor table plays an important role since it contains the coarse-granular layout of the file system, and therefore is imperative to the reconstruction of content data. This thesis presents an approach that finds files in an Ext4 file system by means of various search patterns for utilizing methods of file carving and recovers them by meta data analysis. Additionally, a forensic tool which is able to reconstruct files based solely on Ext4-specific parameters is presented. This approach omits reading the superblock and group descriptor table entirely, which is why it manages to recover files even from corrupted or overwritten Ext4 file systems.

Inhaltsverzeichnis

1

EINLEITUNG

Im Bereich der Datenträgerforensik spielt die Rekonstruktion von Dateien eine große Rolle [8]. Da sich die digitale Forensik auf Spuren stützt, die auf digitalen Medien gesichert wurden, ist auch die Rekonstruktion von gelöschten oder verloren gegangenen Dateien von Interesse. Spezifische Gegebenheiten, die für die Rekonstruktion von Daten von großer Bedeutung sind, müssen bei einer forensischen Untersuchung beachtet werden. Diese Gegebenheiten hängen vom Datei- oder Betriebssystem auf dem zu untersuchenden Datenträger ab, denn dadurch werden die Daten unterschiedlich auf dem Datenträger verteilt und organisiert [4].

Ein mögliches Dateisystem stellt das Ext4-Dateisystem dar. Die Ext-Dateisystemfamilie ist in verschiedenen Bereichen als Standarddateisystem anzutreffen, so wie unter UNIX-ähnlichen Betriebssystemen und dem Android-Betriebssystem auf mobilen Geräten [12]. Das Grundprinzip der Ext-Dateisystem-Familie besteht in einem schnellen und zuverlässigen Dateisystem, das Ausfallsicherheit bietet. Es wurden bei der Entwicklung des Ext4-Dateisystems, ausgehend von den Vorgängerversionen Ext2/3, Funktionalitäten entworfen und hinzugefügt, die einige der Problematiken der Vorgängerversionen beheben. Diese verändern allerdings den Aufbau des Dateisystems und erfordern eine Anpassung der Methodiken bei der Rekonstruktion von Dateien.

Für die Interpretation von Ext-Dateisystemen, dazu zählen auch die Vorgängerversionen von Ext4, ist die Integrität des sogenannten Superblocks essentiell, um die Struktur des Dateisystems auslesen zu können. Darin befinden sich wichtige Informationen über die internen Parameter, auf denen das Dateisystem aufbaut. Seine Verfügbarkeit wird durch verteilte, redundante Kopien auf der Festplatte sichergestellt [22]. Falls der Superblock jedoch unlesbar ist, da er zum Beispiel durch Überformatierung überschrieben wurde und somit auch die redundanten Kopien schwer zugänglich sind, so ist es schwierig, die nötigen Ext4-Parameter zu ermitteln, um die verbliebenen Daten zu rekonstruieren, da die Organisation der Dateien auf der Festplatte durch diese Parameter festgelegt wird [4]. Gleiches gilt für die Gruppen-Deskriptor-Tabelle, die ebenfalls durch im Dateisystem verteilte, redundante Kopien abgesichert ist.

Weiterhin bietet das Ext4-Dateisystem die Möglichkeit, Ausfallsicherheit durch ein Journal zu gewährleisten. In diesem werden Änderungen an Dateien und ihren Metadaten protokolliert, wodurch diese im Falle eines Systemabsturzes während einer Festplattenoperation rekonstruiert werden können. Da Ext4 auch als Dateisystem unter dem Android-Betriebssystem Verwendung findet, spielt das Dateisystem auch für die Forensik auf mobilen Geräten eine große Rolle. Dort ergibt sich u.A. die sogenannte „Journaling of Journal“-Anomalie, bei dem das auf Android-Systemen häufig verwendete SQLite-Datenbankverwaltungssystem Änderungen an seiner Datenbank in der eigenen Journaldatei dokumentiert, woraufhin das Ext4-Journal eben diese Änderung zusätzlich dokumentiert [19].

Mit dem Sleuthkit [7] von Brian Carrier wird ein Open-Source-Werkzeug zur Verfügung gestellt, mit dem forensische Untersuchungen durchgeführt werden können. Im Gegensatz zu kommerziellen Lösungen bieten Open-Source-Lösungen die inhärente Nachvollziehbarkeit der Ergebnisse. Das Sleuthkit bietet sowohl Kommandozeilen-Programme, als auch graphische Oberflächen für die Analyse. Zusätzlich bietet das Sleuthkit ein Framework; für dieses können Module entwickelt und eingehängt werden, so dass sich die Möglichkeit bietet, seine Funktionalität nach Belieben zu erweitern.

In dieser Arbeit wird ein Ansatz vorgestellt, der ohne Informationen aus dem Superblock oder der Gruppen-Deskriptor-Tabelle des Ext4-Dateisystems Dateien rekonstruieren kann. Dabei wird mittels Suchmustern nach Inodes – der Repräsentation von Dateien in Ext4 – gesucht, und diese mittels Metadatenanalyse rekonstruiert. In diesem Ansatz wird die File-Carving-Methode mit einer speziellen Metadatenanalyse-Methode kombiniert, um somit die Vorteile beider Vorgehensweisen nutzen zu können. Für die Rekonstruktion der Dateien werden zwei verschiedene Methodiken geboten, die sich in den resultierenden Ergebnissen unterscheiden.

Desweiteren wird ein forensisches Werkzeug vorgestellt, mit dem dieser Ansatz umgesetzt wurde. Dieses Werkzeug wurde als Modul für das Sleuthkit-Framework entwickelt und ist von Nutzerseite konfigurierbar. In dieser Arbeit wird detailliert die Funktionsweise dieses Moduls erläutert. Bei der Umsetzung wurde stets auf die Nachvollziehbarkeit und die Korrektheit der Ergebnisse geachtet, da es als forensisches Werkzeug dienen soll.

In diesem Kapitel wird zuerst auf die Motivation der Arbeit eingegangen. Dabei wird erläutert, aus welchen Anwendungsszenarien heraus Bedarf an den Beiträgen, die diese Arbeit präsentiert werden, besteht. Hierfür werden grundlegende Rekonstruktionsmethoden mit ihren Einschränkungen aufgezeigt und es wird dargestellt, wie die vorgestellte Arbeit diese abschwächt. Anschließend werden die Beiträge dieser Arbeit erläutert. In diesem Abschnitt werden die Ziele, die durch den vorgestellten Ansatz erreicht wurden, dargelegt. Darauf folgt ein Abschnitt über den aktuellen Stand der Technik, in dem verwandte Arbeiten präsentiert und in einen Kontext zur vorgestellten Arbeit gesetzt werden. Schlussendlich wird ein Überblick über den verbleibenden Teil der Arbeit gegeben.

1.1 Motivation

Das Ext4-Dateisystem ist ein weit verbreitetes Dateisystem, welches heutzutage nicht nur als Standard für Linux-Distributionen verwendet wird, sondern auch auf mobilen Geräten Anwendung findet [12]. Somit ist ein Ansatz zur Datenrettung auf diesem Dateisystem ein für die digitale Forensik relevantes Ziel. Ext4 und seine Vorgänger speichern die Metadaten, die ihre Struktur vorgeben, im sogenannten Superblock und der Gruppen-Deskriptor-Tabelle ab. Ohne diese Metadatenstrukturen ist es nur schwer möglich, das Dateisystem korrekt zu interpretieren und die Daten zu rekonstruieren. Im Falle einer Überformatierung sind somit alle Daten verloren, selbst wenn die eigentlichen Speicherbereiche, in denen sich Inhaltsdaten befinden, nicht überschrieben worden sind. Der in dieser Arbeit beschriebene Ansatz hat das Ziel, ohne die Hilfe dieser Metadatenstrukturen Dateien zu rekonstruieren.

Die neueste Generation der Ext-Dateisystem-Familie stellt Rekonstruktionsansätze vor neue Probleme, wie es auch von Fairbanks [11] beschrieben wird. Für die Vorgängerversionen stehen Ansätze und forensische Werkzeuge zur Rekonstruktion von Dateien zur Verfügung, die sich die vorhandenen Inhalte der Metdadatenstrukturen zu Nutze machen [29]. Beim Ext3-Dateisystem werden beispielsweise in Inodes – die Datenstrukturen in denen die Metadaten zu den einzelnen Dateien zu finden sind – indirekte Blockzeiger auf Inhaltsdatenblöcke gehalten. Durch die Dereferenzierung dieser Blockzeiger ist es möglich, die Dateiinhalte zu rekonstruieren. Beim Ext4-Dateisystem werden die Verweise auf die Dateninhalte anders behandelt, weswegen dafür eine Erweiterung nötig ist.

Die Struktur mit den indirekten Blockzeigern wurde durch sogenannte Extents ersetzt. Mit diesen lässt sich die Dateifragmentierung verringern und auch die Performance von Dateizugriffen verbessern. Allerdings sind dadurch Ansätze, die auf indirekten Blockzeigern anstelle von Extents aufbauen, nicht auf einem Ext4-Dateisystem anwendbar. Aus diesem Grund müssen neue Ansätze, wie der in der vorgestellten Arbeit dokumentierte, entwickelt werden.

Durch den verbreiteten Einsatz des Ext4-Dateisystems auf Mobilgeräten, sind Rekonstruktionsansätze für dieses Dateisystem für die digitale Forensik von besonderer Bedeutung. Die Ausfallsicherheit, die durch das Ext4-Journal gegeben ist, geht allerdings mit einem weiteren Problem bei der Rekonstruktion von Dateien einher. Aufgrund der oben genannten „Journaling of Journal"-Anomalie, bietet es sich an, das Ext4-Journal zu deaktivieren. Quellen wie Kim u.a. [20] geben an, dass beispielsweise eine `INSERT`-Anweisung auf einem SQLite-Datenbanksystem mit einer Nutzlast von 100 Byte zu Schreibvorgängen von über 36KiB führen können.

Durch das Deaktivieren des Ext4-Journals wird das beschriebene Problem zwar gelöst, allerdings werden dann Änderungen an Dateien nicht protokolliert. Dies führt bei einem Systemabsturz dazu, dass die Rekonstruktion von Dateien des Dateisystems erschwert wird. Aus diesem Grund sind Ansätze, die Dateirekonstruktion basierend auf dem Ext4-Journal unterstützen, in diesem Fall nicht anwendbar. Der in dieser Arbeit vorgestellte Ansatz kann das Journal zur Rekonstruktion von Dateien nutzen, jedoch ist dies nicht sein Hauptziel. Vielmehr werden beim vorgestellten Ansatz keinerlei Metadaten für die Verwaltung des Journals ausgelesen und ausgewertet. Da das korrekte Auslesen des Journals ohne Superblock bzw. Gruppen-Deskriptor-Tabelle nicht gewährleistet ist, kann das Journal nicht aktiv zur Rekonstruktion von Dateien genutzt werden. Dennoch gelingt es dem in dieser Arbeit vorgestellten Ansatz, Dateien durch das Journal zu rekonstruieren, ohne das Journal zu interpretieren.

Bereits existierende Ansätze zur Dateirekonstruktion mittels Metadatenanalyse bieten den Nachteil, dass die notwendigen Metadatenstrukturen intakt sein müssen. Im Falle eines Ext4-Dateisystems gibt es verschiedene Mechanismen zur Ausfallsicherheit und damit zur Erhaltung der wichtigen Metadatenstrukturen, so wie das Journal und Kopien des Superblocks und der Gruppen-Deskriptor-Tabelle. Der Anspruch an die Intaktheit der gelesenen Metadaten bleibt allerdings stets vorhanden.

Zusätzlich stellt die Auffindung beziehungsweise Rekonstruktion der Metadatenstrukturen eine Herausforderung dar. Beispielsweise kann sich in jeder Blockgruppe eine Kopie des Superblocks befinden. Jedoch muss nicht bekannt sein, wie groß die Blockgruppen sind; somit ist die physikalische Adresse des ersten intakten Superblocks nicht zwingend bekannt. Durch den vorgestellten Ansatz soll auf einen Großteil der Metadatenstrukturen verzichtet werden, damit möglichst wenig Abhängigkeit zu den Gegebenheiten auf dem Datenträger existiert.

Die Herausforderungen, die durch die Rekonstruktion von Dateien mittels Metadatenanalyse aufkommen, werden durch den File-Carving-Ansatz umgangen. Beim reinen File-Carving werden keine Anforderungen an das Datei- bzw. Betriebssystem gestellt, allerdings an den Dateiinhalt. Im direkten Vergleich zur Metadatenanalyse kann man hier von einer Verschiebung der Abhängigkeiten von System- zur Datei-Ebene sprechen. Da Dateiinhalte verschiedener Dateitypen unterschiedlich organisiert werden, muss auf jeden Dateityp gesondert eingegangen werden.

Ein Kernproblem des File-Carving-Ansatzes ist die Dateifragmentierung. Da die Inhaltsdaten nicht sequentiell abgespeichert werden müssen, muss mittels spezieller Vorgehensweisen entschieden werden, welche Datenblöcke zu einer Datei gehören, und welche aufgrund von Fragmentierung zwischen den gesuchten Blöcken positioniert sind. Für die genannten Probleme gibt es bereits verschiedene Ansätze, jedoch sind diese stark vom gesuchten Dateiinhalt abhängig und behandeln daher oft nur ein ausgewähltes Dateiformat.

Der vorgestellte Ansatz verbindet File-Carving und Metadatenanalyse. Dadurch entsteht eine Abhängigkeit zum Dateisystem, in diesem Fall zum Ext4-Dateisystem, allerdings ist der vorgestellte Ansatz unabhängig vom Dateiinhalt. Dabei wird File-Carving nach Metadaten genutzt, im speziellen Ext4-spezifische Inodes, um diese zu analysieren. File-Carving-Ansätze müssen das Fragmentierungsproblem dateispezifisch lösen, jedoch sind Inodes niemals fragmentiert und ihre Struktur ist weitgehend im Voraus bekannt. Diese Struktur bildet die Grundlage für eine konfigurierbare Mustersuche, wie sie für Carving-Ansätze typisch ist. Die so gefundenen Inodes können ohne Abhängigkeit vom Superblock oder der Gruppen-Deskriptor-Tabelle analysiert werden.

Da es in der digitalen Forensik eine Rolle spielt, essentielle Spuren auf Datenträgern zu finden, es jedoch durch die Vielfalt von Ansätzen zu komplex ist, besagte Datenträger manuell zu untersuchen, müssen entsprechende Werkzeuge für ihre Sicherung bereit gestellt werden. Das Sleuthkit, als Beispiel für ein frei verfügbares Open-Source Toolkit, unterstützt viele der gängigen Dateisysteme, so wie die Ext-Dateisystem-Familie und NTFS [5]. Es bietet verschiedene Werkzeuge, die für die forensische Analyse genutzt und in Kombination für die bereits erwähnten Ansätze verwendet werden können.

Im Gegensatz zu Open-Source-Werkzeugen bieten auch kommerzielle Alternativen Funktionalitäten, die für Datenträgeranalysen benötigt werden, jedoch sind diese oftmals nicht quelloffen, womit ein Vertrauensvorschuss der Dokumentation des Werkzeugs entgegengebracht werden muss. Das offensichtliche Nachvollziehen der Ergebnisse ist in diesem Fall nicht möglich. Im forensischen Bereich ist es allerdings von äußerster Wichtigkeit, zu wissen, wie die verwendeten Werkzeuge die Ergebnisse generieren, damit sie juristische Beweiskraft haben. Dadurch, dass quelloffene Werkzeuge per Definition vollständige Nachvollziehbarkeit der Ergebnisse aufweisen, spielen diese eine signifikante Rolle im forensischen Bereich.

Die Erweiterung des Sleuthkit-Toolkits um frei verfügbare bzw. selbst geschriebene Module bietet Ermittlern die Möglichkeit, die Funktionalität zu erzeugen, die für forensische Untersuchungen notwendig ist. Zusätzlich ermöglicht das Bereitstellen selbst geschriebener Module es, den Funktionsumfang des Sleuthkits zu erweitern und die Analyse eines Datenträgers mit einer Reihe verschiedenster Rekonstruktionsalgorithmen durchzuführen. Aus diesem Grund wurde sich dazu entschieden, den in dieser Arbeit vorgestellten Ansatz in das bereitgestellte Framework des Sleuthkits einzubinden.

1.2 Beiträge dieser Arbeit

Diese Arbeit stellt einen Ansatz vor, der zur Auffindung von Dateien auf einem Ext4-Dateisystem und zu deren Rekonstruktion entwickelt wurde. Dabei werden die zentralen Metadatenstrukturen des Dateisystems, wie etwa der Superblock und die Gruppen-Deskriptor-Tabelle, nicht ausgelesen. Dadurch, dass diese Abhängigkeit umgangen wird, kann das gegebene Dateisystem erkannt und ohne zusätzliche Informationen rekonstruiert werden.

Ein denkbarer Anwendungsfall für den vorgestellten Ansatz trifft zu, wenn Teile der Festplatte überschrieben oder überformatiert wurden und somit viele Metadaten verloren sind. Dieser Ansatz setzt direkt an der Stelle im physikalischen Speicherbereich an, an dem die entsprechenden Metadaten für die Dateien stehen: den Inodes. Somit erlaubt er es, auch Dateien zu rekonstruieren, die trotz Überschreibung von Speicherbereichen auf der Festplatte vorhanden waren. Damit findet der Ansatz Anwendung, wenn eine klassische Metadatenanalyse keine Informationen rekonstruieren kann. Um Dateien auf dem System ausfindig zu machen, werden Suchmuster verwendet, wie es auch im File-Carving-Ansatz der Fall ist. Dadurch ist der vorgestellte Ansatz als eine Kombination aus File-Carving und Metadatenanalyse zu verstehen.

Mittels Mustervergleichen, die sich auf Inodes eines Ext4-Dateisystems beziehen, werden potentielle Dateien erkannt und ihre Inhaltsdaten können anschließend durch Interpretieren des gefundenen Inodes wiederhergestellt werden. Mögliche Suchmuster stellen beispielsweise Zeitstempel oder Zugriffsrechte dar; fast alle Attribute eines Inodes können hierfür verwendet werden. Dabei ist allerdings zu beachten, dass durch einige Suchmuster eine semantische Selektion getroffen wird, die die Ergebnisse beeinflussen kann. Die Auswirkung der Suchmuster und damit die getroffene Selektion wurde im Laufe der Arbeit untersucht.

Die Intaktheit eines Inodes muss nicht sichergestellt sein: lediglich die Verweise auf die Datenblöcke – die Extent-Strukturen – müssen interpretierbar sein und der Inode muss durch die Suchmustervergleiche als gültig akzeptiert werden. Dabei gibt es keine Einschränkung durch das Dateiformat; einzig der Dateityp, den der Inode repräsentiert, hat einen Einfluss auf die spätere Interpretation der Inhaltsdaten. Für die Rekonstruktion der Dateien wurde allerdings eine Einschränkung bezüglich des Inode-Dateityps getroffen: Nur reguläre Dateien und Verzeichnisse werden in der Arbeit betrachtet. Da diese beiden Dateitypen mittels der Extent-Struktur auf ihre Inhaltsdaten verweisen, geschieht das Auslesen der Verzeichnisse und der regulären Dateien auf die gleiche Weise; dabei besteht keine Abhängigkeit vom Dateiinhalt. Die Korrektheit bezüglich der Vollständigkeit und Intaktheit der rekonstruierten Dateien wurde ebenfalls evaluiert.

Der vorgestellte Ansatz wurde als Modul für das Sleuthkit-Framework implementiert. Dadurch ist es möglich, diesen Ansatz neben anderen Untersuchungen in einem Sleuthkit-Programmaufruf auf ein Dateisystem anzuwenden. Dabei bietet die Implementierung zwei verschiedene Modi, den Inhaltsdaten- und den Metadaten-Modus, durch die verschiedene Ansätze zur Rekonstruktion von Dateien bereitgestellt wurden. Der Inhaltsdaten-Modus stellt hierbei eine Rekonstruktion bereit, die sich allein auf die Inhaltsdaten regulärer Dateien bezieht. Lediglich ein Mindestmaß an Informationen über das Dateisystem ist notwendig, um in diesem Modus Dateien zu rekonstruieren. Mit dem Metadaten-Modus können zusätzlich Dateinamen und die Verzeichnishierarchie des Dateisystems rekonstruiert werden. Da die Verzeichniseinträge interpretiert und die Nummern der Inodes berechnet werden müssen – schließlich sollen keine Metadaten des Dateisystems verwendet werden – kann die Rekonstruktion der Dateien nur mit erweiterten Kenntnissen über die strukturellen Parameter des Dateisystems durchgeführt werden. Allerdings bietet der Ansatz die Möglichkeit, von Standardwerten auszugehen, so dass der Nutzer keine zusätzlichen Kenntnisse braucht, wenn das Dateisystem standardmäßig formatiert wurde.

Das Modul bietet zudem die Möglichkeit, ohne jegliche vom Nutzer angegebenen Informationen zu arbeiten, allerdings müssen dafür einige Voraussetzungen erfüllt werden. Dafür muss die zu untersuchende Partition als Datenträgerabbild existieren, so dass die Dateisystemgröße anhand der Größe des Datenträgerabbilds bestimmt werden kann. Wurden bei der Erstellung des Dateisystems keine besonderen Parameter angegeben, hängen diese allein von dieser Dateisystemgröße ab. Ist dies der Fall, können alle weiteren Parameter des Ext4-Dateisystems durch Standardparameter geschätzt werden.

Mit diesem Ansatz soll gezeigt werden, dass Daten von Ext4-Dateisystemen selbst ohne strukturelle Metadaten wiederhergestellt werden können. Dabei kann nicht nur der Dateiinhalt wiederhergestellt werden, sondern auch der ursprüngliche Name der Datei, bzw. der Pfad, an dem die Datei auf dem Datenträger zu finden ist. Dem Nutzer wird durch die Implementierung als ein Modul für das Sleuthkit-Framework die Möglichkeit geboten, einige Optionen selbst zu konfigurieren. Dadurch kann das entwickelte forensische Werkzeug in einem breiten Spektrum von Testfällen eingesetzt werden.

1.3 Stand der Technik

Brian Carrier [4] beschreibt in seinem Buch verschiedene Partitions- und Dateisysteme. Aufbauend auf der Organisation einer Festplatte in Partitionen wird der Begriff des Dateisystems erläutert. Hierbei wird Bezug genommen auf Dateisysteme wie Ext2/3, NTFS und FAT, welche in eine eigene Klassifikation eingeordnet werden. Für diese Dateisysteme werden verschiedene Vorgehensweisen bei der forensischen Untersuchung diskutiert; ebenso wird die juristisch korrekte Erhebung und Dokumentation gefundener Spuren beschrieben. Hierbei liegt der Kontext im Feld der digitalen Forensik.

Carrier stellt außerdem verschiedene Werkzeuge, die die genannten Vorgehensweisen unterstützen, vor. Darunter befindet sich das von ihm entwickelte Sleuthkit [5], das u.a. verschiedene Kommandozeilen-Tools für die digitale Forensik bietet. Die vorgestellte Arbeit beschäftigt sich mit dem Aufbau des Ext4-Dateisystems, weshalb dessen Vorgänger Ext3 ein wichtiger Bestandteil der zu Grunde liegenden Theorie ist. Darüber hinaus wurde das in dieser Arbeit vorgestellte forensische Werkzeug als Modul in das Sleuthkit-Framework integriert, wodurch es Teil eines Open-Source-Toolkits ist.

Open-Source-Werkzeuge stehen im Bereich der digitalen Forensik, wie auch in jedem anderen Feld, stets in Konkurrenz mit kommerziellen Produkten. Das Ziel forensischer Werkzeuge ist die Beschaffung von Spuren zur gerichtlichen Nutzung, weshalb die Nachvollziehbarkeit ihres Zustandekommens von besonderer Wichtigkeit ist. Durch den frei zur Verfügung gestellten Programmtext von Open-Source-Werkzeugen ist die Nachvollziehbarkeit der Ergebnisse stets gegeben, worüber bei kommerziellen Lösungen keine Aussagen getroffen werden können. Von Manson u.a. [23] wird das Sleuthkit mit zwei kommerziellen Alternativen auf ihre Nutzbarkeit, Robustheit und die Verlässlichkeit und Verifizierbarkeit der Ergebnisse verglichen. Sie ziehen den Schluss, dass kommerzielle Lösungen besseren Nutzerkomfort gegenüber dem Sleuthkit bieten, jedoch auch, dass die Ergebnisse des Sleuthkits in keiner Weise hinter denen kommerzieller Alternativen stehen und erwähnen, dass Open-Source-Werkzeuge wie das Sleuthkit die am weitesten verbreiteten Werkzeuge im Bereich der digitalen Forensik sind.

Auch Hofherr [16] befasst sich in seiner Arbeit mit forensischen Open-Source Werkzeugen, exemplarisch mit dem Sleuthkit, für Post-Mortem-Analysen von Datenträgern. Hierbei untersucht er das Sleuthkit auf seinen Funktionsumfang und seine Nutzbarkeit im forensischen Kontext. Er betont hierbei die Ergebnisqualität, die Nachvollziehbarkeit und die Erweiterbarkeit des Sleuthkits. Jedoch wird von ihm erwähnt, dass sich die Autopsy-Oberfläche des Sleuthkits besser für Standardvorgehensweisen als für komplexere Analysen eignet. Zu diesen wird die manuelle Verkettung der vom Sleuthkit zur Verfügung gestellten Werkzeuge empfohlen.

Die genannten Arbeiten betonen die Wichtigkeit von Open-Source Werkzeugen im forensischen Bereich, insbesondere dem Sleuthkit. Dies dient als Motivation für die vorgestellte Arbeit, das forensische Werkzeug als Modul umzusetzen, das in das Sleuthkit eingebunden werden kann.

Die für Open-Source-Werkzeuge typische Nachvollziehbarkeit der Ergebnisse ist unabdingbar für das Feld der digitalen Forensik, da rechtliche Rahmenbedingungen und Ansprüche an ihre Untersuchungsergebnisse gestellt werden. Die Arbeit von Casey [8] erklärt eingehend das Vorgehen bei digitalen forensischen Analysen. Dabei diskutiert er sowohl technische Details, die bei der Datenakquise eine Rolle spielen, als auch Vorgehensmodelle für digitale Untersuchungen. Besagte technische Details beziehen sich auf die Speicherung von Daten in Computersystemen; so werden sowohl für gängige Datei- und Betriebssysteme potentielle Fundorte von Spuren genannt, als auch Low-Level-Details wie binäre Darstellungen von Daten und deren Interpretation erklärt. Von ihm aufgestellte Grundsätze, wie das Vermeiden der Erzeugung eigener Spuren während der forensischen Untersuchung, werden in der vorgestellten Arbeit daher strikt befolgt. Auch die Notwendigkeit der Kenntnis des betrachteten Systems, wie sie von Casey in seiner Arbeit gefordert wird, ist in der vorgestellten Arbeit durch die Einschränkung auf Ext4-Dateisysteme gegeben, um dateisystemspezifische Rekonstruktionsmethoden anwenden zu können.

Auch im Buch von Nelson u.a. [26] werden verschiedene Ansätze zur Datenwiederherstellung in der digitalen Forensik vorgestellt, ebenso wie schon existierende kommerzielle und frei zugängliche forensische Werkzeuge. Im Gegensatz zum Buch von Casey [8] wird in dieser Arbeit spezieller auf einzelne Teilgebiete der digitalen Forensik eingegangen, so wie etwa der Mobilgeräte- und der Cloud-Forensik.

Craiger [9] beschreibt in seiner Arbeit Methoden der digitalen Forensik, mit denen Daten von Linux-System wiederhergestellt werden können. Insbesondere das Wiederherstellen gelöschter und versteckter Dateien, sowie Daten aus flüchtigem Speicher und Dateien mit geänderter Dateiendung wird hervorgehoben. Es wird auch darauf hingewiesen, dass viele forensische Methoden automatisiert wurden, so wie die Berechnungen von Prüfsummen, aber auch viele noch eine manuelle Eingabe erfordern, wodurch die Analyse von großen Festplatten mit vielen Daten erschwert wird. Insbesondere die automatische Identifikation, Wiederherstellung und Untersuchung digitaler Spuren sind laut Craiger für große Dateisysteme notwendig. Die Modulpipeline des Sleuthkit-Frameworks bietet die Möglichkeit, diese Automatisierung vorzunehmen. Das vorgestellte forensische Werkzeug, das als Modul für das Sleuthkit-Framework implementiert wurde, bietet eine automatisierte Identifikation und Wiederherstellung von Dateien von Ext4-Dateisystemen.

Da bei einer forensischen Untersuchung das Zielsystem entscheidend ist, muss hierbei der Aufbau und die Funktionsweise des untersuchten Dateisystems, wie im Falle der vorgestellten Arbeit das Ext4-Dateisystem, bekannt sein. Fairbanks u.a. [12] vergleichen in ihrer Arbeit das Ext4-Dateisystem mit seinen Vorgängern. Dabei wird darauf eingegangen, wie die Unterschiede zwischen Ext4 und den Vorgängern die Dateisystem-Forensik beeinflussen. Das Ext4-Dateisystem erfreut sich laut ihnen großer Beliebtheit, stellt die digitale Forensik jedoch auch vor neue Herausforderungen, die auch in dieser Arbeit behandelt werden. Diese entstehen durch die Änderung einiger wichtiger Metadatenstrukturen, wie beispielsweise in der Nutzdatenreferenzierung, die für das vorgestellte forensische Werkzeug von besonderer Wichtigkeit ist. Jedoch bewirken diese Änderungen nicht nur neue Herausforderungen, sondern liefern auch neue Möglichkeiten für die forensische Untersuchung. Beispielsweise stellt der neu hinzugekommene Zeitstempel der Erstellung einer Datei ein weiteres, rekonstruierbares Dateimerkmal dar.

In einer anderen Arbeit von Fairbanks [11] beschreibt er das Ext4-Dateisystem im Detail und stellt Erweiterungen gegenüber Ext3 vor. Diese werden eingehend auf Dateisystemebene beschrieben. Dabei dokumentiert die Arbeit vor allem Low-Level-Merkmale wie Extents, HTrees und Flex-Gruppen. Es wurden ebenso Stellen in den Metadaten identifiziert, in denen sich ungenutzter Speicher befindet, der zum Verstecken von Daten verwendet werden kann. Das Hinzukommen von Prüfsummen in wichtigen Metadatenstrukturen des

Ext4-Dateisystems bedeutet, dass Veränderungen dieser Daten sich in besagten Prüfsummen wiederspiegeln könnten. Fairbanks fordert weitere Forschung auf eben diesem Gebiet.

Das Beachten der Ext4-spezifischen Merkmale ist für den vorgestellten Ansatz wichtig, da diese den Aufbau des Dateisystems und seiner Metadaten beeinflussen. Da diese Metadaten zur Rekonstruktion von Dateien benötigt werden, muss ihr genauer Aufbau bekannt sein.

Mathur u.a. [24] beschreiben in ihrer Arbeit die Gründe für die Ablösung von Ext3 als Standard-Dateisystem für Linux. Sie bezeichnen dafür u.A. die maximale Dateisystemgröße von 16 TiB, da diese vor allem im kommerziellen Umfeld eine spürbare Einschränkung darstellt. Nichtsdestotrotz wurde Ext3 als Grundlage für ein neues Dateisystem gewählt, da vor allem seine Stabilität und Zuverlässigkeit nicht aufgegeben werden sollte. Desweiteren werden die neuen Möglichkeiten von Ext4 aufgezeigt und seine Performance mit anderen Dateisystemen verglichen. Auch die Methoden, wie Ext3-Dateisysteme auf das neue Ext4-Dateisystem migriert werden, finden hier Erwähnung. Da sich von Ext3 auf Ext4 migrierte Dateisysteme von nativen Ext4-Dateisystemen in der Nutzdatenreferenzierung unterscheiden, muss zwischen diesen beiden Fällen bei der Rekonstruktion von Dateien unterschieden werden. Das vorgestellte Werkzeug betrachtet ausschließlich native Ext4-Dateisysteme.

Unabhängig vom vorliegenden Datei- und Betriebssystem kann die Rekonstruktion von Dateien mit dem sogenannten File-Carving-Ansatz durchgeführt werden. Dabei wird der Datenträger nach Mustern bestimmter Dateitypen durchsucht, etwa mittels Headern und Footern. Garfinkel [14] beschreibt in seiner Arbeit seine Lösung für eines der Hauptprobleme des File-Carving-Ansatzes: die Dateifragmentierung. Diese bewirkt, dass das alleinige Finden eines Dateianfangs und -endes nicht zur Rekonstruktion der Datei ausreicht. Die von Garfinkel beschriebene Lösung besteht darin, Teilobjekte im Bezug auf ihr Dateiformat rapide zu validieren. Bei der Rekonstruktion von Dateien mittels File-Carving müssen potentielle Fragmentketten im Bezug darauf validiert werden, ob sie Teile gültiger Dateien darstellen. Die vorgeschlagenen Validierungsmechanismen betreffen u.A. Header und Footer, Container-Strukturen und komprimierte Dateien. Diese beschränken sich in der Arbeit auf einige ausgewählte Dateiformate, wie zum Beispiel JPEG- und ZIP-Dateien. Das vorgestellte forensische Werkzeug betreibt ebenfalls Objektvalidierung beim Carving potentieller Inodes in Kombination mit semantischen Einschränkungen, wie etwa dem Eingrenzen der Ergebnisse auf Zeitintervalle. Hierbei handelt es sich nicht um einen reinen File-Carving-Ansatz, da mit der Festlegung auf Ext4-Inodes keine Dateisystemunabhängigkeit gegeben ist.

In der Arbeit von Hand u.a. [15] wird ein Ansatz präsentiert, der ohne Metadaten Dateien rekonstruiert. Die Suche beschränkt sich hierbei auf ausführbare Binärdateien im ELF-Format, die zwar Header-, aber keine Footer-Informationen besitzen. Die wesentliche Lösung des Fragmentierungsproblems besteht in der Kontrollflussanalyse des durch teilweise Disassemblierung erreichten Maschinencodes. Durch diese Analyse werden potentielle Anknüpfungspunkte für Dateifragmente gefunden, welche heuristisch sicher verknüpft werden können. Das systemnahe ELF-Format wird von Unix-artigen Betriebssystemen verwendet, weshalb die eben zitierte Arbeit einen Teil der File-Carving-typischen Betriebssystemunabhängigkeit verliert. Durch diese Konzession wird jedoch erreicht, dass die wiederhergestellten ausführbaren Dateien eher universeller Natur sind, als beispielsweise Bild- oder Tondateien. Diese Verschiebung der Abhängigkeiten vom Dateiinhalt zu Systemmerkmalen tritt auch in der vorgestellen Arbeit auf: Die Einschränkung auf native Ext4-Dateisysteme bewirkt zwar eine Abhängigkeit von systemspezifischen Merkmalen, ermöglicht jedoch die Rekonstruktion universell nutzbarer Inodes. Somit ist kein Wissen über Inhaltsdateien nötig.

Wissen über den konkreten Inhalt einer Datei wird dann notwendig, wenn systemunabhängiges Carving, beispielsweise nach Bild- oder Tondateien, durchgeführt werden soll. Poisel u.a. [28] etwa, vergleichen bestehende und neue Ansätze zur Wiederherstellung von Multimediadateien. Dabei stellen sie einen eigenen File-Carver vor, der fragmentierte Multimediadateien mittels Greedy-Heuristiken wiederherstellt. Die Arbeit von Sajja [34] beschäftigt sich ebenfalls mit fragmentierten Multimediadateien, im Speziellem mit MP3-Dateien mit variabler Bitrate. Der eben zitierte Ansatz erhebt statistische Informationen über die Veränderungen der Bitrate über die Zeit, wodurch Fragmentzugehörigkeiten geschätzt werden können. Wie bereits erwähnt muss die vorgestellte Arbeit keine Aussagen über Dateiformate treffen, um alle Dateien des Dateisystems zu rekonstruieren.

Eine weitere Form der Verschiebung von Abhängigkeiten beim File-Carving wird in der Arbeit von Foster [13] präsentiert. In dieser Arbeit wird ein Ansatz vorgestellt, in dem Dateien in Blöcke aufgeteilt, von diesen Prüfsummen gebildet und diese mit einer Datenbank von Prüfsummen von Blöcken im Vorfeld bekannter Dateien abgeglichen werden. Ein Nachteil dieser Methode besteht darin, dass nur Datenblöcke gesucht werden können, die vollständig bekannt sind. Weiterhin übersieht dieser Ansatz bekannte Blöcke, wenn diese zum Blockanfang versetzt beginnen. Es besteht also eine Abhängigkeit dazu, dass die gesuchten Dateien bereits genau bekannt sein müssen. Für den Rekonstruktionsalgorithmus selbst hat dies allerdings keine Bewandnis: Er kann vollständig unabhängig vom Datei- oder Betriebssystem ausgeführt werden und wird nicht von Dateiinhalten in seiner Wirkungsweise beeinflusst. Die algorithmische Unabhängigkeit von den rekonstruierbaren Dateiinhalten teilt sich der Ansatz mit dem vorgestellten forensischen Werkzeug.

Einen Ansatz, dessen Fokus auf der Performance statt auf der Wiederherstellung bestimmter Dateiformate liegt, präsentieren Richard u.a. [33] unter dem Begriff „In-Place File Carving“. Hierbei werden wiederhergestellte Dateien nicht explizit aus dem untersuchten Datenträger herauskopiert, sondern eine Datenbank über die Metadaten der gefundenen Dateien geführt. Mittels dieser Informationen können nachträglich relevante Dateien gezielt rekonstruiert und *false positives* und irrelevante Dateien ignoriert werden. So werden sowohl Zeit- als auch Speicherplatzbedarf minimiert. Auf der Implementierungsebene teilt sich das vorgestellte forensische Werkzeug das Prinzip der verzögerten Auflösung potentieller Resultate mit diesem Ansatz. Zwar wird kein reines In-Place-Carving der Ergebnisdateien angeboten, jedoch werden die direkten Carving-Ergebnisse – potentielle Inodes – nie gänzlich abgespeichert. Lediglich Merkmale, die ein Verwerfen des Inodes ermöglichen, werden temporär geladen. Ausschließlich die physikalischen Adressen der Inodes werden langfristig abgespeichert und abgerufen, wenn Dateien rekonstruiert werden sollen. Mit den direkten Carving-Ergebnissen wird also In-Place-Carving betrieben.

Als Gegensatz des File-Carvings bezüglich der bereits genannten Abhängigkeitsverschiebungen können Ansätze verstanden werden, die Dateisystem-Metadaten einsetzen, um Dateien beliebiger Formate wiederherzustellen. Auch das vorgestellte Werkzeug bedient sich einer Metadaten-Analyse: Die durch Carving ermittelten Inodes werden zur Rekonstruktion von Dateien verwendet. Generell gilt: Je mehr Metadaten zur Verfügung stehen, desto gezielter können Dateien gesucht werden und desto weniger muss über die gesuchten Dateien gewusst werden.

Beispielsweise stellen Lee u.a. [21] Methoden vor, durch die bei Ext2- und Ext3-Dateisystemen gelöschte Dateien mittels Metadatenstrukturen wiederhergestellt werden und vergleichen diese mit bereits existierenden Methoden. Eine andere Art von Metadaten, die sich terminologisch gesehen nicht in Meta-, sondern in Nutzdatenbereichen aufhält, stellt das Dateisystemjournal dar. Narváez[25] beschreibt eine Methode zur Rekonstruktion von Dateien aus einem Ext3-Dateisystem mit Hilfe der Metadaten seines Journals. Dabei wird ebenfalls in Betracht gezogen, dass sich das Ext3-Journal nicht auf dem gleichen Datenträger befinden muss, wie das dazugehörige Dateisystem.

Die Suche nach Dateisystem-Metadaten kann auf eine Weise erfolgen, bei der eine Reihe von Annahmen über die Positionen und Struktur der Metadaten getroffen wird. Sind etwa alle Parameter des Dateisystems bekannt, lassen sich alle Positionen der Metadaten aus ihnen erschließen. Stehen diese Informationen jedoch nicht zur Verfügung, können potentielle Metadaten mit Mustervergleichen gefunden werden.

Auf einer solchen Mustervergleichssuche basiert die Arbeit von Pomeranz [29]: Mit ihr wird ein Ansatz zur Datenwiederherstellung in Ext2- und Ext3-Dateisystemen vorgestellt, bei dem durch die Verwendung von indirekten Blockzeigern Nutzdaten wiederhergestellt werden. Dieser basiert auf der Suche nach einem Muster, welches den Bereich von Inodes kennzeichnet, in dem sich Blockzeiger befinden. Typischerweise sind die ersten 48 KiB des Inhalts einer Datei nicht signifikant fragmentiert, weshalb die ersten 12 Blockzeiger für gewöhnlich sequentiell durchnummeriert sind. Finden sich auf dem Datenträger also 12 durchgehend nummerierte, 4 Byte lange Adresswerte, liegt ein potentieller Inode vor.

Dieses spezielle Suchmuster lässt sich zwar auf Ext4-Dateisysteme nicht anwenden, da anstelle von indirekten Blockzeigern Extent-Strukturen zur Referenzierung von Nutzdaten verwendet werden, jedoch bietet der vorgestellte Ansatz eine Reihe anderer Suchmuster, die in Kombination eine Vorauswahl von Inodes aus dem Datenträger ermöglichen.

1.4 Überblick

In Kapitel 2 wird auf die technischen Grundlagen eingegangen, auf denen diese Arbeit beruht. Die Ext-Dateisystemfamilie, insbesondere das Ext4-Dateisystem, wird eingehend erläutert und auf technischer Ebene erklärt. Auch die Konzepte verschiedener Rekonstruktionsmethoden, wie etwa die des File-Carvings und der Metadatenanalyse, werden näher beschrieben. Ebenso wird das Sleuthkit und sein bereitgestelltes Framework vorgestellt.

Darauf aufbauend wird anschließend in Kapitel 3 auf die Implementierung des in dieser Arbeit vorgestellten forensischen Werkzeugs eingegangen. Die Funktionsweise und der Aufbau des Moduls für das Sleuthkit-Framework werden hier im Detail erklärt. Dabei wird sowohl die Umsetzung des vorgestellten Ansatzes in algorithmischer Hinsicht erläutert, als auch die technische Umsetzung, näher beleuchtet.

In Kapitel 4 werden die Ergebnisse, die durch den Ansatz erreicht wurden, aufgezeigt. Dabei wird ein Datensatz für die Evaluation vorgestellt, mit der anschließend die Korrektheit des forensischen Werkzeugs aufgezeigt wird. Die Selektionsrate der einzelnen Suchmuster wird anhand eines Testfalles evaluiert. Für die Überprüfung der Vollständigkeit werden sowohl Realfälle, als auch Spezialfälle, analysiert. Mit einigen Beispielen aus diesem Datensatz wird auch die Laufzeit analysiert.

Schließlich wird in Kapitel 5 auf die Arbeit im Gesamten eingegangen und zusammenfassend die geleisteten Beiträge hervorgehoben. Es werden sowohl Einschränkungen des vorgestellten Ansatzes, als auch mögliche Erweiterungen erwähnt.

2

TECHNISCHE GRUNDLAGEN

Heutzutage existieren viele verschiedene Dateisysteme, die zum Teil auch für bestimmte Anwendungsfälle ausgelegt sind. So, wie NTFS für Windows das Standard-Dateisystem ist, so gilt die Ext-Dateisystem-Familie als Standard für Linux-Betriebssysteme. Seitdem das Android-Betriebssystem für mobile Geräte Version 2.3 [19] erreicht hat, kann Ext4 auch auf diesen Geräten als Dateisystem verwendet werden. Das Grundprinzip der Ext-Dateisystem-Familie besteht in einem schnellen und zuverlässigen Dateisystem, das Ausfallsicherheit bietet. Wichtige Datenstrukturen werden deswegen durch redundante Kopien gespeichert. Zur Verringerung unnötiger I/O-Wartezeiten liegen alle Daten, die sich auf eine konkrete Datei beziehen, nah beieinander. Während der Entwicklung des Ext4-Dateisystems wurden Funktionalitäten entworfen und hinzugefügt, die einige der Problematiken der Vorgängerversionen beheben. Diese veränderten auch im Zuge dessen den Aufbau des Dateisystems und erforderten eine Anpassung der Methodiken bei der Rekonstruktion von Dateien.

Ext4-Dateisysteme werden typischerweise mithilfe eines Programms namens *mkfs* erstellt. Dieses Programm ist ein Standardbestandteil von UNIX und UNIX-ähnlichen Betriebssystemen. Unter UNIX kann eine Festplatte erst dann eingehängt werden, wenn sie vorher mit einem Dateisystem formatiert und von der Dateisystem-Hierarchie des Betriebssystems akzeptiert wurde. Beim Installieren des Betriebssystems muss die Partition, auf der sich das Betriebssystem später befindet, formatiert werden. Diese Funktion erfüllt mkfs. Dabei werden Standardwerte für den Aufbau des Dateisystems gesetzt, sofern der Nutzer nicht manuell eigene Werte angibt.

Zur Rekonstruktion verloren gegangener Dateien existieren verschiedene Ansätze, diese bedienen sich meist mindestens einer von zwei grundlegenden Methodiken. Hierzu gehört einerseits das vom verwendeten Dateisystem unabhängige *File-Carving* und andererseits das vom Dateiformat unabhängige Analysieren von Dateisystem-Metadaten; beide Ansätze werden in Abschnitt 2.2.1 näher erläutert. Die beiden genannten Ansätze stellen komplementäre Anforderungen und sind meist für verschiedene Ausgangslagen geeignet; somit begründet sich die Annahme, dass die Kombination beider Ansätze ein breites Nutzungsspektrum bieten kann.

Derartige Rekonstruktionsansätze sind nur mit großem Aufwand manuell durchführbar; es besteht die Notwendigkeit des Einsatzes von automatisierten Werkzeugen. Da mit solchen Werkzeugen in forensischen

Fällen Hinweise gefunden werden sollen, muss die exakte Funktionsweise der verwendeten Software bekannt sein, da sonst der Anspruch an die Nachvollziehbarkeit und Dokumentierbarkeit, wie er im forensischen Kontext gefordert wird, nicht erfüllt werden kann. Open-Source-Software bietet die maximal mögliche Nachvollziehbarkeit gefundener Ergebnisse durch die Verfügbarkeit des Quellcodes. Das *Sleuthkit* [7] von Brian Carrier ist ein bekanntes Beispiel für ein forensisches Toolkit mit offenem Quellcode. Dieses lässt sich mittels der graphischen Oberfläche *Autopsy* verwenden, die verschiedene Optionen zur Darstellung und Verwaltung von Datenträgerabbildern anbietet. Die Möglichkeit, sowohl das Sleuthkit als auch Autopsy mittels eigens geschriebener Module – oder Module von Drittanbietern – zu erweitern, spannt ein großes Spektrum von Funktionalitäten auf, die in verschiedenen Anwendungsfällen Verwendung finden.

In diesem Kapitel wird zuerst auf die Ext-Dateisystem-Familie eingegangen. Nach einer einführenden Entwicklungshistorie werden Ext2 und Ext3 im Detail erläutert, um darauf aufbauend die durch Ext4 hinzu gekommenen Neuerungen anschließend darzulegen. Der typische Aufbau samt der Unterteilungshierarchie und der Metadatenstrukturen wird speziell für das Ext4-Dateisystem eingehend aufgezeigt. Darauf aufbauend werden verschiedene Ansätze zur Dateirekonstruktion genannt. Dabei wird auf das File-Carving, das Daten ohne die Hilfe von Metadaten rekonstruiert, und die Dateirekonstruktion durch Interpretation von Dateisystem-Metadaten eingegangen. Abschließend wird die Architektur des Sleuthkits beleuchtet. Hierbei werden sowohl die vom Sleuthkit zur Verfügung gestellten Kommandozeilenwerkzeuge, als auch die Verwendungsmöglichkeiten des Sleuthkit-Frameworks und die verschiedenen Autopsy-Versionen vorgestellt.

2.1 Die Ext-Dateisystem-Familie

Das erste Ext-Dateisystem wurde 1992 von Rémy Card entwickelt [10]. Es war dabei das erste Dateisystem, das speziell für Linux entwickelt wurde, nachdem Einschränkungen des Minix-Dateisystem beobachtet wurden. Unter Anderem waren Partitionen auf 64 MiB beschränkt und Dateinamen konnten nicht länger als 14 Byte sein. Durch diese und andere Aspekte war das Minix-Dateisystem nicht für große Systeme geeignet.

Ext wurde durch das klassische UNIX-Dateisystem (UFS) inspiriert, wie sich beispielsweise darin zeigt, dass *Inodes* aus ihm übernommen wurden, wenn auch vereinfacht. In Inodes werden essentielle Metadaten von Dateien gespeichert, wie etwa Zeitstempel und Zugriffsrechte. Die UNIX-Dateisystemstrukturen, die nur zu Zwecken der Abwärtskompatibilität im UFS dienten, wurden bei der Entwicklung des Ext-Dateisystems entfernt. Zusätzlich wurden die oben genannten Einschränkungen durch das Ext-Dateisystem gelockert. Dateinamen durften nun bis zu 255 Byte lang sein und das Dateisystem durfte bis zu 2 GiB umfassen [18]. Jede Datei besaß lediglich einen Zeitstempel, der den letzten Zugriff jeglicher Art dokumentierte [37]. Erstellungs-, Modifikations- und Lesezeiten waren somit nicht zu unterscheiden.

Im Jahr 1993 wurde das zweite Extended-Dateisystem (Ext2) von Rémy Card entwickelt. Die maximale Größe des Dateisystems wurde auf 16 TiB[1] angehoben und es gab mehrere Zeitstempel für verschiedene Zugriffe auf eine Datei. Parallel dazu wurde ein weiteres Dateisystem, das xiafs-Dateisystem, entwickelt, doch dieses konnte sich nicht gegen Ext2 durchsetzen. Das Ext2-Dateisystem wurde zum am weitesten verbreiteten Standard-Dateisystem unter Linux. Später wurden noch einige zusätzliche Funktionen zum Dateisystem hinzugefügt, wie z.B. *Sparse Superblocks*.

Im Folgenden wird erläutert, wie sich von Ext2 aus die Ext-Dateisystem-Familie über ihre Generationen entwickelt hat. Dabei werden detailliert der Aufbau und die Funktionsweise der Dateisysteme Ext2 und Ext3 beschrieben. Im darauf folgenden Abschnitt wird, von Ext3 ausgehend, die Entwicklung zu Ext4 illustriert und seine tiefgreifendsten Änderungen betont.

---

[1] Die maximale Dateisystemgröße hängt von verschiedenen Faktoren ab. So lassen Linux-Versionen vor einschließlich Version 2.4 keine Partitionen zu, die größer sind als 2 TiB. Ein weiterer Faktor ist die Dateisystem-Blockgröße, welche durch die Hauptspeicher-Seitengröße maximiert wird. Typischerweise umfasst diese 4 KiB. Seitengrößen von 8 KiB werden ab Ext2 auch unterstützt. Solche Systeme haben dann eine maximale Dateisystemgröße von 32 TiB.

2.1.1 Ext2 und Ext3 im Detail

Ein Großteil der Informationen in diesem Kapitel stammt aus dem Buch „File System Forensic Analysis“ von Brian Carrier [4].

Ext3, wie auch seine Vorgänger, wurde mit dem Grundgedanken, ein zuverlässiges und schnelles Dateisystem zu bieten, im Jahr 2001 entwickelt. Hierbei handelt es sich um eine direkte Erweiterung des Ext2-Dateisystems um einige zusätzliche Funktionalitäten. Die wichtigste dieser Neuerungen stellt das *Journal* dar, welches ein Protokoll über die Dateiaktivitäten auf der Festplatte darstellt und somit die Wiederherstellung aus inkonsistenten Zuständen, wie etwa Abstürzen ermöglicht. Terminologisch gesehen erweitert Ext3 damit seinen Dateisystemtyp um die Bezeichnung „Journaling-Dateisystem“. Ext3 bietet außerdem die Möglichkeit, die Gruppen-Deskriptor-Tabelle mit vorab reserviertem Speicher auszustatten, um zukünftige Vergrößerungen des Dateisystems im laufenden Betrieb durchführen zu können. Des Weiteren unterstützt Ext3 mit dem sogenannten *HTree* eine Datenstruktur zur effizienten Suche von Dateinamen in großen Verzeichnissen. Dieser stellt prinzipiell einen nach Dateinamen-Hashes geordneten B-Baum dar. Da sich Ext2 und Ext3 nicht grundlegend unterscheiden, werden in Aussagen, die beide Dateisysteme betreffen, diese im Folgenden gesammelt als „Ext2/3“ bezeichnet.

Aufbau

Ext2/3 basiert auf sequentiell angeordneten Blöcken und Blockgruppen, wie es auch in Abbildung 2.1 zu sehen ist.

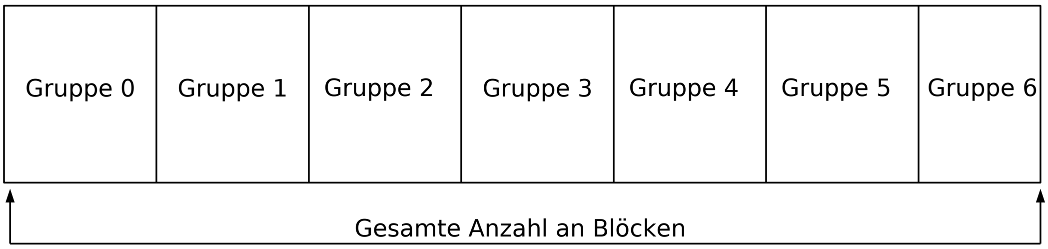


Abbildung 2.1: Layout eines Dateisystems mit insgesamt sieben Blockgruppen

Blöcke können 1024 Byte, 2048 Byte oder 4096 Byte groß sein. Diese werden, ihren Positionen auf dem Datenträger entsprechend, aufsteigend durchnummeriert. Hierbei trägt der erste Block die Nummer 0 und befindet sich im ersten Sektor des Dateisystems. Diese Nummer wird als *Blockadresse* bezeichnet. Da Blockgruppen durch Bitmaps repräsentiert werden, und diese einen Block groß sind, umfassen Blockgruppen standardmäßig so viele Blöcke, wie es Einträge in einer solchen Bitmap gibt. Für die oben genannten, gültigen Blockgrößen sind dies folglich 8192, 16384 bzw. 32768 Blöcke pro Blockgruppe. Blockgruppen werden genau wie Blöcke durchnummeriert. Spezielle Systeme mit einer Seitengröße von 8192 Byte erlauben auch diesen Wert als Blockgröße, woraus 65536 Blöcke pro Blockgruppe folgen. Jede Blockgruppe bis auf die letzte, ist gleich groß, da diese den Verschnitt beinhaltet, der entsteht, wenn die Dateisystemgröße nicht ganzzahlig durch die Blockgruppengröße teilbar ist. In Formel 2.1 wird gezeigt, wie anhand der Blockadresse die Blockgruppe errechnet werden kann, in der ein bestimmter Block zu finden ist.

$$G = \left\lfloor \frac{a-r}{n} \right\rfloor \quad \forall\, a \geq r \tag{2.1}$$

Hierbei steht G für die Nummer der Blockgruppe, a für die Blockadresse, für die der Wert errechnet werden soll, und n für die Anzahl der Blöcke pro Gruppe. Falls vor dem ersten Datenblock noch eine Anzahl reservierter Blöcke r vorhanden sein sollte, so gehören Blöcke dieses Speicherbereichs nicht zu einer Blockgruppe, sind aber durchaus nummeriert.

Jede Blockgruppe beinhaltet Metadaten, die ihre innere Struktur dokumentieren. Im allgemeinen Fall sind alle Blockgruppen im Dateisystem gleich aufgebaut, so wie es in Abbildung 2.2 gezeigt wird.

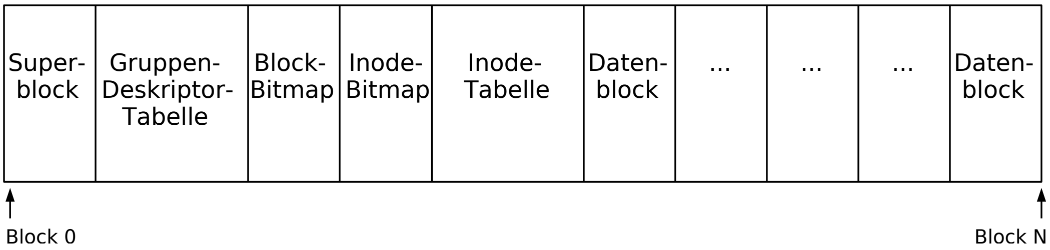


Abbildung 2.2: Layout einer Blockgruppe in einem Ext3-Dateisystem

In der ersten Blockgruppe im Dateisystem (Blockgruppe 0), befinden sich vor dem Superblock 1024 Bytes ohne Inhalt. Da der Superblock selbst 1024 Bytes lang ist, entsteht bei einer Blockgröße von 1024 Byte ein vollständig leerer Datenblock, der als reservierter Speicher gekennzeichnet wird. Der Superblock selbst beinhaltet viele essentielle Metadaten des Dateisystems. Dazu gehören die Anzahl und Größe der Datenblöcke, die Anzahl der Inodes, unterstützte Features, die Anzahl reservierter Blöcke vor dem Superblock und eine Vielzahl weiterer Attribute des Dateisystems. Auch nicht essentielle Daten stehen im Superblock, wie zum Beispiel der letzte Zeitpunkt und der letzte Pfad, an dem das Dateisystem eingehängt wurde, und der Name der Partition.

An den Superblock schließt die Gruppen-Deskriptor-Tabelle an. Diese und der Superblock sind die einzigen Datenstrukturen in der Blockgruppe, die einen festen Platz einnehmen. Alle anderen Metadatenbereiche besitzen zwar eine standardmäßige Aufstellung, wie sie in Abbildung 2.2 zu sehen ist, jedoch kann diese bei der Erstellung des Dateisystems umstrukturiert werden. Gruppen-Deskriptor-Tabellen beinhalten einen Eintrag, einen sogenannten Gruppen-Deskriptor, für jede Blockgruppe im Dateisystem. Der Aufbau eines solchen Gruppen-Deskriptors wird in in Tabelle 2.1 gezeigt.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Startende Blockadresse der Block-Bitmap
4	4	Startende Blockadresse der Inode-Bitmap
8	4	Startende Blockadresse der Inode-Tabelle
12	2	Anzahl der nicht allozierten Blöcke der Blockgruppe
14	2	Anzahl der nicht allozierten Inodes der Blockgruppe
16	2	Anzahl der Verzeichnisse der Blockgruppe
18	30	Unbenutzt

Tabelle 2.1: Aufbau eines Eintrags in der Gruppen-Deskriptor-Tabelle

Die Datenstruktur ist für jede Blockgruppe 32 Bytes lang. Es kann unter Ext3 auch der 64-Bit-Modus aktiviert werden, wodurch die Größe der Gruppen-Deskriptoren auf 64 Byte ansteigt. Ob der 64-Bit-Modus aktiviert ist, kann im Superblock ausgelesen werden.

Die darauffolgenden Bitmaps zeigen die Nutzung der Datenblöcke und Inodes innerhalb einer Blockgruppe an. Da pro Bitmap ein Block genutzt wird, limitiert sich so die Anzahl der Blöcke pro Blockgruppe auf die Bitanzahl in einem Block. Auch für die Anzahl der Inodes ist dies die obere Grenze, jedoch entspräche dies einem Inode pro Block. Da die Anzahl zugehöriger Blöcke je Inode im Realfall über eins liegt, werden einer Blockgruppe weniger Inodes zugewiesen, als durch die Inode-Bitmap möglich wäre. Folglich bleibt ein Teil von dieser unbenutzt.

Wie bereits erwähnt, wurde Ext3 im Laufe seiner Entwicklung um die Option des *Sparse Superblocks* erweitert. Durch diese Option wird die Anzahl der redundanten Kopien des Superblocks und der Gruppen-Deskriptor-Tabelle reduziert. So sind besagte Sicherungskopien nur in Blockgruppen zu finden, deren Nummer in der Menge S aus Formel 2.2 enthalten ist.

$$S = \left\{ x \in \mathbb{N}^0 \mid x = 0 \lor x = k^y \ , \ k \in \{3, 5, 7\} \land y \in \mathbb{N}^0 \right\} \tag{2.2}$$

Daraus folgt, dass Kopien des Superblocks und der Gruppen-Deskriptor-Tabelle in Blockgruppen wie 0, 1, 3, 5, 7, 9, 25, 27, 49, usw. zu finden sind. Dies garantiert einerseits eine hohe Dichte redundanter Kopien auf kleinen Systemen und andererseits einen überschaubaren Anstieg der Kopienzahl bei wachsender Dateisystemgröße. Durch diese Steigerung des Nutzdaten-zu-Metadaten-Verhältnisses erhöht sich die Speicherplatzeffizienz des Dateisystems.

Inodes

In Ext2/3-Dateisystemen werden essentielle Metadaten einer Datei in ihrem Inode gespeichert. Dazu gehören unter Anderem Zeitstempel, zugehörige Datenblöcke und Zugriffsrechte. Alle Inodes des Dateisystems haben die gleiche Länge, welche im Superblock auszulesen ist. Für jede Datei und jedes Verzeichnis wird ein Inode alloziert. Dabei versucht das Ext3-Dateisystem, die Daten des allozierten Inodes in der selben Blockgruppe unterzubringen, damit die I/O-Operationen performant bleiben [2]. Inodes sind von 1 beginnend durchnummeriert. Der Wert 0 wird verwendet, um Inodes als nicht alloziert oder ungültig zu kennzeichnen. Inode-Tabellen beinhalten eine Menge von Inodes, deren Anzahl ebenfalls im Superblock ausgelesen werden kann. Bei der Erstellung eines Dateisystems durch das mkfs-Programm berücksichtigt dieses bei der Anzahl der Inodes pro Blockgruppe einen Wert, der *Inodeverhältnis* genannt wird. Dieser repräsentiert die durchschnittliche Anzahl an Bytes, die von einem Inode belegt werden, wodurch sich die Gesamtzahl der Inodes als Quotient aus der Größe des Dateisystems und des Inodeverhältnisses errechnet.

Die Stammversion der Inodes besitzt eine Länge von 128 Byte. Die Inodes werden sequentiell durchgezählt in der Inode-Tabelle ihrer jeweiligen Blockgruppe gespeichert. Die Nummer eines speziellen Inodes hängt allein von der Position des Inodes in seiner jeweiligen Tabelle ab; somit werden Blockgruppen Bereiche von Inodenummern zugewiesen. Wenn eine Blockgruppe beispielsweise wenige, sehr große Dateien beinhaltet, werden nicht zwingend alle Inodenummern dieser Gruppe vergeben, bevor eine neue Blockgruppe benötigt wird.

Es existieren spezielle Inodes, die für festgelegte Funktionen reserviert sind und nicht vom Nutzer alloziert werden können. Diese sind standardmäßig Inode 1 bis Inode 10. Zwei dieser Inodes zeichnen sich durch besondere Wichtigkeit aus: Das Wurzelverzeichnis (standardmäßig Inode 2) und der Journal-Inode (standardmäßig Inode 8), mittels dem sich das in Ext3 hinzugekommene Journal finden lässt.

Unabhängig von ihren Funktionen sind alle Inodes aufgebaut, wie es in Tabelle 2.3 skizziert ist. Einige der für die vorliegende Arbeit interessanten Felder von Inodes werden im Folgenden näher beleuchtet.

Der erste Eintrag eines Inodes besteht aus dem Datei-Modus, der sich aus den Zugriffsrechten für die Datei und ihrem Dateityp zusammensetzt. Die Zugriffsrechte sind Linux-typisch aufgeteilt auf die Rechte des Besitzers, von Gruppen und aller Nutzer. Für diese Berechtigungsrollen sind jeweils Lese-, Schreib- und Ausführungsrechte definiert, woraus sich die unteren 9 Bits dieses Feldes ergeben. Die nächsten 3 Bits betreffen ausführbare Dateien und Verzeichnisse, da diese das Verhalten bei der Ausführung beeinflussen. Diese Bits enthalten Flags für das *Sticky Bit* und das Setzen der Gruppen- und User-ID. Die obersten 4 Bits beschreiben den Dateityp des Inodes. Ein Inode kann ausschließlich einen der 7 Dateitypen annehmen, die in Tabelle 2.4 aufgelistet werden.

Je nach Dateityp ist der Inhalt, auf den der Inode zeigt, unterschiedlich zu behandeln. Reguläre Dateien und Verzeichnisse, aber auch Geräte (Devices) wie Monitore, USB- und Eingabegeräte sind über Inodes ansprechbar. Bei einem symbolischen Link beschreibt der Inhalt in den Datenblöcken einen Verweis auf den Inode der Datei, zu der der Link gehört. Wenn die Datei eine reguläre Datei ist, so wird in den Datenblöcken der Dateiinhalt abgespeichert.

Weiterhin beinhaltet ein Inode mehrere Zeitstempel. Bei Ext3 gibt es für verschiedene Operationen, die auf einer Datei ausgeführt werden können, verschiedene Zeitstempel. Der Zeitstempel für den letzten Zugriff (*atime*) auf die Datei beschreibt den letzten dokumentierten Zeitpunkt, zu dem auf die Datei lesend zugegriffen wurde. Die letzte Änderung (*ctime*) dagegen bezieht sich auf die Veränderung der Metadaten der Datei, wie z.B. den Zugriffsrechten. Der Zeitstempel für die Modifikation (*mtime*) ändert sich bei einer Änderung des Dateiinhalts und der Zeitstempel für die Löschung zeigt den Zeitpunkt, wann die Datei gelöscht wurde.

Byte-Offset	Länge in Byte	Beschreibung
0	2	Datei-Modus (Dateityp und Zugriffsrechte)
2	2	Untere 16 Bit der User-ID
4	4	Untere 32 Bit der Dateigröße in Byte
8	4	Zeitstempel des letzten Dateizugriffs
12	4	Zeitstempel der letzten Änderung
16	4	Zeitstempel der letzten Modifikation
20	4	Zeitstempel der Löschung
24	2	Untere 16 Bit der Gruppen-ID
26	2	Anzahl der Links
28	4	Anzahl der Sektoren
32	4	Flags
36	4	Unbenutzt
40	48	12 direkte Blockzeiger
88	4	1 einfach indirekter Blockzeiger
92	4	1 doppelt indirekter Blockzeiger
96	4	1 dreifach indirekter Blockzeiger
100	4	Generationsnummer (NFS)
104	4	Erweiterter Attributblock (Datei-ACL)
108	4	Obere 32 Bits der Dateigröße / Verzeichnis-ACL
112	4	Blockadresse des Fragments
116	1	Fragmentindex im Block
117	1	Fragmentgröße
118	2	Unbenutzt
120	2	Obere 16 Bit der User-ID
122	2	Obere 16 Bit der Gruppen-ID
124	4	Unbenutzt

Tabelle 2.3: Aufbau eines Inodes in der Inode-Tabelle

Alle Zeitstempel geben die vergangene Zeit seit dem 01.01.1970 um 00:00:00 UTC in Sekunden an. Da die Werte 4 Byte groß sind, werden diese Zeitstempel im Januar 2038 überlaufen und können unter Umständen das Verhalten verschiedener Programme beeinträchtigen.

Die Anzahl der Links eines Inodes gibt an, wie viele *Hard-Links* auf ihn zeigen. Ein Hard-Link ist ein Verweis von Verzeichniseinträgen auf Inode-Metadatenstrukturen. Wenn mehrere Verzeichniseinträge auf den gleichen Inode verweisen, so merkt sich das der Inode in diesem Wert. Verzeichniseinträge stellen einen in den Datenblöcken gelegenen Metadatenmechanismus dar, der die Verzeichnishierarchie außerhalb der Inodes repräsentiert und auch zur Speicherung von Dateinamen genutzt wird. Das Thema der Verzeichniseinträge wird im späteren Verlauf dieser Arbeit noch genauer beleuchtet werden.

Die Flags eines Inodes enthalten zusätzliche, dateispezifische Parameter. Diese können für jeden Inode unterschiedlich gesetzt sein und müssen vom Dateisystem dementsprechend behandelt werden. Flags zeigen beispielsweise an, ob der Dateiinhalt von Ext3 im Journal dokumentiert, der Zeitstempel für den Dateizugriff aktualisiert und, ob sogenannte Hash-Index-Verzeichniseinträge genutzt werden sollen.

Zu beachten ist, dass weder die Inodenummer noch der Dateiname oder ihre Lage in der Verzeichnishierarchie im Inode selbst gespeichert werden. Durch die physikalische Adresse des Inodes und den Aufbau des Dateisystems, gegeben durch die Metadaten im Superblock und die Gruppen-Deskriptor-Tabelle, lässt sich die Inodenummer eindeutig bestimmen. Insofern wäre es redundant, diese noch zusätzlich im Inode abzuspeichern und somit zusätzlichen Speicherplatz zu gebrauchen. Der Name der Datei wird in Verzeichniseinträgen gespeichert, zusammen mit der Zuordnung zur Inodenummer. Es wäre also zusätzlich redundant, den Namen der Datei im Inode abzuspeichern; darüber hinaus muss der Inode so weder dynamische Größe, noch ein großes, dünnbesetztes Namensfeld unterstützen.

Hex-Wert	Beschreibung
0x1	FIFO
0x2	Character-Device
0x4	Verzeichnis
0x6	Block-Device
0x8	reguläre Datei
0xa	symbolischer Link
0xc	UNIX-Socket

Tabelle 2.4: Dateitypen, die ein Inode annehmen kann

Die Einträge im Inode, die auf den Inhalt der Datei verweisen, werden Blockzeiger genannt. Diese Datenstrukturen erlauben es, mehrere Datenblöcke zu adressieren, selbst, wenn diese über das Dateisystem verstreut sind [30], wie in einem fragmentierten Dateisystem. Die Blockzeiger befinden sich ab Byte 40 im Inode. Beispielhaft skizziert wird die Struktur der indirekten Blockzeiger in Abbildung 2.3.

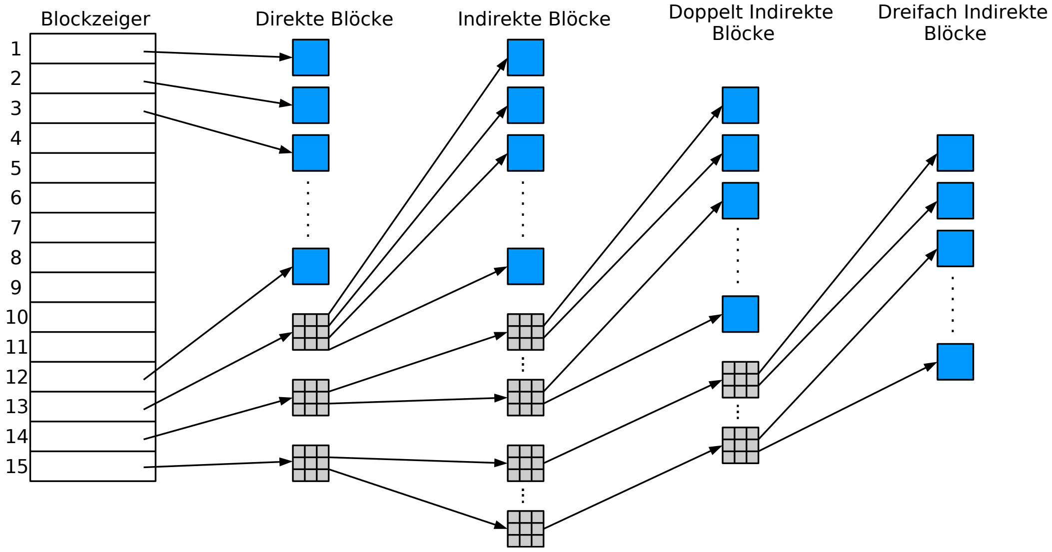


Abbildung 2.3: Struktur der indirekten Blockzeiger in einem Ext2/3-Dateisystem

In der Abbildung ist auf der linken Seite zu sehen, wie die Blockzeiger im Inode angeordnet sind. Insgesamt befinden sich 15 Einträge im Inode, die jeweils 4 Byte groß sind. Somit werden durch diese Struktur 60 Byte eingenommen, wodurch annähernd die Hälfte des Speicherplatzbedarfs eines Inodes dieser Datenstruktur zukommt.

Die ersten 12 Blockzeiger referenzieren direkt Datenblöcke. Je nach Blöckgröße lassen sich somit Dateien einer Größe von bis zu 48 KiB anlegen. Wenn eine Datei mehr als 48 KiB Platz benötigt, so wird durch den 13. Blockzeiger, den sogenannten indirekten Blockzeiger, auf einen Block verwiesen. Dieser ist ebenfalls ein Datenblock, beinhaltet allerdings weitere Zeiger auf weitere Datenblöcke. Bei einer Blockgröße von 4096 Byte können in diesem Datenblock bis zu 1024 weitere Blockzeiger gespeichert werden. Durch diese Indirektion ist es möglich, Dateien einer Größe von bis zu 4 MiB + 48 KiB abzuspeichern.

Dieser Indirektionsmechanismus wird im 14. Blockzeiger mit doppelter und im 15. Blockzeiger mit dreifacher Tiefe wiederholt. Da in einem 4096 Byte großen Block insgesamt 1024 Adressen à 4 Byte Platz finden, multipliziert sich für jede Indirektionsstufe die Speicherkapazität mit 1024, und somit ergibt sich als maximale Dateigröße 4 TiB + 4 GiB + 4 MiB + 48 KiB. Dies ist jedoch ein theoretisches Maximum; 32-Bit-Linux-Betriebssysteme begrenzen Dateigrößen auf 2 TiB [1].

Diese Kapazität erkauft sich Ext3 mit hohem Referenzierungsaufwand: Selbst bei zusammenhängenden Dateien, bei denen ein Zeiger theoretisch ausreichen würde, müssen alle Blockzeiger separat gesetzt werden. Für jeden allozierten Block sind mindestens 4 Byte für die direkten Blockzeiger notwendig; bei großen Dateien entsteht zusätzlicher Platzbedarf für indirekte Blockzeiger.

Verzeichnisse

Abhängig vom Dateityp, dem ein bestimmter Inode zugehört, ist der Inhalt seiner Datenblöcke unterschiedlich geartet. Falls der Inode ein Verzeichnis darstellt, so befinden sich in den Datenblöcken Verzeichniseinträge, die dem in Tabelle 2.5 skizzierten Aufbau entsprechen.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Inodenummer
4	2	Länge des Verzeichniseintrags
6	2	Länge des Namens
8	4–255	Name in ASCII

Tabelle 2.5: Aufbau eines Verzeichniseintrags

Ein Verzeichniseintrag beginnt auf einem Ext2/3-Dateisystem mit der Inodenummer. Durch diese lässt sich der Verzeichniseintrag mit der Datei, die er in dem Verzeichnis repräsentieren soll, in Verbindung bringen. Anschließend folgt die Länge des Verzeichniseintrags, die mindestens 12 Byte betragen muss. Dieser Eintrag wird ebenfalls dazu genutzt, um leeren Platz in einem Datenblock zu überspringen, der etwa durch gelöschte Einträge und die so hinterlassenen Lücken entstehen kann. Der letzte Verzeichniseintrag in einem Datenblock hat eine Länge, die dem gesamten restlichen Datenblock entspricht. Erstreckt sich ein Verzeichnis über mehrere Blöcke, so kann dieser Mechanismus zu mehreren Verschnitten am Blockende führen. Da der Verzeichniseintrag ungenutzte Daten enthalten kann, wird noch ein Attribut mit der Länge des Namens benötigt, damit auch sichergestellt ist, dass der Name separat gelesen werden kann und die Länge des Verzeichniseintrags davon unabhängig ist.

Auch der HTree-Mechanismus nutzt die Länge des Verzeichniseintrags, um seine Baumstruktur vor dem linearen Durchsuchen des Verzeichnisses zu verstecken. So wird gewährleistet, dass HTree-Strukturen unabhängig von der Version des Ext-Dateisystems ausgelesen werden können. Ext2-Mechanismen können so zwar die Vorteile des HTrees nicht nutzen, sind aber mit ihm kompatibel, da die Blätter des Baums herkömmlichen Verzeichniseinträgen entsprechen.

Dadurch, dass die Länge des Dateinamens auf 255 Byte beschränkt ist, reicht es auch aus, die Länge des Dateinamens mit einem Byte zu beschreiben. Deswegen gibt es noch eine andere Darstellungsweise für Verzeichniseinträge, wie er in Tabelle 2.6 gezeigt wird.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Inodenummer
4	2	Länge des Verzeichniseintrags
6	1	Länge des Namens
7	1	Dateityp
8	4–255	Name in ASCII

Tabelle 2.6: Aufbau eines Verzeichniseintrags

In dieser Darstellungsweise wird das frei gewordene Byte zur Kennzeichnung des Dateityps verwendet. Die Dateitypen, die hier angegeben werden, sind die selben, wie sie auch im entsprechenden Inode mit der dazugehörigen Inodenummer zu finden sind und in Tabelle 2.4 aufgelistet werden. Die konkreten Werte für dieses Byte, die diese Dateitypen repräsentieren, sind jedoch andere, wie es in Tabelle 2.7 zu sehen ist.

Im Falle eines unbekannten Dateityps unterscheidet sich der Verzeichniseintrag nicht von einem Eintrag ohne Beachtung des Dateityps. Somit sind beide Versionen miteinander kompatibel.

Verzeichnisse bestehen aus Verzeichniseinträgen. Diese werden linear nacheinander aufgelistet und folgen, bis auf zwei Ausnahmen, keiner definierten Ordnung. Jedes Verzeichnis beginnt mit den Einträgen für '.'

Wert	Beschreibung
0	Unbekannter Typ
1	reguläre Datei
2	Verzeichnis
3	Character-Device
4	Block-Device
5	FIFO
6	UNIX-Socket
7	Symbolischer Link

Tabelle 2.7: Werte für den Dateityp in einem Verzeichniseintrag

und '..', in dieser Reihenfolge. Hierbei gilt für mehrere Blöcke umfassende Verzeichnisse, dass diese beiden Einträge nicht in jedem Block wiederholt werden. In Listing 2.1 wird beispielhaft das Wurzelverzeichnis für ein Ext4-Image gezeigt, auf dem die Linux-Distribution Ubuntu installiert wurde.

```
0x023ac000  02 00 00 00 0c 00 01 02  2e 00 00 00 02 00 00 00  |................|
0x023ac010  0c 00 02 02 2e 2e 00 00  0b 00 00 00 14 00 0a 02  |................|
0x023ac020  6c 6f 73 74 2b 66 6f 75  6e 64 00 00 01 fd 01 00  |lost+found......|
0x023ac030  0c 00 03 02 65 74 63 00  01 ee 0b 00 10 00 05 02  |....etc.........|
0x023ac040  6d 65 64 69 61 00 00 00  01 fa 03 00 0c 00 03 02  |media...........|
0x023ac050  62 69 6e 00 01 f4 07 00  0c 00 04 02 62 6f 6f 74  |bin.........boot|
0x023ac060  01 f1 09 00 0c 00 03 02  64 65 76 00 01 f7 05 00  |........dev.....|
0x023ac070  0c 00 04 02 68 6f 6d 65  03 fd 01 00 0c 00 03 02  |....home........|
0x023ac080  6c 69 62 00 02 fa 03 00  10 00 05 02 6c 69 62 36  |lib.........lib6|
0x023ac090  34 00 00 00 02 f7 05 00  0c 00 03 02 6d 6e 74 00  |4...........mnt.|
...
```

Listing 2.1: Hexdump-Ausschnitt des Wurzelverzeichnisses eines Ubuntu-Images

Am Beginn des Verzeichnisses, an der Adresse `0x023ac000`, beginnt der erste Eintrag des Verzeichnisses. Dieser stellt das betreffende Verzeichnis selbst, bezeichnet als '.', dar und ist rot hervorgehoben. Im Vergleich mit Tabelle 2.6 kann man in den ersten 4 Byte die Inodenummer erkennen. Da dieser Hexdump im Little-Endian-Format notiert ist, entspricht der Wert der Inodenummer `0x00000002` und somit 2. Da die Inodenummer 2 eine reservierte Nummer ist, kann man daraus erkennen, dass der betrachtete Verzeichniseintrag das Wurzelverzeichnis beinhaltet. Die folgenden 2 Byte beinhalten die Länge des Verzeichniseintrags, hexadezimal `0x000c`, damit 12 Byte.

Die nächsten 2 Byte stellen nun den Unterschied zwischen Tabelle 2.5 und Tabelle 2.6 dar. Wie schon erwähnt, kann ein Dateiname nicht länger als 255 Byte lang sein. Wenn die 2 Bytes `0x0201` als Länge interpretiert werden würden, so ergäbe dies eine Namenslänge von 513 Byte. Allerdings trifft in diesem Fall die Teilung dieser 2 Byte-Werte zu, so dass `0x01` die Länge des Dateinamens und `0x02` der Dateityp ist. Wie Tabelle 2.7 entnommen werden kann, steht `0x02` für ein Verzeichnis.

An letzter Stelle steht in der Bytefolge der Dateiname. Aus den bisherigen Werten kann erschlossen werden, dass der Name 1 Byte lang ist. So erschließt sich der Name aus dem Hexwert `0x2e`, der in ASCII '.' entspricht. Der restliche Verzeichniseintrag ist mit Nullen aufgefüllt, da der nächste Eintrag erst beim 13. Byte beginnt, denn die Länge des Verzeichniseintrags beträgt minimal 12.

Für das blau gekennzeichnete Feld gelten die gleichen Bedingungen wie für den ersten Eintrag. Hier wird die Inodenummer, die ersten 4 Bytes aus dem blau markierten Feld, als `0x00000002` herausgelesen. Die Länge des Verzeichniseintrags ist ebenfalls 12 Byte lang und vom Dateityp ein Verzeichnis, nur, dass dieses Mal die Namenslänge 2 Byte beträgt. So können als Dateiname die nächsten 2 Bytes als Name interpretiert werden. Ihr Wert lautet in ASCII interpretiert '..' und bezeichnet damit das Elternverzeichnis des Wurzelverzeichnisses. Da das Wurzelverzeichnis allerdings kein Elternverzeichnis hat, verweist die Elternreferenz des Wurzelverzeichnisses typischerweise auf sich selbst. Dies stellt den einzigen Fall dar, in dem '.' und '..' die gleiche Inodenummer zugewiesen sein kann.

Nach diesen zwei Einträgen folgen unsortiert die restlichen Einträge des Verzeichnisses. Diese sind ebenfalls nach dem Schema aus Tabelle 2.6 aufgebaut und können auch so interpretiert werden.

Wie anfangs bereits kurz erwähnt wurde, kann in einem Inode ein Flag gesetzt werden, ob Hash-Index-Verzeichniseinträge genutzt werden sollen. Dies eignet sich für die Suche nach bekannten Dateinamen in großen Verzeichnissen, da sie auf einem modifizierten B-Baum basiert, dessen Schlüsselwerte Hashsummen der Dateinamen sind. Die Verzeichnisstruktur muss folglich um innere Knoten dieser Baumstruktur und zusätzliche Metadaten über den Baum erweitert werden. Durch die Hash-Sortierung werden die Verzeichniseinträge nicht linear und unsortiert in die Datenblöcke geschrieben, sondern sortiert in einer Baumstruktur abgelegt. Dabei ist jeder Datenblock, der einem Verzeichnis zugeordnet ist, entweder ein Blatt oder innerer Knoten in diesem Baum. Alle Datenblöcke, die einen inneren Knoten in der Baumstruktur darstellen, verweisen auf nach Dateinamenhashes basierte Weise auf Datenblöcke. Blätter beinhalten ausschließlich herkömmliche Verzeichniseinträge.

Diese Datenstruktur beginnt nach den Einträgen für '.' und '..', da diese per Standard immer den Beginn des Verzeichnisses darstellen müssen. Dabei wird die Länge für den Verzeichniseintrag von '..' auf die Blockgröße minus 12 Byte gesetzt, damit es so aussieht, als würde nach diesem Eintrag nichts mehr folgen. Das sorgt dafür, dass das lineare Auslesen von Verzeichniseinträgen unberührt bleibt, und trotzdem die Hash-Index-Verzeichnisstruktur genutzt werden kann. Auch innere Knoten verstecken auf diese Weise ihre Metadatenstruktur. Dies ist allerdings für Ext3 ein optionales Feature, das 2002 von Phillips [27] implementiert und von Brian Carrier in seinem Buch [4] für Ext2/3 erklärt wurde.

Journal

Das Journal beim Ext3-Dateisystem zeichnet transaktional Aktualisierungen an Metadaten auf, so dass das Dateisystem nach einem Absturz einfacher wiederhergestellt werden kann. Das Journal hat, wie jede andere Datei auch, einen Inode – standardmäßig Inode 8 – der auf die Datenblöcke verweist, in denen das Journal die Änderungen des Dateisystems mitprotokolliert. Dabei unterteilt sich das Journal in vier verschiedene Datenstrukturen.

- Journal-Superblock
- Deskriptor-Blöcke
- Commit-Blöcke
- Revoke-Blöcke

Jede dieser Datenstrukturen hat eine eigene Signatur, damit zwischen normalen Journal-Blöcken und administrativen Journal-Blöcken unterschieden werden kann. Der Journal-Superblock befindet sich im ersten Block des Journals und beinhaltet die Metadaten des Journals. Deskriptor-Blöcke befinden sich vor den geloggten Daten und beinhalten ihre Zieladresse und eine Sequenznummer. Commit-Blöcke markieren geloggte Transaktionen als bereits auf der Festplatte gesichert. Revoke-Blöcke markieren Blöcke, welche nicht mehr geloggt werden. Transaktionen dieser Blöcke mit geringerer Sequenznummer als der des Revoke-Blocks sind nicht als gültig zu verstehen und für die Wiederherstellung nicht in Betracht zu ziehen. Im Gegensatz zu den anderen Ext2/3-Datenstrukturen ist das Journal im Big-Endian-Format beschrieben.

Da im Journal Änderungen an Metadaten auf dem Dateisystem festgehalten werden, tauchen hier auch die Metadaten von Inodes wieder auf. So kann es passieren, dass ein Inode mehrfach auf dem Dateisystem vorkommt, z.B. wenn die Zugriffsrechte geändert wurden und deswegen ein Eintrag dafür im Journal entsteht. Jedoch befinden sich diese Inodes in den Datenblöcken des Journals, weswegen dort keine Verwechslung mit einem regulären Inode entstehen kann, da diese sich ausschließlich in den Inode-Tabellen befinden können.

2.1.2 Das Ext4-Dateisystem

Ext2/3 war sehr beliebt als Linux-Dateisystem, da es sehr zuverlässig war, viele Features bot, gute Performance leistete und zwischen seinen Versionen gut kompatibel war. Allerdings skaliert Ext3 aufgrund seiner

Einschränkungen, wie etwa der der maximalen Dateisystemgröße von 16 TiB, auf großen Systemen nicht ausreichend gut. Um Schwächen bezüglich der Skalierbarkeit auszugleichen, wurde 2007 der Nachfolger, Ext4, ins Leben gerufen. Mit dieser Nachfolgerversion sollten außerdem die Performance und Zuverlässigkeit verbessert werden [24].

Die folgenden Informationen, die in diesem Kapitel präsentiert werden, stammen größtenteils vom Ext4-Wiki [2].

Ext4 führt einen 64-Bit-Modus ein, in dem Blockadressen 64 Bit breit sind. Dieser ist standardmäßig deaktiviert und muss manuell bei der Erstellung des Dateisystems gefordert werden; Im Superblock wird diese Einstellung in einem Flag gespeichert. In Tabelle 2.8 werden sowohl für den 32-Bit-Modus, als auch für den 64-Bit-Modus die Wertebereiche für Dateisystemgrenzen aufgezeigt, abhängig von der Blockgröße b. Die erlaubten Blockgrößen entsprechen denen aus Ext2/3, inklusive des Spezialfalls 8 KiB.

Beschreibung	Wert (32 Bit)	Wert (64 Bit)
Anzahl Blöcke im Dateisystem	2^{32}	2^{64}
Anzahl Inodes im Dateisystem	2^{32}	2^{32}
Anzahl Blöcke pro Blockgruppe	$8 \cdot b$	$8 \cdot b$
Anzahl Inodes pro Blockgruppe	$8 \cdot b$	$8 \cdot b$
Anzahl Blöcke pro Datei (Extents)	2^{32}	2^{32}

Tabelle 2.8: Maximalwerte für dateisystemspezifische Einschränkungen

Die maximale Dateisystemgröße errechnet sich aus der maximalen Anzahl an Blöcken im Dateisystem multipliziert mit der Blockgröße, also $2^{32} \cdot b$ oder $2^{64} \cdot b$, je nach Bitbreite. Das entspricht bei 4096 Byte als Blockgröße einer Größe von 16 TiB bzw. 64 ZiB für das Dateisystem.

Unter Ext4 wird aus Kompatibilitätsgründen die Struktur mit den indirekten Blockzeigern von Ext2/3 weiterhin unterstützt. Zusätzlich dazu bietet Ext4 einen neuen Mechanismus zur Referenzierung von Datenblöcken an: Die sogenannten *Extents*. Diese können mehr Speicher adressieren als indirekte Blockzeiger. Der in der eben zitierten Tabelle genannte Wert bezieht sich auf die Nutzung von Extents. Ext4 nutzt indirekte Blockzeiger jedoch ausschließlich in dem Fall, in dem ein bestehendes Ext2/3-Dateisystem als Ext4 reformatiert wird. Es gibt keine Möglichkeit, ein neues Ext4-System mit indirekten Blockzeigern zu erstellen.

Aufbau

Der Aufbau des Ext4-Dateisystems hat sich, gegenüber seinem Vorgänger nur an wenigen Stellen geändert. Auch Ext4 baut auf Blöcken und Blockgruppen auf, an deren Anfängen Metadatenstrukturen zu finden sind, wie auch in Abbildung 2.2 zu sehen ist.

Das mkfs-Tool, ein Standard-Linux-Programm, das zum Formatieren von Festplatten benutzt wird, legt bei neuen Ext4-Dateisystemen nach der Gruppen-Deskriptor-Tabelle einen Bereich für weitere Gruppen-Deskriptoren an, um dem Dateisystem das Wachsen zu ermöglichen. Durch diesen reservierten Speicherbereich können nachträglich weitere Blockgruppen alloziert und dort eingetragen werden, wodurch die Gruppen-Deskriptor-Tabelle weiter wächst. Dabei kann das Dateisystem maximal um das 1024-fache seiner bereits vorhandenen Größe wachsen, jedoch wird Gruppen-Deskriptoren nie mehr Platz als 1024 Datenblöcke zugesprochen.

Mit Ext4 wurde ein zusätzlich ein neues Feature eingeführt, das der sogenannten *Flex-Gruppen*. Wenn dieses Feature benutzt wird, so kann dies in einem Flag im Superblock ausgelesen werden. Flex-Gruppen fassen mehrere Blockgruppen zu einer logischen Blockgruppe zusammen. Hierbei werden die Daten- und Inode-Bitmaps und die Inode-Tabellen mehrerer Blockgruppen ausschließlich in der ersten Blockgruppe der Flex-Gruppe gesammelt gespeichert. Es wird zusätzlicher Platz für diese Metadatenstrukturen in der ersten Blockgruppe alloziert, dafür folgen in den restlichen Blockgruppen der Flex-Gruppe keinerlei Bitmaps oder Inode-Tabellen. Dadurch wird das Suchen nach Dateien effizienter, da die Metadaten örtlich stärker konzentriert sind.

Flex-Gruppen beinhalten eine festgelegte Anzahl von Blockgruppen. Diese Anzahl muss eine Zweierpotenz sein, etwa 4 oder 16. Wenn beispielsweise eine Flex-Gruppengröße von 4 angenommen wird, so würde in der ersten Blockgruppe die Daten- und Inode-Bitmaps und Inode-Tabellen für 4 Blockgruppen gespeichert werden. Standardmäßig werden Reihen von gleichartigen Metadatenstrukturen angelegt; im Beispiel etwa zunächst 4 Daten-Bitmaps, dann 4 Inode-Bitmaps und anschließend 4 Inode-Tabellen, jedoch wird ein gestaffelter Aufbau unterstützt, der gegebenenfalls in der Gruppen-Deskriptor-Tabelle ausgelesen werden kann. In den drei weiteren Blockgruppen dieser Flex-Gruppe wird zu Beginn nur der Superblock und die Gruppen-Deskriptor-Tabelle vor den Datenblöcken gespeichert, so wie es in Abbildung 2.4 gezeigt wird. Die restlichen Blöcke der Blockgruppe werden somit als Datenblöcke genutzt.

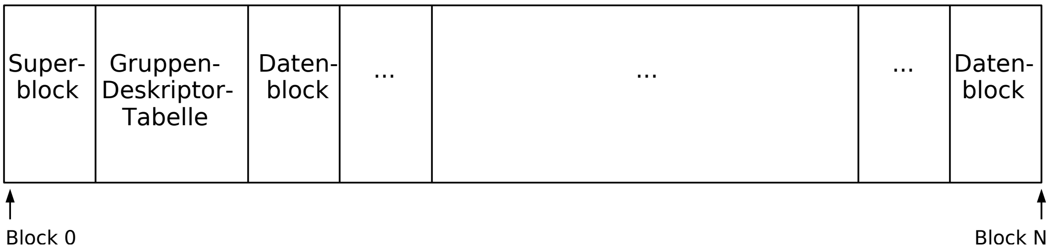


Abbildung 2.4: Aufbau einer Blockgruppe mit dem Flex-Gruppen-Feature

Wie bereits erwähnt, besteht ein Vorteil der Flex-Gruppen darin, dass durch das Zusammenfassen der Metadaten, diese schneller geladen werden können. Zusätzlich können große Dateien zusammenhängend abgespeichert werden, da mehr zusammenhängende Datenblöcke zur Verfügung stehen. Nun kann das Flex-Gruppen-Feature mit dem Sparse-Superblock-Feature kombiniert werden. Dadurch entstehen Blockgruppen, die ausschließlich aus Datenblöcken bestehen und keinerlei Metadaten beinhalten.

Dies kommt dadurch zustande, dass Flex-Gruppen in 2er-Potenzen arrangiert werden und Sparse-Superblöcke in Potenzen von 3, 5 und 7. Somit ist die einzige Blockgruppe, in denen sowohl Superblock und Gruppen-Deskriptor-Tabelle, als auch die Bitmaps und Inode-Tabelle vorhanden sind, Blockgruppe 0. Durch das Sparse-Superblock-Feature treten anschließend in Blockgruppe 1 der Superblock und die Gruppen-Deskriptor-Tabelle auf, wie es auch in Abbildung 2.4 gezeigt wird. Je nach Größe der Flex-Gruppe kann nun in Blockgruppe 2 eine neue Flexgruppe anfangen, so dass dort wieder Bitmaps und Inode-Tabelle stehen, oder die folgenden Blockgruppen der Flex-Gruppe bestehen ausschließlich aus Datenblöcken, bis auf jene Blockgruppen, die wieder durch das Sparse-Superblock-Feature Kopien des Superblocks und der Gruppen-Deskriptor-Tabelle beinhalten.

Im Jahr 2012 kamen zu Ext4 Metadaten-Prüfsummen hinzu, die die wichtigsten Ext4-Datenstrukturen abdecken. Ext4 unterstützt verschiedene Prüfsummenalgorithmen, von denen einer dateisystemweit gilt. Die Wahl des Algorithmus lässt sich im Superblock auslesen. Dabei beinhalten der Superblock, die Verzeichniseinträge, Extents, Bitmaps, Inodes und Gruppen-Deskriptoren Prüfsummen, die meist 32 Bit lang sind und sich aus verschiedenen Attributen der jeweiligen Datenstruktur errechnen. Zur Gewährleistung der Abwärtskompatibilität wurde die Struktur des Superblocks von Ext2/3 zu Ext4 unwesentlich verändert. Es wurden einige unbenutzte Blöcke, die es im Superblock gab, in Ext4 mit weiteren Attributen versehen, wie beispielsweise der Prüfsumme.

Die Einträge der Gruppen-Deskriptor-Tabelle, deren Aufbau in Tabelle 2.1 bereits gezeigt wurde, wurden in Ext4 erweitert. Dabei sind die ersten 18 Bytes unverändert geblieben. Der unbenutzte Speicherbereich, der anschließend folgt, wird nun wie in Tabelle 2.9 verwendet.

Die Flags, die in Byte 18 gespeichert werden, beinhalten Werte darüber, ob die Inode-Tabelle und die Bitmaps initialisiert wurden, ob die Block-Bitmap im Speziellen nicht initialisiert wurde und, ob die Inode-Tabelle mit Nullen überschrieben wurde. Die meisten übrigen Attribute stellen diverse Prüfsummen dar. Falls der 64-Bit-Modus für das Dateisystem aktiviert ist, so sind Einträge in der Gruppen-Deskriptor-Tabelle 64 statt 32 Byte groß. Dadurch ergeben sich weitere 32 Byte, in denen für die meisten Einträge, die in Tabelle 2.1 und in Tabelle 2.9 zu sehen sind, jeweils zusätzliche obere Bits abgespeichert werden. Die auch im 32-Bit-Modus verfügbaren Prüfsummen werden dann als untere Bits verstanden.

Byte-Offset	Länge in Byte	Beschreibung
18	2	Flags für die Blockgruppe
20	4	(Untere) 32 Bit der Adresse für Snapshot-Exclusion-Bitmap
24	2	(Untere) 16 Bit der Block-Bitmap-Prüfsumme
26	2	(Untere) 16 Bit der Inode-Bitmap-Prüfsumme
28	2	(Untere) 16 Bit der Anzahl nicht allozierter Inodes
30	2	Gruppen-Deskriptor-Prüfsumme

Tabelle 2.9: Aufbau eines Eintrags in der Gruppen-Deskriptor-Tabelle in Ext4

Da durch den 64-Bit-Modus ein größerer Speicherbereich angesprochen werden kann, reichen 32 Bit für die Adressierung auf der Festplatte nicht aus. Damit die bestehende, 32 Byte lange Struktur nicht nachträglich verändert werden muss, werden die nun auf 64 Bit verlängerten Werte in mehreren Feldern gespeichert. Diese werden beim Auslesen entsprechend zusammengesetzt.

Inodes

Inodes sind auch bei Ext4 die zentrale Datenstruktur, die für jede Datei und jedes Verzeichnis angelegt wird. Sie werden, wie auch bei den Vorgängerversionen, in Inode-Tabellen abgelegt. Aus Kompatibilitätsgründen wurde die Struktur eines Inodes nur marginal verändert. Jedoch wurden einige Speicherbereiche, die vorher ungenutzt waren, durch neue Attribute ersetzt, wie es in Tabelle 2.10 aufgezeigt wird.

Byte-Offset	Länge in Byte	Beschreibung
		...
36	4	Betriebssystemabhängige Einträge (vorher unbenutzt)
40	60	Indirekte Blockzeiger oder Extent-Datenstruktur
		...
116	12	Betriebssystemabhängige Einträge
128	2	Zusätzliche Größe für den Inode
130	2	Obere 16 Bit für die Inode Prüfsumme
132	4	Zusätzliche Bits für Zeitstempel für letzte Änderung
136	4	Zusätzliche Bits für Zeitstempel für letzte Modifikation
140	4	Zusätzliche Bits für Zeitstempel für letzte Dateizugriff
144	4	Zeitstempel für Erstellung der Datei
148	4	Zusätzliche Bits für Zeitstempel für Erstellung der Datei
152	4	Obere 32 Bit für Versionsnummer

Tabelle 2.10: Zusätzliche besetzte Attribute in einem Inodes in der Inode-Tabelle ab Ext4

Für die in Tabelle 2.3 ab Byte 32 notierten Flags hat sich in Ext4 der Wertebereich vergrößert. Unter Ext2/3 waren die meisten Flags unbenutzt oder noch nicht implementiert. Ext4 erweitert diese um Angaben wie etwa, ob ein Inode die Extent-Datenstruktur benutzt oder, ob Inhaltsdaten einer Datei immer durch das Journal geschrieben werden müssen.

Ab Byte 128 beginnen zusätzliche Informationen. Die Länge des zusätzlichen Bereichs ist in Byte 128 notiert. In einem kleinen Dateisystem hat Ext4 die Möglichkeit, diese zusätzlichen Attribute nicht zu nutzen, so dass ein Inode nur 128 Byte groß bleibt. In allen anderen Fällen wird als Standard der Inode mit 256 Byte angelegt. So steht auch mit erweiterten Attributen noch genug Platz zur Verfügung.

Nach der eben genannten Größenangabe folgen Prüfsummen und zusätzliche Einträge für die Zeitstempel. Durch diese Einträge ist es Ext4 möglich, die Zeitstempel um den Faktor 10^9 feiner, also im Nanosekundenbereich, zu erfassen. Zusätzlich zu den bereits vorhandenen mtime-, atime- und ctime-Zeitstempeln wird außerdem ein Erstellungszeitstempel (*crtime*) erfasst. Auch dieser kann nanosekundengenau abgespeichert werden.

Das Problem von Ext3, dass die Zeitstempel im Jahr 2038 überlaufen, wurde durch verschiedene Ansätze gelöst [31]. Einer davon besteht in der Idee, dass Einträge für die Zeitstempel in Ext4 als vorzeichenlose Ganzzahlen interpretiert werden, wodurch ein weiteres Bit zur Verfügung steht, das für die Zeitdokumentation benutzt werden kann. Dadurch verschiebt sich das Problem ins Jahr 2106. Die zusätzlichen Zeitstempel, die für die Nanosekunden-Genauigkeit hinzugezogen wurden, erweitern die Frist bis zum Überlauf noch zusätzlich. Da für die Nanosekunden-Genauigkeit nur 30 Bit notwendig sind, stehen die untersten 2 Bits dieser zusätzlichen Zeitstempel zur Verfügung, um das Zeitstempel-Feld für die Sekunden-Genauigkeit zu erweitern. Durch diese zusätzlichen 2 Bit wird dieses Problem bis ins Jahr 2514 verschoben.

Eine weitere Neuerung in Ext4 ist die Möglichkeit, sogenannte Inline-Dateien und Inline-Verzeichnisse in dem für erweiterte Attribute und Extents zur Verfügung gestellten Speicherplatz zu lagern. Dadurch ist es möglich, Dateien und Verzeichnisse, die sehr klein sind, innerhalb des Inodes zu speichern und nicht in einen Datenblock auszulagern. Hierbei wird der Speicherverbrauch minimiert und auch das Nachschlagen nach dem Dateiinhalt performanter. Es werden zuerst die 60 Byte, die sonst für die indirekten Blockzeiger oder die Extent-Datenstruktur reserviert sind, benutzt. Falls dieser Platz nicht ausreicht, können Inodes mit erweiterten Attributen ausgestattet werden und somit zusätzlichen Speicherplatz zur Verfügung stellen.

In allen anderen Fällen, in denen es sich lohnt, für eine Datei einen Datenblock anzulegen, wird im reinen Ext4-Dateisystem die Extent-Datenstruktur benutzt. Dies ist eine performantere Lösung als die Variante mit den indirekten Blockzeigern, da zusammenhängende Datenblöcke einfacher adressiert werden können und weniger Intialisierungsaufwand besteht [32]. In Abbildung 2.5 wird beispielhaft gezeigt, wie die Baumstruktur, die von den Extents aufgebaut wird, aussehen kann.

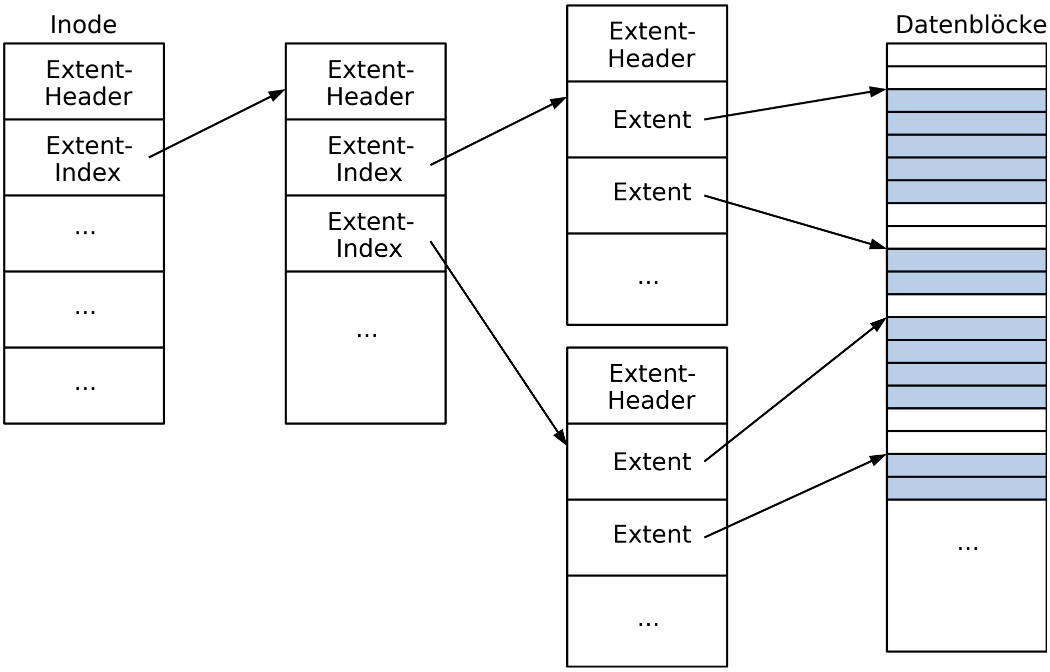


Abbildung 2.5: Extent-Datenstruktur in einem Ext4-Dateisystem

Der Inode der Beispieldatei ist links in der Abbildung zu sehen. Ihm stehen 60 Byte zur Verfügung, um seine Extent-Struktur zu verwalten. Ein Extent-Eintrag ist 12 Byte lang, womit sich Platz für 5 Einträge ergibt. Jede Extent-Datenstruktur beginnt jedoch mit einem Header der Länge 12 Byte, dessen Aufbau in Tabelle 2.11 aufgezeigt wird. Daraus ergibt sich im Inode Platz für 4 Extent-Einträge.

Zu Beginn des Headers steht eine *Magic Number*, mit der der Beginn einer Extent-Datenstruktur gekennzeichnet wird. Anschließend folgen Felder mit der Anzahl der tatsächlichen und maximal möglichen Extent-Einträge. Wenn dieser Header im Inode direkt steht, so lautet die maximal mögliche Anzahl an Einträgen 4, wie oben bereits erwähnt. Da Extents auf Datenblöcke verweisen, stehen in allen tieferen Stufen des Extent-Baumes die Einträge in Datenblöcken. In diesen können dem Header so viele 12-Byte Einträge fol-

Byte-Offset	Länge in Byte	Beschreibung
0	2	Magic Number (`0xf30a`)
2	2	Anzahl der validen Extent-Einträge nach dem Header
4	2	Maximale Anzahl der Extent-Einträge, die folgen können
6	2	Tiefe dieses Extent-Knotens im Extent-Baum
8	4	Generationsnummer

Tabelle 2.11: Extent-Header im Extent-Baum

gen, wie in einen Block passen. Die Einträge, die dem Header folgen können, werden in Tabelle 2.12 und in Tabelle 2.13 gezeigt. Alle Kinder eines Knotens haben die selbe Tiefe und sind innere Knoten oder Blätter. Beim aus diesen Einträgen aufgebauten Baum gilt eine Maximaltiefe von 5. Die dadurch umfassbare, maximale Speichermenge übersteigt den mit 32 Bit adressierbaren Speicherbereich, weshalb ein tieferer Baum schlichtweg keine Anwendung fände.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Nummer des ersten Blocks des eigenen Teilbaums ab Dateibeginn
4	4	Untere 32 Bit der Blockadresse des Kindeintrags
8	2	Obere 16 Bit der Blockadresse des Kindeintrags
10	2	Unbenutzt

Tabelle 2.12: Ein Extent-Eintrag für einen inneren Knoten im Extent-Baum

Innere Knoten halten die Information über den ersten Datenblock, den sie in ihrem Teilbaum erfassen. Diese Information wird ab dem ersten Datenblock der Datei, bei 0 beginnend, gezählt. Somit lautet dieser Wert für den logisch ersten Extent-Eintrag im Wurzelknoten immer 0. Innerhalb eines Knotens sind die Extent-Einträge nicht zwingend sortiert. Anschließend steht in zwei verschiedenen Einträgen die insgesamt 48 Bit große Blockadresse des referenzierten Kindknotens.

Bezüglich der Extent-Datenstruktur gelten Knoten mit der Tiefe 0 als Blätter; diese referenzieren Datenblöcke. Diese Blätter sind in Abbildung 2.5 in der Spalte vor den Datenblöcken zu sehen und werden dort als „Extent“ bezeichnet. Dies steht im Gegensatz zu den inneren Knoten, welche in der Terminologie von Ext4 als „Extent Index“ bezeichnet werden.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Nummer des ersten Blocks des eigenen Teilbaums ab Dateibeginn
4	2	Anzahl der Blöcke, die dieser Extent abdeckt
6	2	Obere 16 Bit der Blockadresse der referenzierten Datenblöcke
8	4	Untere 32 Bit der Blockadresse der referenzierten Datenblöcke

Tabelle 2.13: Ein Extent-Eintrag für ein Blatt im Extent-Baum

Innere Knoten und Blätter haben zwar die erste betroffene Blocknummer formattechnisch gemeinsam, sind aber trotz ihrer logischen Ähnlichkeit unterschiedlich strukturiert. Blätter besitzen eine explizite Angabe über die Anzahl der Blöcke, die sie abdecken. Hierbei gibt das oberste Bit an, ob der Extent uninitialisiert ist, wodurch für die tatsächliche Anzahl referenzierter Datenblöcke 15 Bit übrig bleiben. Die Teile der Blockadresse der Datenblöcke, die sie referenzieren, ist außerdem an anderen Stellen im Eintrag vorzufinden. Der Vergleich beider Strukturen folgt aus der Betrachtung der Tabellen 2.12 und 2.13.

Einschränkungen der Extent-Struktur sind allein durch die Anzahl der Bits gegeben. So kann ein Extent nie mehr als 2^{15} Datenblöcke abdecken, was bei einer Blockgröße von 4096 Byte nicht mehr als 128 MiB umfasst. Selbst, wenn mehrere Gibibyte an Daten zusammenhängend auf dem Dateisystem liegen, so müssen mehrere Extents alloziert werden, um den Dateiinhalt vollständig zu adressieren. Dennoch bewältigen Extents das Problem großer, zusammenhängender Dateien zufriedenstellender als indirekte Blockzeiger.

Verzeichnisse

Ext4-Verzeichnisse unterscheiden sich nicht von den aus Ext2/3 bekannten Versionen. Sie sind ebenfalls linear angeordnet und beginnen mit den Einträgen für '.' und '..'. Ungenutzte Verzeichniseinträge besitzen die Inodenummer 0, so dass diese übersprungen werden können und nicht als gültiger Verzeichniseintrag interpretiert werden. Da ein Verzeichnis über mehrere Datenblöcke verteilt sein kann, und in jedem Datenblock linear die Verzeichniseinträge abgelegt werden, musste für Ext4 ein Mechanismus gefunden werden, um Prüfsummen für Verzeichniseinträge einzubauen, ohne die Abwärtskompatibilität zu Ext2/3 zu verlieren. So wird als letzter Verzeichniseintrag pro benutztem Datenblock des Verzeichnis eine 12 Byte lange Struktur am Ende eingefügt, wie sie in Tabelle 2.14 gezeigt wird.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Inodenummer
4	2	Länge des Verzeichniseintrags
6	1	Länge des Namens
7	1	Dateityp
8	4	Prüfsumme des Verzeichnisblocks

Tabelle 2.14: Letzter Eintrag im Datenblock mit der Prüfsumme für Verzeichniseinträge

In diesem Fall ist die Inodenummer 0, damit dieser von älteren Versionen als ungenutzter Verzeichniseintrag gedeutet wird. Da 12 Byte die minimale Größe für einen Verzeichniseintrag ist, steht diese auch als Länge des Verzeichniseintrags in der Struktur. Da diese Struktur keinen Namen hat, wird die Länge des Namens mit 0 angegeben. Der Dateityp wird hierfür auf `0xde` festgelegt. Die letzten 4 Byte beinhalten schließlich die Prüfsumme für den Datenblock mit Verzeichniseinträgen.

Unter Ext3 wurde nachträglich ein Feature für Verzeichnisse entwickelt, um diese effizienter zu gestalten: die sogenannten Hash-Index-Verzeichniseinträge. Das Ext4-Dateisystem beinhaltet diese nun als offizielles Feature. Da unsortierte, linear aufgebaute Verzeichnisse bei der Suche nach bestimmten Einträgen lineare Laufzeit erfordern, gibt es die Möglichkeit, Verzeichnisse hierarchisch nach Hash-Werten sortiert in das Verzeichnis einzutragen. Dabei wird ein Baum aufgespannt, wodurch es ermöglicht wird, unbegrenzt viele Verzeichniseinträge pro Verzeichnis zu verwalten. Dies ist so erst in Ext4 möglich; Ext3 gibt eine Schranke von 32000 Verzeichniseinträgen pro Verzeichnis vor.

Im Flags-Attribut des Inodes, dem ein Verzeichnis zugeordnet wurde, kann ausgelesen werden, ob dieses Verzeichnis Hash-Index-Verzeichniseinträge benutzt. Durch diese Option wird ein modifizierter B-Baum, ein sogenannter H-Baum, aufgespannt, dessen Schlüsselwerte Hashsummen der gespeicherten Dateinamen sind. Zur Wahrung der Abwärtskompatibilität ist die dafür nötige Baumstruktur im Verzeichnis auf eine Weise versteckt, dass beim linearen Durchsuchen des Verzeichnisses alle Baum-Metadaten übersprungen werden. Dies wird durch die Erhöhung der Länge vorhergehender Einträge erreicht. In Tabelle 2.15 wird der Beginn des H-Baumes gezeigt, der nach den Einträgen für '.' und '..' steht.

Nach den zwei, vom Standard vorgegebenen, ersten Einträgen im Verzeichnis folgt die Metadatenstruktur, die den H-Baum beschreibt. Diese gliedert sich in einen Satz sogenannter Bauminformationen und eine Reihe von Referenzen auf Kindelemente, von denen die erste einen Spezialfall darstellt. Die Bauminformationen bestehen aus Metadaten, die für den gesamten Baum gelten – Ein reservierter Speicherbereich, der verwendete Hashsummenalgorithmus, die eigene Länge, die Tiefe des Baums und weiterer, ungenutzter Speicher. Die Länge der Bauminformationen ist hierbei konstant auf 8 Byte gesetzt.

Ab der zweiten Referenz auf Kindelemente sind diese wie folgt aufgebaut: Zuerst wird die untere Schranke von Hash-Werten, die der betroffene Teilbaum umfasst, angegeben. Die obere Schranke errechnet sich implizit durch die untere Schranke des nächsten Elements. Da der gesamte Baum den Wertebereich einer vorzeichenlosen 4-Byte-Ganzzahl aufspannt, sind die untere Grenze des ersten Elements (mit 0) und die obere Grenze des letzten Elements (mit 2^{32}) implizit bekannt. Dies bedeutet, dass für die erste Kindelementreferenz kein expliziter Hash-Wert angegeben werden muss. Darin begründet sich ihre besondere Position: Die so frei gewordenen 4 Byte werden verwendet, um die maximalen und tatsächlichen Anzahlen folgender Kindelementreferenzen anzugeben.

Byte-Offset	Länge in Byte	Beschreibung
24	4	Reservierter Bereich
28	1	Hash-Version
29	1	Länge der Bauminformation
30	1	Tiefe des Baumes
31	1	Unbenutzte Flags
32	2	Maximale Anzahl an Einträgen, die noch folgen können
34	2	Anzahl an Einträgen, die folgen
36	4	Nummer des Blocks innerhalb des Verzeichnisses
40	4	Minimaler Hash-Wert des zweiten Kindelements
44	4	Nummer des Blocks innerhalb des Verzeichnisses
		...

Tabelle 2.15: Wurzelknoten eines H-Baums

Um eine Datei mittels dieser Hash-Werte schnell zu finden, wird der Hash-Wert des Dateinamens, nach dem gesucht wird, berechnet und mit den Hash-Werten in den Einträgen verglichen. Durch die Sortierung der Einträge nach Hash-Werten und den hierarchischen Aufbau des Baums kann so der Block, in dem sich der gesuchte Dateiname befindet, in logarithmischer Laufzeit gefunden werden. Ist die Suche beim entsprechenden Blatt angekommen, kann der gefundene Block linear nach dem gewünschten Dateinamen durchsucht werden. Sonst zeigen die Einträge auf innere Knoten, die wieder mit einem Header beginnen, der dem linearen Algorithmus einen leeren Datenblock vorspielen, wie es in Tabelle 2.16 gezeigt wird.

Byte-Offset	Länge in Byte	Beschreibung
0	4	Inodenummer (0)
4	2	Länge des Verzeichniseintrags (Blockgröße)
6	1	Länge des Namens (0)
7	1	Dateityp (0)
8	2	Maximale Anzahl an Einträgen, die noch folgen können
10	2	Anzahl an Einträgen, die folgen
12	4	Nummer des Blocks innerhalb des Verzeichnisses

Tabelle 2.16: Inner Knoten in einem H-Baum eines Verzeichnisses

Die Inodenummer, Länge des Namens und der Dateityp wird in diesem Fall wieder auf 0 und die Länge des Verzeichniseintrags wird auf die Blockgröße gesetzt, damit der lineare Algorithmus diesen Block als leer interpretiert und zum nächsten Block übergeht. Anschließend folgen herkömmliche H-Baum-Kindreferenzen. Die erste Referenz ist erneut wie oben aufgebaut, da die untere Grenze des zugehörigen Hash-Wertbereichs vom Elternknoten bekannt ist.

Ein Beispiel für den Aufbau eines Verzeichnisses mit Hash-Index-Struktur ist in Listing 2.2 gegeben.

```
0x82309000  70 14 02 00 0c 00 01 02  2e 00 00 00 04 fd 01 00  |p...............|
0x82309010  f4 0f 02 02 2e 2e 00 00  00 00 00 00 01 08 00 00  |................|
0x82309020  fc 01 0c 00 01 00 00 00  fe b0 c3 1c 08 00 00 00  |................|
0x82309030  34 24 79 3d 04 00 00 00  08 0f 1a 5c 07 00 00 00  |4$y=.......\....|
0x82309040  7c 61 de 77 02 00 00 00  2e e6 d6 85 0c 00 00 00  ||a.w............|
0x82309050  44 47 b9 93 05 00 00 00  58 02 38 a4 0b 00 00 00  |DG......X.8.....|
0x82309060  2e a8 76 b7 03 00 00 00  0e b7 68 cb 09 00 00 00  |..v.......h.....|
0x82309070  f4 81 a0 df 06 00 00 00  c2 00 1c f0 0a 00 00 00  |................|
0x82309080  7f 14 02 00 10 00 08 01  61 63 6f 6e 6e 65 63 74  |........aconnect|
0x82309090  80 14 02 00 14 00 0c 01  61 63 70 69 5f 66 61 6b  |........acpi_fak|
...
```

Listing 2.2: Hexdump-Ausschnitt des Beginns eines Verzeichnisses mit Hash-Index Verzeichnis (/usr/bin/)

Wie erwartet, beginnt das Verzeichnisses mit den ersten Einträgen für '.' und '..'. Diese befinden sich in den ersten 24 Byte. Es folgen, in rot markiert, die Bauminformationen. Die ersten 4 Byte haben standardmäßig den Wert 0 und werden gefolgt von der verwendeten Hash-Version, konkret MD4. Danach folgt der Wert der Länge der Bauminformation mit 8 und die Tiefe des Baumes mit 0. Es folgt in dunkelviolett die erste Kindreferenz mit der maximal möglichen und der tatsächlichen Anzahl an Kindreferenzen anstelle der Hash-Grenze. Diese entsprechen im Beispiel `0x1fc` und `0x0c`, und damit 12 von 508 möglichen Einträgen.

In blassblau markiert sind die 11 weiteren Einträge mit jeweils 8 Byte, die diesem Header folgen. Dabei entsprechen die ersten 4 Byte dem Hash-Wert der jeweiligen Untergrenze und die letzten 4 Byte der Blocknummer.

Journal

Das Journal, so wie es in Ext3 eingeführt wurde, erhielt in Ext4 noch einige zusätzliche Funktionen, wie z.B. die Unterstützung des 64-Bit-Modus. Zusätzlich werden Journal-Transaktionen mit Prüfsummen versehen und ein asynchrones Schreiben von Commit-Blöcken ist nun möglich.

Standardmäßig ist das Journal 128 MiB groß, was der Größe einer Blockgruppe entspricht (bei einer Blockgröße von 4096 Byte). Das Maximum, welches das Journal in Ext4 einnehmen kann, beträgt 2^{32} Blöcke. Da das Journal als normale Datei im Dateisystem versteckt ist, und 2^{32} auch die maximale Größe für eine reguläre Datei ist, kann auch das Journal dieses Limit nicht übersteigen.

Grundlegend bleibt das Journal jedoch vom Aufbau identisch zu dem, das in Ext3 schon eingeführt wurde.

2.2 Dateirekonstruktion

Verschiedene Dateisysteme bieten Möglichkeiten zur Ausfallsicherung, wie etwa das Journal beim Ext4-Dateisystem, mit dem Dateien oder das gesamte Dateisystem wiederhergestellt werden können. Jedoch gibt es Fälle, in denen eine Wiederherstellung von Dateien mittels dateisystemspezifischen Mechanismen nicht möglich ist. Ebenso existieren Fälle, in denen diese Mechanismen – obwohl sie genutzt werden könnten – nicht verwendet werden dürfen. Ein solches Szenario stellt die forensische Untersuchung dar, bei der gewährleistet sein muss, dass keine Veränderung der zu untersuchenden Daten stattfindet. Zusätzlich kann der Bedarf bestehen, gelöschte Dateien zu rekonstruieren.

Prinzipiell gibt es hierfür zwei Ansätze. Einerseits können Dateisystem-Metadaten interpretiert und somit Dateien wiederhergestellt werden. Auch bei einem beschädigten Dateisystem können so Nutzdateien extrahiert werden, ohne das Dateisystem selbst zu reparieren und somit zu verändern. Andererseits existiert die Möglichkeit des File-Carvings. Dabei wird nach speziellen Merkmalen einer Datei und ihres Inhalts gesucht und diese damit wiederhergestellt. Wenn diese beiden Ansätze im forensischen Kontext eingesetzt werden, gelten für diese verschiedene Ansprüche.

Die genannten Ansätze werden von forensischen Toolkits, wie dem Sleuthkit, zur Dateiwiederherstellung durch die verschiedenen bereitgestellten Werkzeuge unterstützt. Da sich dieses unter Anderem aufgrund seines offenen Quelltexts in der digitalen Forensik gut etabliert hat, und, da die vorliegende Arbeit einen kombinierten Rekonstruktionsansatz präsentiert und diesen in das Sleuthkit-Framework integriert, werden im Folgenden jene Teilansätze, forensische Ansprüche und das Sleuthkit vorgestellt.

2.2.1 Ansätze zur Rekonstruktion

Zur Rekonstruktion von Datenträgerinhalten existieren im Wesentlichen zwei grundlegende Ansätze: Das File-Carving und die Analyse der Metadaten, die vom Datenträger zugrunde liegenden Dateisystem noch zur Verfügung stehen. Im Folgenden werden beide Ansätze erklärt und diskutiert. Insbesondere wird auf die Unterschiede, Gemeinsamkeiten und die Möglichkeit zur Kombination beider eingegangen.

File-Carving

Eine mögliche Methode zur Dateirekonstruktion ist das sogenannte File-Carving [36]. Hierbei wird auf dem zu untersuchenden Datenträger nach Merkmalen gesucht, die auf das Vorhandensein einer Datei schließen lassen, unabhängig vom verwendeten Dateisystem. Dabei werden typischerweise nach Header- und, falls anwendbar, Footer-Informationen gesucht, die den Beginn bzw. das Ende der Datei markieren. In diesen Informationen gibt es sogenannte Magic Values, die immer an der selben Stelle – meist direkt am Anfang – stehen und damit eine Datei identifizierbar machen. Dabei hängt es vom Dateityp ab, wie diese Informationen aussehen und aufgebaut sind. Bei diesem Ansatz ist es daher unerlässlich, diese Merkmale für die zu suchenden Dateitypen zu kennen, damit nach diesen gesucht werden kann. In Tabelle 2.17 werden einige gängige Dateitypen mit den entsprechenden Magic Values der Header und Footer gezeigt [3].

Header	Footer	Dateiendung	Name des Formats
`0xffd8ffe00010`	`0xffd9`	jpg	JPEG
`0x504e47`	`0xfffcfdfe`	png	Portable Network Graphics
`0x474946383961`	`0x00003b`	gif	Graphics Interchange Format
`0x25504446`	`0x25454f46`	pdf	Portable Document Format
`0x526172211a070100`	`0xc43d7b00400700`	rar	Roshal Archive

Tabelle 2.17: Magic Values für Header und Footer gängiger Dateitypen

Anhand der Magic Values, die in der ersten Spalte der Tabelle zu sehen sind, kann der Beginn einer möglichen Datei erkannt werden. Jedoch ist es nicht ausgeschlossen, dass ein anderer Festplatteninhalt diese Bytefolge beinhaltet, wodurch *false positives* entstehen, die dann von einem Carver als gültige Datei dieses Formats erkannt werden, aber in Wirklichkeit weder den angenommenen Dateityp haben, noch überhaupt Dateien sein müssen.

Zweck des File-Carvings ist es, die Rekonstruktion von Dateien auf Festplatten ohne die Nutzung von Dateisysteminformationen zu erreichen. Dadurch ist es möglich, auch in einem korrupten Dateisystem noch Dateien zu finden, da man auf dessen Struktur nicht angewiesen ist. Unter Dateisystemen der Ext-Familie würden die Metadatenstrukturen wie der Superblock und die Gruppen-Deskriptor-Tabelle nicht ausgelesen werden. Dadurch ist nichts über die Struktur des Dateisystems bekannt, was zu anderen Problemen bei der Rekonstruktion führt.

Ein solches Problem, das sich aus diesem Ansatz ergibt, besteht darin, dass Dateinamen, Verzeichnisstrukturen und andere dateisystemspezifische Metadaten nicht mitrekonstruiert werden können, was im Fall einer forensischen Untersuchung die Zuordnung der gefundenen Dateien erschwert. Ein weiteres Problem stellt die Fragmentierung von Dateien dar. Da Inhalte von Dateien nicht zwingend zusammenhängend auf der Festplatte abgelegt sein müssen, erschwert der Grad der Fragmentierung das File-Carving wesentlich. Einerseits gilt es, eine Menge von Dateisystemblöcken zu finden, die dem Gesamtinhalt der betreffenden Datei entspricht und andererseits, diese zu ordnen. Beide Schwierigkeiten sind in ihrer Grundform ohne Heuristiken oder zusätzliche Annahmen nicht in vertretbarer Zeit lösbar.

Ein weiteres Problem, das beim Carven nach Dateien beachtet werden muss, sind eingebettete Dateien. Beispielsweise bei PDF- oder Word-Dokumenten, in denen JPEGs eingefügt wurden, können diese separat als gültiges JPEG verstanden werden. Ein Carver, der false positives umgehen will, indem er bei zwei aufeinanderfolgenden Dateianfängen den ersten ignoriert (wenn für diesen ein erwarteter Footer fehlt), übersieht in diesem Beispiel das umgebende PDF- oder Word-Dokument.

Um die eben genannten Probleme des File-Carvings zu lösen, gibt es verschiedene Ansätze. Einen davon stellt die bereits in Kapitel 1 genannte Arbeit von Hand u.a. [15] dar, in welcher die Inhalte von ausführbaren Dateien auf ihren Kontrollfluss analysiert werden, um plausible Anknüpfungspunkte für Dateifragmente zu finden. Derartige Heuristiken sind zwar abhängig vom Dateityp, aber neben statistischen Techniken eine beliebte Lösung. Eine Heuristik, die eine geringere Abhängigkeit vom Dateityp aufweist, ist die Verknüpfung von Fragmenten mittels Entropieanalysen [17]. Hierbei wird davon ausgegangen, dass die Entropie sich innerhalb einer Datei nicht sprunghaft verändert, womit sich Sprünge im Entropieverlauf auf Fragmentierung zurückführen lassen.

Interpretation der Dateisystem-Metadaten

Die Interpretation von Dateisystem-Metadaten ist eine andere Möglichkeit, Dateien zu sichern. Im Gegensatz zum File-Carving werden hier die Metadaten des Dateisystems interpretiert, unabhängig von den Daten auf der Festplatte. Die Abhängigkeit liegt hierbei beim verwendeten Dateisystem und nicht beim Dateityp.

Die Interpretation von Dateisystem-Metadaten entspricht im Kern der herkömmlichen Nutzung eines Dateisystems und hat das gleiche Ziel: Die strukturierte Interpretation eines Datenträgers. Ohne ein Dateisystem ist auf einem solchen Datenträger lediglich eine Menge von Dateiinhalten, wie sie ein Carver erfassen würde, zu lesen. Hierbei fehlen Informationen über Dateigröße, -Name, entsprechende Verzeichnisstruktur, Zeitstempel und weitere, dateisysteminterne Informationen. Zum reinen Lesen des Inhalts einer zusammenhängenden Datei, deren Anfang und Ende erkannt werden können, sind diese Informationen auch nicht zwingend notwendig, jedoch enthält das Dateisystem auch Informationen über die Lage und Zusammensetzung von Dateifragmenten. Darüber hinaus können Verzeichnis- und Dateinamen für eine forensische Untersuchung nicht nur relevante Hinweise zur Beweisführung geben, sondern kann auch ein strukturiert benanntes Verzeichnissystem eine Durchsuchung vereinfachen. File-Carver können diese Namen und Verzeichniszugehörigkeiten nicht ermitteln.

Der Fall, in dem alle Dateisystem-Metadaten bekannt sind, ermöglicht die absolute Unabhängigkeit vom Wissen über Dateiinhalte und -formate. Dateianfänge, -ausdehnung und Fragmentstrukturen sind ohne die Analyse der Dateien selbst bekannt; die Metadatenanalyse kann also als eine Art Gegenstück zum File-Carving verstanden werden. Zwar stellt der Fall, in dem jede Form von Metadaten intakt ist, keine besondere Rekonstruktionsproblematik dar; schließlich liegt ein herkömmliches Dateisystem vor, das gemäß Standard interpretiert werden kann. Für die digitale Forensik ist es nun von Interesse, zu analysieren, welche Möglichkeiten ein beschädigter Metadatensatz bietet und, ob eine Kombination aus File-Carving und Metadatenanalyse einen zusätzlichen Erkenntnisgewinn mit sich bringt. Spezieller ist von besonderem Belang, mit welchem Minimalsatz an intakten Metadaten die Inhalte des Datenträgers wiederhergestellt werden können und auch, wie Dateien, die vom Dateisystem nicht angezeigt werden (weil diese z.B. gelöscht wurden) aber vorhanden sind, auffindbar gemacht werden können.

Derartige Überlegungen sind immer im Kontext mit den Grundprinzipien der analysierten Dateisysteme zu verstehen. So kann in einem Journaling-Dateisystem das Wissen um die Lage des Journals zusätzliche Ansatzpunkte für die Rekonstruktion liefern, Systeme mit *Fork*-Mechanismen können die Rekonstruktion älterer Dateiversionen ermöglichen. Dateisysteme für Flashspeicher, wie etwa YAFFS2, können ebenfalls die Rekonstruktion älterer Dateiversionen ermöglichen, da durch das sequenzielle Beschreiben der Dateisystemseiten ältere Dateiinhalte so lange wie möglich auf dem Datenträger bestehen bleiben.

Im Falle der Ext-Dateisystem-Familie ist das Zweigespann aus dem sogenannten Superblock und der Gruppen-Deskriptor-Tabelle der Einstiegspunkt für die Auswertung von Metadaten. Alle Dateisystem-Metadaten werden entweder vom Superblock oder der Gruppen-Deskriptor-Tabelle enthalten oder referenziert. Zur Steigerung der Ausfallsicherheit sind in der Regel mehrere Kopien beider Metadatenstrukturen auf dem Datenträger verteilt. Durch diese wird eine Wiederherstellung der Dateisystem-Metadaten auch bei Beschädigung von Datenträgerbereichen, die Metadaten beinhalten, ermöglicht. So wird gewährleistet, dass das beschädigte Dateisystem noch immer interpretiert werden kann.

Es ergibt sich eine Abhängigkeit zwischen den Rekonstruktionsmöglichkeiten und der Ausrichtung des Dateisystems: Je ausfallsicherer ein Dateisystem ausgelegt ist, desto wahrscheinlicher ist eine Rekonstruktion bei Beschädigung der Dateisystem-Metadaten möglich. Ihre Rekonstruktion ermöglicht die Wiederherstellung der Metadaten einer Datei (z.B. Zugriffsrechte, Owner-ID, . . .) mit ihrem Namen und Ort innerhalb der Verzeichnisstruktur.

Unabhängig vom Dateisystem gibt es eine unumstößliche Einschränkung dieses Ansatzes: Auch, wenn viele Metadaten fehlen, lässt sich unter Umständen eine beschädigte Datei zwar rekonstruieren, aber wenn die entsprechenden Referenzen vom Metadatum zur Nutzdatei fehlen oder beschädigt sind, ist es nicht möglich, den Anfang der Datei zu finden oder gar ihre Fragmentierung aufzulösen. In solch einem Fall kann allerdings das File-Carving Abhilfe schaffen.

2.2.2 Forensische Ansprüche

Da die vorgestellte Arbeit einen Ansatz zur Rekonstruktion von Dateien im forensischen Kontext behandelt, soll an dieser Stelle auf die in diesem Umfeld typischen Rahmenbedingungen eingegangen werden. Sowohl die vorgestellten Ansätze, als auch das in dieser Arbeit vorgestellte Werkzeug, müssen im Hinblick auf diese Ansprüche strukturiert sein, falls sie für forensische Untersuchungen eingesetzt werden sollen.

Aus gerichtlicher Sicht besteht der Zweck einer forensischen Untersuchung darin, Spuren zu sichern und zu untersuchen, um Beweismittel zu liefern. Die Beweiskraft von Spuren verfällt, sobald eine Verfälschung der Quelle einer Spur stattfindet. Spuren bezeichnen in diesem Kontext für eine Ermittlung relevante Gegebenheiten; diese können als Beweismittel vor Gericht dienen. Aufgrund dessen muss bei jeder forensischen Untersuchung großes Augenmerk darauf gelegt werden, die ursprünglichen Spuren nicht zu verändern.

Im Fall der digitalen Forensik folgt aus diesem Anspruch beispielsweise die Forderung, dass zur Analyse eingelesene Datenträgerabbilder nicht verändert werden dürfen. Um dies zu garantieren, muss eingehend dokumentiert werden, wie mit den verwendeten Eingabedaten umgegangen wird, beispielsweise durch Prüfsummenberechnung. Ändert sich eine Prüfsumme eines Datenträgerabbilds im Laufe der Untersuchung, kann nicht mehr gefordert werden, dass dieses nie verändert worden ist.

Auch, um gefundene Ergebnisse in ihren jeweiligen Kontext setzen zu können, muss jede gefundene Spur dokumentiert werden, damit ihr Wiederauffinden unabhängig wiederholbar ist. Andernfalls wäre das Zustandekommen der betreffenden Spur nicht nachvollziehbar und würde sich somit nicht als Teil einer logischen Beweiskette einsetzen lassen. Schließlich darf die Aussagekraft eines Beweismittels nicht an Bedingungen geknüpft sein, die sich durch Details der zur Untersuchung verwendeten Technologie ergeben. Wird beispielsweise zu einer Spur nicht gespeichert, an welcher Adresse im Datenträgerabbild sie gefunden wurde, könnte sie nicht durch manuelle Wiederholung des zugrunde liegenden Ansatzes wiedergefunden werden, da hierfür der Einstiegspunkt fehlen würde.

Weiterhin dürfen gefundene Spuren nicht bei ihrer Rekonstruktion verändert wiedergegeben werden; vielmehr müssen die genauen Umstände ihres Zustandekommens bekannt sein. Aus diesem Grund eignen sich, wie bereits erwähnt, Open-Source-Werkzeuge für forensische Untersuchungen: Im Zweifelsfall kann jeder Untersuchungsschritt überprüft werden. Zusätzlich kann durch eigene Kompilation garantiert werden, dass der für diese Argumentation betrachtete Quelltext tatsächlich der ausgeführte ist. Auch in diesem Fall gilt: Wird der zugrunde liegende Algorithmus von einem Ermittler von Hand wiederholt und eine Spur nicht wiedergefunden, ist das Zustandekommen dieser Spur zweifelhaft und sie kann nicht angenommen werden.

2.2.3 Tools zur Rekonstruktion am Beispiel Sleuthkit

Für die Rekonstruktion von Dateien existieren viele verschiedene Programme und Tools. Darunter finden sich sowohl kommerzielle als auch Open-Source-Tools. Im forensischen Bereich ist es wichtig, zu wissen, wie das verwendete Tool seine Ergebnisse produziert und ob diese damit auch aussagekräftig sind. Bei Closed-Source-Programmen muss der Dokumentation ein Vertrauensvorschuss geleistet werden, was ein Vorteil für Open-Source-Tools ist, denn bei diesen ist präzise nachzuvollziehen, wie die Ergebnisse entstehen.

Das Sleuthkit ist ein weit verbreitetes, forensisches Open-Source-Toolkit von Brian Carrier, das verschiedene forensische Werkzeuge zur Dateisystemanalyse bereit stellt. Von anderen Entwicklern werden neue Module und Programme für das Sleuthkit geschrieben und hinzugefügt, welche dieses Toolkit sehr umfangreich machen. Diese können in das gegebene Framework des Sleuthkits, oder der graphischen Oberfläche Autopsy, eingebunden werden.

Allgemein

Das Sleuthkit (TSK) [7] hat sich im Laufe der Jahre als beliebte Open-Source-Sammlung forensischer Werkzeuge, als forensisches Framework und als Programmbibliothek mit einer eigenständigen graphischen

Oberfläche (Autopsy) etabliert. Es ermöglicht die Untersuchung von Datenträgern und Dateisystemen. Seine Bibliothek liefert Programme für die Kommandozeile, mittels derer gezielt nach Dateisystem bezogenen Metadaten und Nutzdaten gesucht werden kann. Diese Programme sind bei der Untersuchung von Dateisystemen nicht auf betriebssystemspezifische Mechanismen angewiesen, wodurch auch versteckte und gelöschte Dateien gefunden und angezeigt werden können.

Sowohl verschiedene Partitions- als auch Dateisysteme werden vom Toolkit unterstützt. Um bei einer kompletten Analyse des Dateisystems einen besseren Überblick zu gewährleisten, wird mit Autopsy eine graphische Schnittstelle zu den Werkzeugen des Sleuthkits zur Verfügung gestellt. Dabei bietet Autopsy verschiedene Methoden zur Verwaltung unterschiedlicher forensischer Fälle, zur Gewährleistung der Integrität des zu untersuchenden Objekts und für automatisierte Abläufe auf dem Datenträgerabbild.

Darüber hinaus ermöglicht das TSK-Framework die Eingliederung von Modulen, die von verschiedenen Entwicklern zur Verfügung gestellt werden oder den eigenen Bedürfnissen entsprechend selbst entwickelt wurden. Auf diese Weise können Funktionalitäten, die das Sleuthkit nicht inhärent anbietet, die aber für besondere Untersuchungsmethoden im Bereich der Dateianalyse benötigt werden, eingegliedert werden.

Das Sleuthkit bietet nicht nur Werkzeuge zur Datenträger- und Dateianalyse, sondern auch zur Organisation und Visualisierung der Funde. Dabei können Details von Metadaten der Dateisysteme und Protokolle der Dateiaktivitäten angezeigt werden. Auch können Dateien nach ihrem Dateityp sortiert werden und Datei-Hashes mit einer Datenbank abgeglichen werden. Verschiedene Werkzeuge des Toolkits sind in Tabelle 2.18 zu sehen, wie sie auch im eigenen Sleuthkit-Wiki aufgelistet zu finden sind [35].

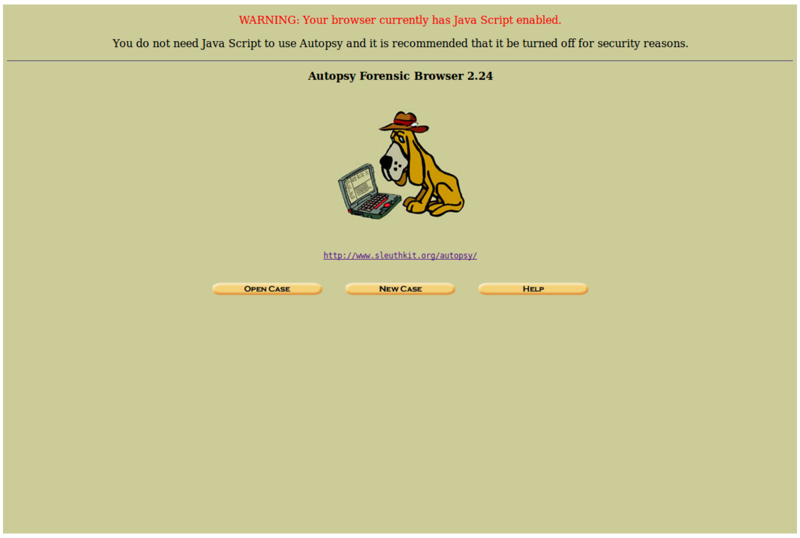
Tool-Name	Beschreibung
tsk_comparedir	Vergleicht lokale Verzeichnishierarchie mit Datenträger
tsk_gettimes	Extrahiert zeitlichen Aktivitätsverlauf eines Datenträgerabbilds
tsk_loaddb	Lädt Metadaten eines Datenträgerabbilds in eine SQLite-Datenbank
tsk_recover	Extrahiert Dateien eines Datenträgerabbilds in ein lokales Verzeichnis
fsstat	Zeigt Details und Aufbau des Dateisystems an
fls	Zeigt Namen aller Dateien innerhalb eines Verzeichnisses
icat	Extrahiert Datenblöcke einer Datei, anhand ihrer Metadaten
istat	Zeigt die Metadaten einer Datei an
blkcat	Extrahiert den Inhalt eines gegebenen Datenblocks
blkstat	Zeigt Details eines gegebenen Datenblocks
mmls	Zeigt den Aufbau eines Datenträgers, mit nicht allozierten Bereichen
mmcat	Extrahiert den Inhalt eines Laufwerks

Tabelle 2.18: Auswahl verschiedener Tools des Sleuthkits

Die ersten genannten Tools sind voll automatisiert und verbinden Datenträger- und Dateisystemanalysen. Als Eingabe erwarten sie ein Datenträgerabbild, erkennen auf ihm Laufwerke und interpretieren eigenständig deren Inhalte. Anstatt manuell die einzelnen Tools des Sleuthkits aufrufen zu müssen, können damit Funktionalitäten mehrerer Tools verkettet aufgerufen werden.

Die restlichen aufgelisteten Tools sind Kommandozeilenbefehle, die einzeln auf ein Datenträgerabbild angewendet werden können. Alle Befehle, die mit einem ’f’ beginnen, beziehen sich dabei auf das Dateisystem und seine Metadaten. Befehle die mit einem ’i’ beginnen, beziehen sich auf Dateien und ihre Metadaten. Beginnt ein Befehl mit ’blk’, so bezieht sich dieser auf einen Datenblock innerhalb des Dateisystems. Befehle, die ein Laufwerk auf einem Datenträger betreffen, beginnen mit ’mm’.

Der Anwendungsbereich des Sleuthkits liegt, wie von Casey [8] und Nelson u.a. [26] beschrieben wurde, im forensischen Bereich. Der Hauptvorteil gegenüber kommerziellen Lösungen liegt in der Nachvollziehbarkeit der Ergebnisse: Da das Sleuthkit eine Open-Source-Lösung ist, bietet sie jedem Ermittler die Möglichkeit, exakt nachzuvollziehen, wie alle gefundenen Ergebnisse zustande gekommen sind. In Arbeiten wie der von Hofherr [16] wird ebenso beschrieben, dass das Sleuthkit durchaus konkurrenzfähige Ergebnisse liefert und sich durch seine Erweiterbarkeit und Nachvollziehbarkeit gegenüber der kommerziellen Konkurrenz hervorhebt.



WARNING: Your browser currently has Java Script enabled.

You do not need Java Script to use Autopsy and it is recommended that it be turned off for security reasons.

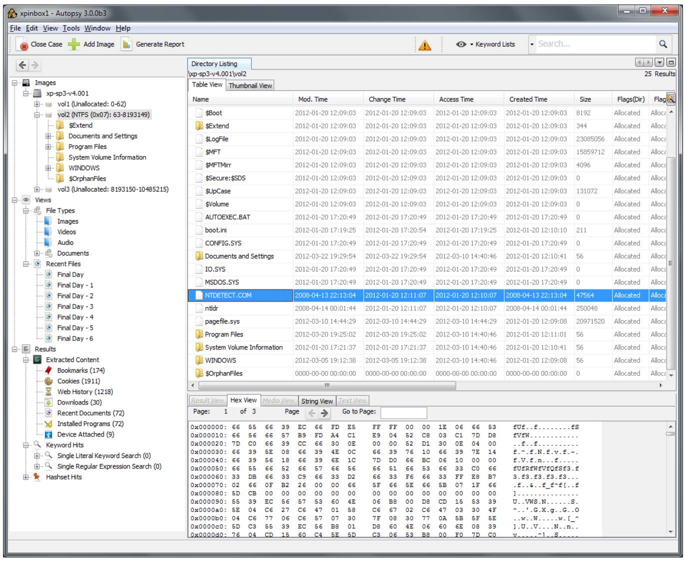
Autopsy Forensic Browser 2.24

http://www.sleuthkit.org/autopsy/

Abbildung 2.6: Screenshot eines Autopsy-Startbildschirms von Version 2 unter Ubuntu Linux

Es gilt, zu beachten, dass zwei Versionen von Autopsy zur Verfügung stehen: Version 2 und Version 3.1. Hierbei stellt Version 2 eine auf UNIX lauffähige, browserbasierte Nutzeroberfläche dar, wie sie in Abbildung 2.6 zu sehen ist. Diese wird nicht mehr aktiv weiter entwickelt. Für die Funktionsweise wird das Sleuthkit separat installiert benötigt, da Autopsy lediglich eine Benutzer-Schnittstelle darstellt.

Version 3.1 von Autopsy hingegen ist eine in Java und Python implementierte Oberfläche, die momentan ausschließlich unter Windows-Betriebssystemen lauffähig ist. In Abbildung 2.7 ist ein beispielhafter Screenshot von Autopsy 3.1 zu sehen. In dieser Version bietet Autopsy die Möglichkeit für Entwickler, weitere Module zu schreiben und diese in die Oberfläche einzugliedern. Hierfür wird ein eigenes Framework angeboten.



\xp-sp3-v4.001\vol2

25 Results

Name	Mod. Time	Change Time	Access Time	Created Time	Size	Flags(Dir)	Flag
$Boot	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	8192	Allocated	Alloca
$Extend	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	344	Allocated	Alloca
$LogFile	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	23085056	Allocated	Alloca
$MFT	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	15859712	Allocated	Alloca
$MFTMirr	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	4096	Allocated	Alloca
$Secure:$SDS	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	0	Allocated	Alloca
$UpCase	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	131072	Allocated	Alloca
$Volume	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	2012-01-20 12:09:03	0	Allocated	Alloca
AUTOEXEC.BAT	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	0	Allocated	Alloca
boot.ini	2012-01-20 17:19:25	2012-01-20 17:20:54	2012-01-20 17:19:25	2012-01-20 12:10:10	211	Allocated	Alloca
CONFIG.SYS	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	0	Allocated	Alloca
Documents and Settings	2012-03-22 19:29:54	2012-03-22 19:29:54	2012-03-10 14:40:46	2012-01-20 12:10:41	56	Allocated	Alloca
IO.SYS	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	0	Allocated	Alloca
MSDOS.SYS	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	2012-01-20 17:20:49	0	Allocated	Alloca
NTDETECT.COM	2008-04-13 22:13:04	2012-01-20 12:11:07	2012-01-20 12:10:07	2008-04-13 22:13:04	47564	Allocated	Alloca
ntldr	2008-04-14 00:01:44	2012-01-20 12:11:07	2012-01-20 12:10:07	2008-04-14 00:01:44	250048	Allocated	Alloca
pagefile.sys	2012-03-10 14:44:29	2012-03-10 14:44:29	2012-03-10 14:44:29	2012-01-20 12:09:08	20971520	Allocated	Alloca
Program Files	2012-03-20 19:25:02	2012-03-20 19:25:02	2012-03-10 14:40:46	2012-01-20 12:11:01	56	Allocated	Alloca
System Volume Information	2012-01-20 17:21:37	2012-01-20 17:21:37	2012-03-10 14:40:46	2012-01-20 12:10:41	56	Allocated	Alloca
WINDOWS	2012-03-05 19:12:38	2012-03-05 19:12:38	2012-03-10 14:40:46	2012-01-20 12:09:08	56	Allocated	Alloca
$OrphanFiles	0000-00-00 00:00:00	0000-00-00 00:00:00	0000-00-00 00:00:00	0000-00-00 00:00:00	0	Allocated	Alloca

Page: 1 of 3 Page Go to Page:

```
0x000000: 66  55  66  39  EC  66  FD  E5    FF  FF  00  00  1E  06  66  53    fUf..f........fS
0x000010: 66  56  66  57  B9  FD  A4  C1    E9  04  52  C8  03  C1  7D  D8    fVfW............
0x000020: 7D  C0  66  39  CC  66  30  0E    00  00  52  D1  30  0E  04  00    ..f..f..........
0x000030: 66  39  5E  08  66  39  4E  0C    66  39  76  10  66  39  7E  14    f.^.f.N.f.v.f.~.
0x000040: 66  39  56  18  66  39  6E  1C    7D  D0  66  BC  06  10  00  00    f.V.f.n...f.....
0x000050: 66  55  66  52  66  57  66  56    66  51  66  53  66  33  C0  66    fUfRfWfVfQfSf3.f
0x000060: 33  DB  66  33  C9  66  33  D2    66  33  F6  66  33  FF  E8  B7    3.f3.f3.f3.f3...
0x000070: 02  66  0F  B2  26  00  00  66    5F  66  5E  66  5B  07  1F  66    .f..&..f_f^f[..f
0x000080: 5D  CB  00  00  00  00  00  00    00  00  00  00  00  00  00  00    ]...............
0x000090: 55  39  EC  56  57  53  60  4E    06  B8  00  D8  CD  15  53  39    U..VWS.N......S.
0x0000a0: 5E  04  C6  27  C6  47  01  58    C6  67  02  C6  47  03  30  4F    ^..'.G.X.g..G..O
0x0000b0: 04  C6  77  06  C6  57  07  30    7F  08  30  77  0A  5B  5F  5E    ..w..W.....w.[_^
0x0000c0: 5D  C3  55  39  EC  56  B8  01    D8  60  4E  06  60  6E  08  39    ].U..V....N..n..
0x0000d0: 76  04  CD  15  60  C4  5E  5D    C3  06  53  B8  00  F0  7D  C0    v.....^]..S.....
```

Abbildung 2.7: Benutzeroberfläche von Autopsy 3.1 (http://www.sleuthkit.org/autopsy/images/v3/overview.png)

Dieses Framework ist abzugrenzen vom Sleuthkit-Framework: Ersteres ermöglicht Entwicklern die Erweiterung der Nutzeroberfläche um weitere Funktionalitäten innerhalb von Autopsy und damit in Java oder Python. Jedoch bildet die Grundfunktionalität das Sleuthkit.

Sleuthkit-Framework

Das Sleuthkit-Framework [6] ermöglicht die Einbindung eigens implementierter Funktionalitäten in Form von Modulen. Es war ursprünglich dafür gedacht, in einer verteilten Umgebung eingesetzt zu werden, wie etwa einem Cluster, um damit Parallelisierung zu erreichen. Allerdings können damit auch Desktop-Anwendungen geschrieben werden, sofern die verfügbaren Ressourcen dies erlauben.

Der Analyseprozess im Framework besteht aus drei Phasen, die über eine zentrale Datenbank kommunizieren, wie es in Abbildung 2.8 zu sehen ist.

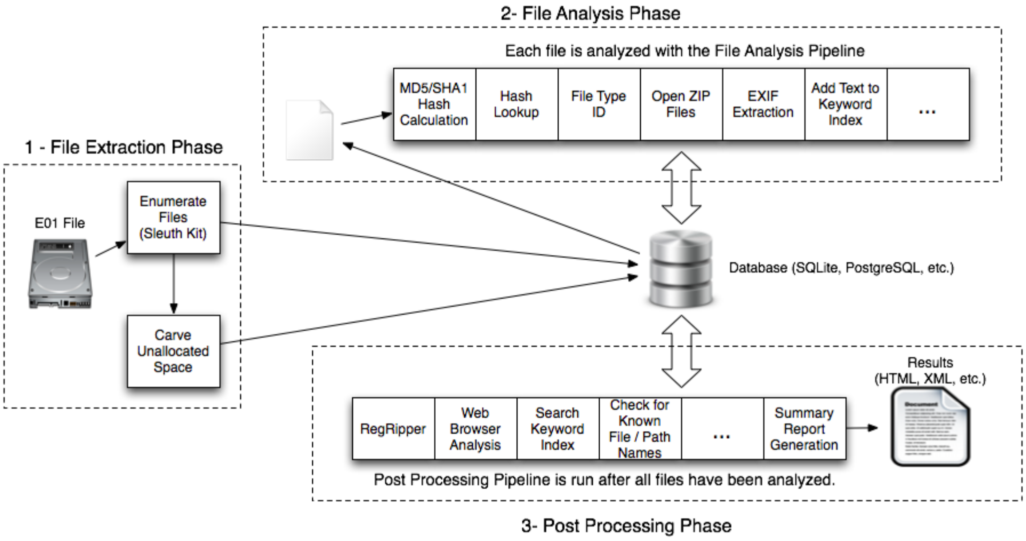


Abbildung 2.8: Aufbau der Modul-Pipeline des Sleuthkit-Frameworks (http://www.sleuthkit.org/sleuthkit/docs/framework-docs/basics_page.html)

Die drei Phasen decken verschiedene Aufgabenbereiche im Verlauf einer forensischen Analyse ab. Die erste Phase besteht aus der Extraktion von Daten aus dem zu untersuchenden Datenträger. Dafür gibt es verschiedene Möglichkeiten, darunter die bereits diskutierten Ansätze für das File-Carving und die Metadaten-Analyse. Diese Beginnphase beinhaltet das Befüllen der Datenbank. An dieser Stelle ist es nicht möglich, ein eigenes Modul in die Pipeline einzuhängen.

In der zweiten Phase, der Dateianalyse, wird auf den Dateien aus der Datenbank gearbeitet. Dabei können dateitypenspezifische Module aufgerufen werden, wie etwas das Öffnen von komprimierten Dateien, das Auslesen von Exif-Daten von Bilddateien, und auch Module, die unabhängig vom Dateityp arbeiten und für alle Dateien aufgerufen werden. Die Ergebnisse aus dieser Phase werden wieder zurück in die Datenbank geschrieben. In dieser Phase können eigens geschriebene Module eingehängt werden, die Aufgaben der Dateianalyse bewältigen.

Die letzte Phase ist die Nachbearbeitung. In diesem Abschnitt werden die Ergebnisse aus der Dateianalyse sortiert und für einen abschließenden Bericht zusammengefasst. Dabei kann nach bestimmten Dateien und Inhalten im Rahmen der forensischen Analyse gesucht werden. Zusätzlich kann eingeschränkt werden, aus welchen Quellen im Dateisystem Dateien wiederhergestellt werden sollen. Somit kann im Bericht mit den Ergebnissen gezielt die Art von Inhalt herausgefiltert werden, die im Rahmen der Untersuchung als relevant angesehen wird. Auch in dieser Phase ist es möglich, eigene Module in die Pipeline einzubringen, allerdings steht am Schluss dieser Phase immer das Modul, das den Abschlussbericht verfasst.

So kann, je nach Bedarf, das Framework für die eigene Verwendung angepasst werden. Dies geschieht mit einer Konfigurationsdatei in XML, die ausschnittsweise in Abbildung 2.9 zu sehen ist.

Aus der zweiten und dritten Phase können einige Module manuell aus der Pipeline entfernt und auch umsortiert werden. Aus dieser Abbildung ist ebenso der Aufbau der Pipeline ersichtlich: Modulen sind im

```
<PIPELINE_CONFIG>
  <PIPELINE type="FileAnalysis">
    <MODULE order="1" type="plugin" location="tskHashCalcModule"/>
    <MODULE order="2" type="plugin" location="tskHashLookupModule" arguments="..."/>
    <MODULE order="3" type="plugin" location="tskFileTypeSigModule"/>
    <MODULE order="4" type="plugin" location="tskEntropyModule"/>
    <MODULE order="5" type="plugin" location="tskZipExtractionModule"/>
    <MODULE order="6" type="plugin" location="tskLibExifModule"/>
  </PIPELINE>
  <PIPELINE type="PostProcessing">
    <MODULE order="1" type="plugin" location="tskRegRipperModule" arguments="..."/>
    <MODULE order="2" type="plugin" location="tskInterestingFilesModule"/>
    <MODULE order="3" type="plugin" location="tskSaveInterestingFilesModule"/>
    <MODULE order="4" type="plugin" location="tskSummaryReportModule"/>
  </PIPELINE>
</PIPELINE_CONFIG>
```

Abbildung 2.9: Konfigurationsdatei für die Pipeline des Sleuthkit-Frameworks

`order`-Attribut Positionen innerhalb der Pipeline zugewiesen und ihre Zuordnung zu der jeweiligen Phase geschieht durch die zwei unterschiedlichen Pipeline-Tags. Das `type`-Attribut beschreibt, ob ein Modul als Plugin innerhalb des Frameworks oder als ausführbare Datei zur Verfügung steht. Ein Plugin ist ausschließlich dazu gedacht, innerhalb des Frameworks zu agieren. Eine ausführbare Datei kann ebenso von der Kommandozeile aufgerufen werden. Hierbei haben ausführbare Dateien niemals Zugriff auf die Kommunikationsdienste des Frameworks, wie z.B. die Datenbank. Zusätzlich beschreiben die Attribute `location`, `arguments` und `output` die Parameter für die Ausführung des jeweiligen Moduls.

3

IMPLEMENTIERUNG

Aufbauend auf dem vorherigen Kapitel wird nun ein forensisches Werkzeug im Detail im Bezug auf seine Funktionsweise, Struktur und Benutzung erläutert. Dieses Werkzeug wurde als C++-Modul für das Sleuthkit-Framework entwickelt und dient zur Rekonstruktion von Dateien auf Ext4-Dateisystemen. Mittels einer File-Carving-Methode, die auf Suchmustern basiert, wird nach Metadatenstrukturen von Inodes gesucht, um deren Inhaltsdaten im weiteren Verlauf wiederherzustellen. Dafür werden verschiedene Suchmuster in dieser Arbeit vorgestellt, und zu einem späteren Zeitpunkt evaluiert.

Der vorgestellte Ansatz soll das Auslesen des Superblocks und der Gruppen-Deskriptor-Tabelle vermeiden. Dadurch soll gewährleistet werden, dass Dateien eines korrupten oder überformatierten Ext4-Dateisystems wiederhergestellt werden können. Hierfür wird ein Minimalsatz an Informationen über das Dateisystem benötigt, aus dem sich das Modul die essentiellen Parameter selbst errechnen kann. Falls Standardwerte benutzt werden, so wie sie das mkfs-Programm bei der Formatierung eines Dateisystems setzt, sind von außen keine weiteren Informationen notwendig.

Für die Wiederherstellung der Inhaltsdaten der Dateien werden besagte Inodes auf ihre Metadatenstruktur untersucht und entsprechend nach Dateityp unterschiedlich behandelt. Dabei beschränkt sich die Rekonstruktion auf nicht gelöschte reguläre Dateien und Verzeichnisse. Mittels Verzeichnissen können Dateinamen, Inodenummern und vollständige Dateipfade für reguläre Dateien erschlossen werden. Die Notwendigkeit dessen ergibt sich daraus, dass sich diese Informationen nicht in Inodes befinden. Allerdings liefern Informationen wie Dateinamen wichtige Anhaltspunkte im forensischen Kontext.

Um das Modul in verschiedenen Anwendungsfällen einsetzen zu können, bietet es zwei verschiedene Modi für die Rekonstruktion der Dateien an. Einen davon stellt der *Inhaltsdaten-Modus* dar, in dem ausschließlich Dateiinhalte regulärer Dateien rekonstruiert werden. Für diesen ist nur die Blockgröße des Ext4-Dateisystems notwendig.

Der andere Modus ist der *Metadaten-Modus*, in dem weitere Ext4-Parameter notwendig sind, da die Inodenummern der gefundenen Inodes berechnet werden müssen. Dadurch ist es jedoch möglich, den Dateinamen und -pfad einer regulären Datei mittels Verzeichnissen zu rekonstruieren. In diesem Modus bietet sich ebenso die Möglichkeit die Dateinamen und -pfade anderer Dateitypen zu rekonstruieren, die anhand der Verzeichniseinträge gefunden wurden, deren Inhalt allerdings nicht rekonstruiert werden kann.

In diesem Kapitel wird eingangs beschrieben, wie sich das typische Anwendungsschema des Programms zusammensetzt. Die Teilschritte des Moduls werden einzeln aufgegliedert und im Detail erläutert. Dabei wird auf die Unterschiede bei der Ausführung des Moduls der beiden Modi eingegangen. Anschließend wird die Implementierung der einzelnen Teilschritte detailliert erklärt. Insbesondere werden algorithmisch relevante Stellen innerhalb des entwickelten Codes dargelegt und erläutert.

Schließlich wird die Benutzung und die Einbindung des Moduls in das Sleuthkit-Framework beleuchtet. Es wird unter Anderem auf die Konfigurationsdatei eingegangen, die verschiedene Optionen und Parameter für den Nutzer zur Verfügung stellt. Hierunter fällt das Überschreiben der mkfs-Standardwerte und das Eingrenzen des Datenträgerabbilds auf eine bestimmte Partition, auf der die Untersuchung stattfinden soll.

3.1 Funktionsweise

Das im Folgenden vorgestellte forensische Werkzeug dient der Wiederherstellung von Dateien auf Datenträgerabbildern, insbesondere auf beschädigten oder überformatierten Ext4-Dateisystemen. Dies geschieht über einen kombinierten Mechanismus, der zunächst File-Carving-Methoden nach Inode-Metadaten-Strukturen auf einem Datenträgerabbild anwendet, um die gefundenen Inodes anschließend durch Interpretation ihrer Metadaten zu rekonstruieren. Dieser Ansatz verzichtet dabei vollständig auf das Auslesen des Superblocks und der Gruppen-Deskriptor-Tabelle.

Das entwickelte Modul durchläuft bei der Rekonstruktion von Dateien folgende Phasen:

1. Initialisierung
2. Inode-Carving-Phase
3. Verzeichnisbaum-Phase (optional)
4. Inhaltsdaten-Phase
5. Leerdaten-Phase (optional)

Bei der Initialisierung werden alle nötigen Parameter zur Rekonstruktion entweder durch eine Konfigurationsdatei gesetzt oder anhand der selbst ermittelten Größe des Datenträgerabbilds mit Hinblick auf die in Abschnitt 3.1.1 beschriebenen mkfs-Standardwerte bestimmt. Da das mkfs-Programm bestimmte Standardwerte für alle Datenträger benutzt und die restlichen Parameter allein von der Größe des Dateisystems abhängen, können diese selbstständig vom Modul festgelegt werden.

Die anschließende Inode-Carving-Phase findet mittels verschiedener Suchmustern Inode-Metadatenstrukturen im zu untersuchenden Ext4-Dateisystem. Dabei werden die gefundenen, potentiellen Inodes direkt nach Dateityp sortiert. Reguläre Dateien und Verzeichnisse, die von der momentanen Version für die Rekonstruktion unterstützt werden, werden im späteren Verlauf unterschiedlich behandelt, weswegen diese einer Unterscheidung bedürfen. Das Ergebnis dieser Phase sind Mengen von Byteadressen auf dem Dateisystem, die potentielle Anfänge von Inodes darstellen.

Mit den ermittelten Inodes von Verzeichnissen wird in der Verzeichnisbaum-Phase eine innere Verzeichnisbaum-Struktur im Programm aufgebaut, mit der im späteren Verlauf die Verzeichnisstruktur des Dateisystems rekonstruiert werden kann. Dafür werden die Verzeichniseinträge aus den Inhaltsdatenblöcken der Verzeichnisse gelesen und interpretiert.

In der darauf folgenden Inhaltsdaten-Phase werden reguläre Dateien mit ihren Inhaltsdaten rekonstruiert. Hierbei wird, je nachdem ob die optionale Verzeichnisbaum-Phase durchlaufen wurde, die Datei mit ihrer physikalischen Adresse als Name oder mit ihrem Dateipfad im Zielordner angelegt.

Die abschließende optionale Leerdaten-Phase erzeugt in Verzeichniseinträgen gefundene Dateinamen und -pfade als leere Dateien, wenn die dazugehörigen Dateien von den vorherigen Phasen nicht rekonstruiert wurden. Diese Phase bedingt das frühere Durchlaufen der Verzeichnisbaum-Phase.

Die Verzeichnisbaum- und Leerdaten-Phase werden zusammen in dem sogenannten Metadaten-Modus ausgeführt. In diesem werden ausschließlich Dateien wiederhergestellt, deren Inodes in Inode-Tabellen gefunden wurden, da nur diesen eine Inodenummer zugeordnet werden kann. Da diese Unterscheidung nicht ohne entsprechende Metadaten über das Dateisystem, wie etwa die verwendete Block- oder Inodegröße, getroffen werden kann, müssen diese bereitgestellt oder von Standardwerten ausgegangen werden.

Falls keine plausible Annahme über das Dateisystem möglich ist, stellt der sogenannte Inhaltsdaten-Modus eine Option dar, mit dem allein anhand der korrekten Ext4-Blockgröße Dateiinhalte rekonstruiert werden können. Dabei wird die Datei mit ihrer physikalischen Adresse des Inodes als Name wiederhergestellt, da der Dateiname nicht ermittelt werden kann.

3.1.1 Initialisierung

Das Ziel der Initialisierung besteht darin, alle benötigten Ext4-Parameter korrekt zu setzen. Hier werden sowohl manuelle Nutzereinstellungen aus einer Konfigurationsdatei als auch mkfs-Standardwerte als Eingabe angenommen. Im Standardfall, in dem das untersuchte Datenträgerabbild ausschließlich ein mit Standardwerten konfiguriertes Ext4-Dateisystem beinhaltet, muss der Nutzer keine weiteren Angaben in der Konfigurationsdatei angeben. Alle weiteren Parameter, lassen sich in diesem Fall erschließen.

Folgende Parameter sind für das Werkzeug von Relevanz:

- Offset
- Dateisystemgröße
- Blockgröße
- Inodegröße
- Inodeverhältnis
- Flex-Gruppengröße
- Adressbreite
- Sparse-Superblock
- Anzahl der Blöcke pro Blockgruppe
- Anzahl der Blöcke und Blockgruppen des Dateisystem
- Anzahl der Inodes pro Blockgruppe
- Größe des Speicherbereichs für die wachsende Gruppen-Deskriptor-Tabelle

Zu Beginn der Initialisierung wird die Größe des übergebenen Datenträgerabbilds bestimmt, auf dem sich die zu untersuchenden Ext4-Partitionen befinden. Falls sich auf dem Datenträgerabbild neben einer einzelnen Ext4-Partition noch weitere Daten befinden, können mittels der Konfigurationsdatei ein Offset und die Dateisystemgröße in Byte angegeben werden, um den zu untersuchenden Abschnitt einzugrenzen. Dies ist notwendig, da nur eine Partition in einem Durchlauf untersucht werden kann. Zusätzlich müssen die weiteren Parameter, die nicht durch die Konfigurationsdatei spezifiziert werden, für diese Partition korrekt bestimmt werden können.

Insbesondere die Dateisystemgröße stellt einen zentralen Wert dar, von dem eine Vielzahl von Parametern direkt abhängt. Einige davon sind direkt aus der Dateisystemgröße ableitbar, wie etwa die Anzahl der Blockgruppen des Dateisystems. Hierfür muss jedoch die Ext4-Blockgröße bekannt sein. Diese hängt, wie auch die Inodegröße und das Inodeverhältnis, bei der Formatierung mittels des mkfs-Programms funktional von der Dateisystemgröße ab, wie in Tabelle 3.1 skizziert wird.

Weiterhin gibt es relevante Werte, für die mkfs-Standardwerte definiert sind, die jedoch nicht von der Dateisystemgröße abhängen. Dazu gehören:

- Anzahl der Blockgruppen einer Flex-Gruppe (Standard: 16)
- Breite der Blockadressen (Standard: 32 Bit)
- Verwendung der Sparse-Superblock-Option (Standard: aktiviert)

mkfs-Typ *(mit Dateisystemgröße)*	Parameter	Wert
floppy		
(bis 3 MiB)	Blockgröße	1024 Byte
	Inodegröße	128 Byte
	Inodeverhältnis	8192
small		
(bis 512 MiB)	Blockgröße	1024 Byte
	Inodegröße	128 Byte
	Inodeverhältnis	4096
default		
(bis 4 TiB)	Blockgröße	4096 Byte
	Inodegröße	256 Byte
	Inodeverhältnis	16384
big		
(bis 16 TiB)	Blockgröße	4096 Byte
	Inodegröße	256 Byte
	Inodeverhältnis	32768
huge		
(ab 16 TiB)	Blockgröße	4096 Byte
	Inodegröße	256 Byte
	Inodeverhältnis	65536

Tabelle 3.1: Von der Dateisystemgröße abhängige Standardparameter des mkfs-Programms für ein Ext4-Dateisystem

Bei einer Formatierung mittels des mkfs-Programms gibt es Einschränkungen für den Wertebereich der genannten Parameter. Diese müssen stets eingehalten werden, selbst wenn manuelle Werte angegeben werden. Ein Beispiel hierfür ist die Blockgröße, die ausschließlich 1024 Byte, 2048 Byte und 4096 Byte annehmen kann, sofern das System keine Speicherseitengröße von 8192 Byte unterstützt. Ebenso muss der Speicherbereich, den die Inodes pro Blockgruppe einnehmen, ganzzahlig durch die Größe eines Blocks teilbar sein, so dass kein Verschnitt entsteht. Die Anzahl der Blockgruppen pro Flex-Gruppe muss eine Zweierpotenz sein und die Breite der Blockadressen kann ausschließlich die Werte 32 Bit oder 64 Bit annehmen. Auch das Inodeverhältnis ist eine Zweierpotenz und zusätzlich ein ganzzahliges Vielfaches der angegebenen Blockgröße. Vom Nutzer anders eingestellte Werte können trotz dieser Einschränkungen nicht geschätzt werden; durch die Konfigurationsdatei ist es allerdings möglich, diese zu übergeben.

Neben dem bislang beschriebenen Parametersatz existieren weitere Werte, die sich vollständig aus diesem ableiten lassen:

- Anzahl der Blöcke pro Blockgruppe
- Anzahl der Blöcke und der Blockgruppen des Dateisystems
- Anzahl der Inodes pro Blockgruppe
- Größe des Speicherbereichs für die wachsende Gruppen-Deskriptor-Tabelle

Sind alle Parameter bekannt, kann für jede Blockgruppe die Position und Größe der jeweiligen Inode-Tabelle errechnet werden. Weiterhin kann damit eine physikalische Adresse, die sich innerhalb einer solchen Inode-Tabelle befindet, auf ihre Inodenummer abgebildet werden.

Durch die Anzahl der Blockgruppen des Dateisystems und der Information, ob der 64-Bit-Modus verwendet wird, erschließt sich die Größe der Gruppen-Deskriptor-Tabelle in einer Blockgruppe. Da diese, zusammen mit dem Superblock, am Anfang jeder Blockgruppe oder denen, die durch die Sparse-Option betroffen sind, steht, muss deren Beitrag zum Offset zur Inode-Tabelle bekannt sein. Zusätzlich muss die Größe des Speicherbereichs für die wachsende Gruppen-Deskriptor-Tabelle berechnet werden, da diese von der Größe des Dateisystems abhängt. Ist die Nutzung von Flex-Gruppen aktiviert, so vergrößern sich die Speicherbereiche der Inode- und Block-Bitmaps und die der Inode-Tabelle. Weiterhin sind diese dann nur in

Blockgruppen vorzufinden, deren Blockgruppennummer ganzzahlig durch die Flex-Gruppengröße teilbar ist.

Neben den Ext4-Parametern gibt es zusätzlich die Möglichkeit, in der Konfigurationsdatei Zeitstempel oder Zugriffsrechte anzugeben. Diese werden in der Inode-Carving-Phase als Suchmuster genutzt. Durch die Zeitstempel wird ein Zeitfenster aufgespannt, in denen sich die letzten Änderungen eines Inodes befinden müssen. Für den Fall, dass keine Einschränkung auf ein Zeitfenster gelten soll, allerdings die Zeitstempel auf ihre Konsistenz überprüft werden sollen, so kann die untere Schranke für diese Zeitstempel auf 0 gesetzt werden.

3.1.2 Inode-Carving-Phase

Sobald alle notwendigen Dateisystemparameter ermittelt worden sind, beginnt die Suche nach potentiellen Inodes auf dem Dateisystem. Aufgrund der inneren Struktur eines Inodes kann nicht jede beliebige Permutation von 128 Byte als Inode interpretiert werden. Damit ein Inode korrekt und plausibel sein kann, müssen bestimmte Zusammenhänge zwischen den Werten in seiner Struktur gelten. Mittels Mustervergleichen kann sichergestellt werden, dass diese Zusammenhänge gelten. Tun sie dies, wird im Folgenden von einem potentiellen Inode gesprochen.

Zu diesem Zweck wird das zu untersuchende Datenträgerabbild byteweise durchsucht und mit einigen Suchmustern abgeglichen. Dabei spannt sich der Suchraum über $D - i + 1$ Byte auf, wobei D für die Anzahl der Bytes im Dateisystem und i für die geschätzte Größe eines Inodes steht. Dabei können nicht alle Adressen des Dateisystems als Beginn eines Inodes interpretiert werden, da sonst über die Grenzen des Dateisystems hinaus gelesen werden würde. Durch die byteweise Suche wird sichergestellt, dass trotz eines falsch gesetzten Offsets zum Dateisystem, Inodes gefunden werden können. Die zugehörigen Dateien dieser Inodes können in diesem Fall allerdings nicht wiederhergestellt werden, da durch den falschen Offset die Blockadressen der zugehörigen Inhaltsdaten falsch interpretiert werden würden. Da dem Nutzer in einem solchen Fall die Adressen der Inodes bekannt werden, kann der Offset unter Umständen manuell ermittelt und dem Modul übergeben werden.

Mittels verschiedener Mustervergleiche wird der Suchraum der gefundenen Inodes eingegrenzt. Da die essentiellen Attribute von Inodes konstante Adressen innerhalb der Datenstruktur besitzen, kann bei einer konkreten Inodeadresse auf die einzelnen Attribute zugegriffen werden. Auf diese werden Suchmuster angewendet.

Ist beispielsweise bekannt, dass nur Dateien mit bestimmten Zugriffsrechten für eine Untersuchung von Relevanz sind, kann eine Menge von Zugriffsrechts-Konfigurationen angegeben werden. In diesem Fall werden nur Dateien angenommen, deren Zugriffsrechte einer der gültigen Konfigurationen entsprechen. Die Zugriffsrechte befinden sich in den niederwertigen 9 Bit der ersten 2 Byte des Inodes; alle Permutationen dieser Bits sind syntaktisch gültig.

Die obersten 4 Bit der ersten 2 Byte beschreiben den Dateityp des Inodes. Da sich die Suche auf reguläre Dateien und auf Verzeichnisse beschränkt, müssen potentielle Inodes, deren angegebener Datentyp dem nicht entspricht, ignoriert werden.

Inodes halten sich ab dem 8. Byte Zeitstempel für verschiedene Operationen vor. Auf diese Zeitstempel können ebenfalls Mustervergleiche durchgeführt werden, wobei der atime-Wert ignoriert wird. Abhängig vom Betriebssystem, kann die regelmäßige Aktualisierung der atime aus Performancegründen deaktiviert worden sein, wodurch dieser Zeitstempel nicht interpretiert werden darf. Für die restlichen Zeitstempel kann ein Zeitfenster in der Konfigurationsdatei angegeben werden. Dateien mit Zeitstempeln außerhalb dieses Fensters werden ignoriert. Für die Erhebung der Zeitwerte werden jeweils nur die unteren 32 Bit in Betracht gezogen, wodurch nur sekundengenaue Angaben bis ins Jahr 2106 angenommen werden können. Zusätzlich werden diese durch eine Konsistenzüberprüfung verifiziert.

Da sich die Suche auf nicht-gelöschte Dateien beschränkt, müssen gültige Extent-Einträge in den Inodes existieren. Bei gelöschten Dateien sind die Extent-Einträge nicht befüllt, wodurch keine Verweise auf rekonstruierbare Inhaltsdaten zur Verfügung stehen, weswegen diese Dateien für eine Rekonstruktion nicht in Frage kommen. Dabei bietet es sich an, im für Flags reservierten Feld eines Inodes das Extent-Flag auszuwerten. Ist dieses nicht gesetzt, kann kein relevanter Inode vorliegen, da andere Strukturen vom vorgestellten Werkzeug nicht unterstützt werden, so wie Inline-Dateien und -Verzeichnisse oder von Ext3 konvertierte Dateien mit indirekten Blockzeigern. Unter dieser Annahme kann zusätzlich im Feld mit den Extent-Datentstrukturen nach der Magic Number `0xf30a` im entsprechenden Header gesucht werden.

Zusätzlich zeichnen sich gelöschte Dateien dadurch aus, dass die Anzahl an Hard Links 0 beträgt. Somit kann als zusätzliches Suchmuster die Anzahl an Hard Links auf einen Inode in Betracht gezogen werden, da alle relevanten Inodes hier einen Wert ungleich 0 aufweisen müssen.

Weitere Suchmuster wären das Feld mit Dateigröße und das Feld mit der Anzahl der reservierten Blöcke des Inodes. Für sich genommen bieten diese beiden Attribute keine Ausschlusskriterien, weil für diese keine ungültigen Werte existieren. Da allerdings der Speicherbereich, der durch die reservierten Blöcke abgedeckt wird, größer sein muss als die angegebene Dateigröße, würde sich so eine Konsistenzüberprüfung ergeben. Jedoch werden beide Werte unter Ext4 auf jeweils zwei Felder aufgeteilt, da größere Dateien unterstützt werden müssen als unter Ext2/3. Genauer gesagt erweitert sich die Inodespezifikation unter Ext4 hier um zwei Felder, die für jeweils einen der genannten Werte zusätzliche, höherwertige Bits enthalten. Diese befinden sich in den unter Ext2/3 unbenutzten Speicherbereichen des Inodes.

Die höherwertigen Bits der Anzahl der Blöcke liegen in Standard-Linux-Distributionen zwar an einer konstanten Adresse, jedoch wird dieser Speicherbereich unter den hurd2- und masix2-Serversystemen anderweitig verwendet. Um Betriebssystem-Unabhängigkeit zu wahren, können diese 16 Bit im Allgemeinen also nicht interpretiert werden. Diese höherwertigen Bits zu ignorieren, bedeutet, dass die gelesene Anzahl reservierter Blöcke immer größer sein könnte, als die niederwertigen Bits vermuten lassen. Das vorgeschlagene Kriterium lautet jedoch, dass eine zu hohe Dateigröße für die Anzahl an reservierten Datenblöcken zu einem Ausschluss des Inodes führt. Wenn jedoch die Anzahl reservierter Blöcke immer größer sein kann, ist ein Ausschluss nur in absoluten Randfällen möglich – etwa, wenn eine Datei so groß ist, dass selbst beim Auffüllen der fehlenden 16 Bit mit Einsen zu wenige Blöcke zur Verfügung stünden. Aufgrund des geringen Mehrwerts bzw. der Betriebssystem-Abhängigkeit wurde sich gegen die Nutzung dieses Suchmusters entschieden.

Prinzipiell könnte jedes Attribut eines Inodes als Objekt eines Suchmusters verwendet werden. Solange einem Nutzer Werte bekannt sind, nach denen er für diese Attribute sucht, könnte eine Reihe weiterer, optionaler und stark semantisch selektiver Suchmuster implementiert werden. Dazu gehört beispielsweise die Angabe einer Gruppen- und Nutzer-ID.

Adressen potentieller Inodes, die allen aktiven Suchmustern genügen, werden in zwei Sammlungen abgespeichert, abhängig davon, ob sie reguläre Dateien oder Verzeichnisse sind. Anhand dieser Adressen kann bei der Rekonstruktion der entsprechende Inode ausgelesen und der dazugehörige Dateiinhalt rekonstruiert werden.

3.1.3 Verzeichnisbaum-Phase

Die potentiellen Inodes von Verzeichnissen, die mit den aktiven Suchmustern gefunden wurden, werden in dieser Phase ausgelesen. Hierfür werden die Extent-Einträge der Inodes auf die gleiche Weise interpretiert, wie es auch in der Inhaltsdaten-Phase geschieht, weswegen die nähere Erläuterung dieses Schrittes in Abschnitt 3.1.4 erfolgt.

Die Einträge des Verzeichnisses werden linear mit Hinblick auf die Namens- und Eintragslängen durchsucht. Da Verzeichnisse stets mit den Einträgen für '.' und '..' beginnen müssen, können bereits zu Beginn des Auslesens eines Verzeichnisses solche Verzeichnis-Inodes nachträglich verworfen werden, für die dieses Kriterium nicht zutrifft. Da HTree-Strukturen mit Eintragslängen ihre internen Metadaten vor linearen Suchen verstecken und, weil in HTrees alle Blätter als eigene Datenblöcke mit herkömmlichen Verzeichniseinträgen realisiert sind, bedarf es keiner zusätzlichen Logik, um die Kompatibilität mit HTrees zu gewährleisten.

Alle ausgelesenen Einträge werden mit ihrer Inodenummer, ihrem Dateinamen und der Inodenummer des Elternverzeichnisses abgespeichert, wodurch sich intern eine logische Baumstruktur ergibt, die der Verzeichnisstruktur des Dateisystems gleicht. Dabei entsprechen Verzeichnisse oftmals inneren Knoten, wogegen reguläre Dateien Blätter repräsentieren. Aus diesem Baum mit Teilnamen wird für jede rekonstruierte Datei während der Namensauflösung der vollständige Dateipfad mittels der Einträge der Elternverzeichnisse generiert.

Jeder Knoten des Baumes, der während einer Namensauflösung für eine rekonstruierte Datei ausgelesen wird, wird markiert. Diese Namensauflösung ist eine Bottom-Up-Rekursion, bei der allen traversierten Knoten ihre Pfade zugewiesen werden. Dabei findet auch eine Markierung statt, die anzeigt, ob der komplette Pfad des inneren Knotens bereits gesetzt wurde. Den Einstiegspunkt bildet die Inodenummer einer regulären Datei. Diese Rekursion bricht ab, wenn sie auf einen markierten Knoten stößt, da er und all seine Elternknoten bereits mit vollständigen Pfaden versehen wurden. Für den Fall, dass ein erwartetes Elternverzeichnis nicht aufgetreten ist, verhindern Abbruchfälle, dass Zyklen im Baum zu Endlosschleifen führen.

Durch den Verzeichnisbaums werden Inodenummern und Dateinamen in Relation gesetzt. Von potentiellen Inodes regulärer Dateien sind allerdings keine Inodenummern bekannt, sondern nur ihre physikalische Adressen. Da in der Initialisierungsphase eine Vielzahl von Dateisystem-Metadaten bekannt geworden ist, können diese Adressen, sofern sie sich in Inode-Tabellen befinden, auf Inodenummern abgebildet werden, ohne dabei auf den Superblock oder die Gruppen-Deskriptor-Tabelle zuzugreifen. Dadurch kann einem gefundenen Inode sein Dateiname und der vollständige Pfad zugeordnet werden. Wird während dieser Berechnung festgestellt, dass sich der potentielle Inode nicht in einer Inode-Tabelle befindet, so wird dieser ignoriert und die dazugehörige Datei nicht rekonstruiert.

Die Berechnung einer Inodenummer aus einer physikalischen Adresse beruht auf dem folgenden Vorgehen. Sind Anfangsadresse s und Endadresse e einer Inode-Tabelle und die Inode-Größe i bekannt, kann aus einer Adresse $a \in [s, e[$ die Inodenummer von a zu $\frac{a-s}{i}$ berechnet werden. Ist dieser Wert nicht ganzzahlig, handelt es sich nicht um eine gültige Inodenummer. Ebenso müssen s und e durch die Inodegröße i teilbar sein, da jede Inode-Tabelle nur volle Blöcke umfasst und die Blöckgröße immer ganzzahlig durch die Inodegröße teilbar ist. Da Inodenummern ab dem Wert 1 durchnummeriert sind, muss der errechnete Wert um 1 erhöht werden, somit erweitert sich der Ausdruck zu $\frac{a-s}{i} + 1$. Dabei unterliegt die Berechnung von s und e einer Annahme, die gelten muss, damit Inodenummern bestimmt werden können. Diese besteht darin, dass die Position der Inode-Tabelle, der Inode-Bitmap und der Block-Bitmap nicht manuell verändert worden sind. Diese Positionen können beliebig sein und gegebenenfalls in der Gruppen-Deskriptor-Tabelle ausgelesen werden. Ist dies der Fall, kann keine Ermittlung der Inodenummern stattfinden, da das Auslesen der Gruppen-Deskriptor-Tabelle und des Superblocks explizit ausgeschlossen werden soll.

Da Inode-Tabellen verschnittfrei angelegt werden und jede Inode-Tabelle die gleiche Anzahl an Inodes umfasst, gilt $e = s + n_{i,BG} \cdot i$ wobei $n_{i,BG}$ die Anzahl der Inodes pro Blockgruppe bezeichnet. Ist eine Flex-Gruppengröße $n_f > 1$ angegeben, befinden sich standardmäßig n_f Inode-Tabellen nacheinander in der ersten Blockgruppe der Flex-Gruppe. Dies hat zur Folge, dass für die zusammenhängenden Inode-Tabellen einer Flex-Gruppe gilt: $e = s + n_f \cdot n_{i,BG} \cdot i$. Somit erweitert sich das Gültigkeitskriterium für die Lage einer Adresse a innerhalb einer Inode-Tabelle im allgemeinen Fall zu $s \leq a < s + n_f \cdot n_{i,BG} \cdot i$.

Wie bereits erwähnt, sind Inodes innerhalb einer Inode-Tabelle nur dann gültig, wenn für ihre Adressen a gilt, dass $\left(\frac{a-s}{i} + 1\right) \in \mathbb{N}$. Die Ermittlung von s hängt nun von verschiedenen Faktoren ab, wie der Blockgruppe bg_a, in der sich a befindet, und sich zu $bg_a = \left\lfloor \frac{a}{b \cdot n_{BG}} \right\rfloor$ berechnet. Hierbei bezeichnen b die Blockgröße in Bytes und n_{BG} die Anzahl der Blöcke pro Blockgruppe. Ist eine Flex-Gruppengröße angegeben, befindet sich womöglich an der eben errechneten Stelle keine Inode-Tabelle, sondern nur in den ersten Blockgruppen jeder Flex-Gruppe. Also muss gelten, dass $\frac{bg_a}{n_F} \in \mathbb{N}_0$.

Die Beginnadresse der errechneten Blockgruppe lässt sich nun durch $bg_a \cdot b \cdot n_{BG}$ bestimmen. An dieser Adresse können sich nun eine Superblock-Kopie, eine Kopie der Gruppen-Deskriptor-Tabelle, ihre zugehörigen Speicherblöcke für späteres Wachstum, eine Inode-Bitmap und eine Block-Bitmap befinden, bevor die Inode-Tabelle beginnt. Superblock-Kopien treten mit Gruppen-Deskriptor-Tabellen und dem Wachstumsspeicher zusammen auf und bilden den für die Sparse-Option variablen Offset o_s in Blöcken.

Dabei beläuft sich die Größe des Superblocks auf einen Block. Der Speicherbereich, den die Gruppen-Deskriptor-Tabelle umfasst, lässt sich dadurch errechnen, dass für jede Blockgruppe ein Gruppen-Deskriptor angelegt wird. So ergeben sich hierfür zunächst $\left\lceil \frac{d \cdot n_G}{b} \right\rceil$ Blöcke für die Tabelle, wobei d für die Größe eines Gruppen-Deskriptors und n_G für die Anzahl der Blockgruppen des Dateisystems steht. Gruppen-

Deskriptoren sind entweder 32 oder 64 Byte lang, je nach dem, ob die Blockadressbreite 32 Bit oder 64 Bit beträgt. Der Wachstumsspeicher für die Gruppen-Deskriptor-Tabelle ist nun so angelegt, dass diese auf $\left\lceil \frac{d \cdot n_G \cdot 1024}{b} \right\rceil$ Blöcke wachsen kann. Die Summe der Blöcke, die für die Gesamtheit aus Superblock, Gruppen-Deskriptor-Tabelle und Wachstumsspeicher angelegt werden, kann allerdings nie 1024 überschreiten. Daraus ergibt sich für den Offset o_s, der für jede Blockgruppe, bzw. solche, die von der Sparse-Option betroffen sind, der Wert $o_s = \min\left\{1024, 1 + \left\lceil \frac{d \cdot n_G \cdot 1024}{b} \right\rceil\right\}$.

Unabhängig von der Sparse-Option befinden sich vor jeder Inode-Tabelle die Block- und Inode-Bitmap, die jeweils einen Block umfassen. Betrachtet man die Flex-Gruppengröße, errechnet sich der zusätzliche Abstand zur Inode-Tabelle o_i zu $2 \cdot n_F$ Blöcken. Dies unterliegt allerdings der Annahme, dass der Aufbau der Blockgruppe nicht vom Standard abweicht, in dem alle Block- und Inode-Bitmaps vor den Inode-Tabellen liegen.

Da der erste Superblock 1024 Byte nach Beginn des Dateisystems liegt, wird bei einer Blockgröße von 1024 Byte, der erste Block des Dateisystems als reservierter Bereich verstanden, womit Blockgruppe 0 erst ab Block 1 beginnt. Dies verschiebt alle Blockgruppen konstant um einen Block, wie es durch die Variable o_r angegeben wird, die genau dann den Wert 1 hat, wenn die Blockgröße 1024 beträgt, und ansonsten den Wert 0 annimmt. Der Beginn der Inode-Tabelle errechnet sich demnach wie in Formel 3.1.

$$s = (bg_a \cdot n_{BG} + o_s + o_i + o_r) \cdot b \tag{3.1}$$

Daraus folgt Formel 3.2 für die Berechnung der Inodenummer einer Adresse a. Hierbei sind o_s und o_r bedingte Variablen. Zusätzlich muss gelten: $a \in [s, e[$ und $\frac{a}{i} + 1 \in \mathbb{N}$ und $\frac{bg_a}{n_F} \in \mathbb{N}_0$.

$$\begin{aligned} o_s &= \min\left\{1024, 1 + \left\lceil \frac{d \cdot n_G \cdot 1024}{b} \right\rceil\right\} \\ o_i &= 2 \cdot n_F \\ o_r &= \begin{cases} 1 & \text{falls } b = 1024 \\ 0 & \text{sonst} \end{cases} \\ bg_a &= \left\lfloor \frac{a}{b \cdot n_{BG}} \right\rfloor \end{aligned}$$

$$f(a) = \left(\frac{a - (bg_a \cdot n_{BG} + o_s + o_i + o_r) \cdot b}{i} + n_{i,BG} \cdot bg_a + 1 \right) \tag{3.2}$$

Mit diesen Berechnungsvorschriften kann die Inodenummer für eine bestimmte Adresse auf einem Ext4-Dateisystem errechnet und somit der Dateiname aus dem Verzeichniseintrag einem Inode zugewiesen werden. Damit kann im Metadaten-Modus die Verzeichnisstruktur aufgebaut werden. Ist ein Verzeichnis nicht rekonstruierbar, so können Dateinamen und -pfade der sich darin befindlichen Dateien nicht korrekt gesetzt werden. In diesem Fall wird die Datei im Zielordner rekonstruiert, mit der Adresse des gefundenen Inodes als Dateinamen.

Da im Inhaltsdaten-Modus als Ext4-Parameter ausschließlich die Ext4-Blockgröße verwendet wird, können Inodenummern nicht berechnet werden, weswegen Verzeichniseinträge nicht interpretiert und so die Verzeichnisstruktur nicht aufgebaut werden kann. Folglich ist die Verzeichnisbaum-Phase nicht Teil des Inhaltsdaten-Modus. Als Pfad wird in diesem Fall das Zielverzeichnis gesetzt, und als Dateiname die Adresse, an der der Inode gefunden wurde.

3.1.4 Inhaltsdaten-Phase

Nachdem für eine reguläre Datei Pfad und Dateiname gesetzt wurden, unabhängig vom verwendeten Modus, werden ihre Inhaltsdaten in der Inhaltsdaten-Phase rekonstruiert. Je nach dem, ob ein betrachteter Inode eine reguläre Datei oder ein Verzeichnis repräsentiert, müssen seine Inhaltsdaten unterschiedlich interpretiert werden, dennoch ist die Verwaltungsstruktur der Inhaltsdaten unabhängig vom Dateityp. Im Falle eines Verzeichnisses wird das Auslesen der Inhaltsdaten in der Verzeichnisbaum-Phase durchgeführt.

Als Verwaltungsmechanismus wird im Ext4-Dateisystem die Extent-Struktur verwendet, wohingegen dies im Ext2/3-Dateisystem mit indirekten Blockzeigern realisiert wird. In der vorliegenden Version des Moduls wird diesbezüglich keine Abwärtskompatibilität zu Ext2/3 angeboten; um diese lässt sich das Werkzeug allerdings erweitern, sobald entsprechende Suchmuster und ein Traversionsalgorithmus bereitgestellt werden. Ebenso werden keine Inline-Dateien und -Verzeichnisse von dieser Version des Moduls unterstützt.

Wie in Kapitel 2 erwähnt, spannt die Extent-Struktur einen Baum auf. Bei der Implementierung des Traversionsalgorithmus für diesen Baum müssen zwei Aspekte beachtet werden: Extent-Einträge liegen nicht zwingend in sortierter Reihenfolge vor und können Dateien aufspannen, die unter Umständen nicht vollständig in den Hauptspeicher passen. Wie diese beiden Aspekte vom vorgestellten Werkzeug behandelt werden, wird in Abschnitt 3.2.4 detailliert dargelegt.

Dadurch, dass im Inhaltsdaten-Modus kein Ausschlusskriterium bezüglich Inode-Tabellengrenzen existiert, können mehrere potentielle Inodes die gleiche Datei bezeichnen, beispielsweise durch Einträge im Journal. Inodenummern sind hierbei nicht verfügbar, sodass ein Kriterium eingeführt werden muss, nach dem Inodes als gleich klassifiziert werden können. Hierfür wurde ein Mechanismus zur Duplikaterkennung bereitgestellt, der sowohl die Dateigröße, als auch die Extent-Datenstrukturen im Inode berücksichtigt.

Die tatsächlichen Dateiinhalte werden hierbei nicht verglichen, sondern ausschließlich die Extent-Struktur. Somit werden Dateien, die durch unterschiedliche Inodes repräsentiert werden, aber den gleichen Inhalt haben, dennoch rekonstruiert. Nicht unterschieden werden Inodes aus Inode-Tabellen und Inodes aus dem Journal. Eine Ausnahme bilden leere Dateien, die unter beiden Gesichtspunkten als Duplikate verstanden werden. Der Inhaltsdaten-Modus stellt aufgrund dessen maximal eine leere Datei wieder her. Da in diesem Modus keine Dateinamen wiederhergestellt werden, ist die Aussagekraft der Anzahl leerer, namenloser Dateien gering. Im Metadaten-Modus sind diese Kriterien nicht notwendig, da nur Inodes mit einer gültigen Inodenummer rekonstruiert werden.

Da durch die Struktur des Ext4-Dateisystems nur ganze Datenblöcke alloziert werden können, Dateien allerdings keine Dateigröße haben müssen, die ein ganzzahliges Vielfaches der Blockgröße sind, entsteht beim Rekonstruieren ein Verschnitt am Ende der Datei. Um zu gewährleisten, dass die MD5- und SHA256-Prüfsummen der rekonstruierten Dateien den erwarteten Wert haben, muss der Verschnitt am Ende der Rekonstruktion korrekt entfernt werden. Dabei wird der die Dateigröße aus dem gefundenen Inode ausgelesen und die Differenz aus der Dateigröße und dem Produkt aus der Anzahl der rekonstruierten Blöcke und der Blockgröße berechnet. Die Dateigröße darf hierbei das besagte Produkt nicht überschreiten. Dies geschieht nachdem die Datei vollständig rekonstruiert und auf die Festplatte geschrieben wurde.

3.1.5 Leerdaten-Phase

Die Leerdaten-Phase ist, wie auch die Verzeichnisbaum-Phase, optional und ergibt sich aus den Ergebnissen der Verzeichnisbaum- und Inhaltsdaten-Phase. Da die Verzeichnisbaum-Phase für diese Phase notwendig ist, kann auch diese nur im Metadaten-Modus ausgeführt werden. Durch das Auslesen der Verzeichnisse werden alle Verzeichniseinträge in den internen Verzeichnisbaum übernommen, unabhängig vom Dateityp oder der Intaktheit der durch diese Verzeichniseinträge repräsentierten Dateien.

Verzeichniseinträge von bislang nicht wiederhergestellten Dateien können zur Rekonstruktion weiterer Informationen genutzt werden. Allein die Rekonstruktion regulärer Dateien bewirkt die Erzeugung von Verzeichnissen und die Markierung der entsprechenden Knoten. Es existieren jedoch Verzeichnisse, die keine regulären Dateien enthalten, wie etwa leere Verzeichnisse. Diese würden ohne die Leerdaten-Phase vom vorgestellten forensischen Werkzeug nicht erstellt werden.

Dateien, die nicht durch die Suchmuster der Inode-Carving-Phase erfasst wurden, können nicht mit ihren Inhaltsdaten rekonstruiert werden, da die dazugehörigen Inodes nicht betrachtet werden. Ihre Verzeichniseinträge stehen allerdings in der Verzeichnisbaum-Struktur zur Verfügung. Diese ermöglichen es, für sie leere Dateien mit den entsprechenden Dateinamen anzulegen. Beispiele solcher Dateien sind symbolische Links oder Gerätedateien.

In der Leerdaten-Phase werden alle nicht markierten Knoten der internen Verzeichnisbaum-Struktur als leere Dateien realisiert. Für diesen Zweck wird in keiner besonderen Reihenfolge über den Verzeichnisbaum iteriert. Die Namensauflösung geschieht auf die selbe Weise wie auch in der Verzeichnisbaum-Phase, da die Inodenummern aus den Verzeichniseinträgen gelesen werden können.

3.2 Aufbau

Das vorgestellte forensische Werkzeug wurde als C++-Modul für das Sleuthkit-Framework (Version 4.1.3) implementiert. Hierbei wurde der C++11-Standard verwendet und das Modul mittels des quelloffenen *gcc*-Kompilierers (Version 4.8.4) übersetzt. Im Folgenden wird auf die genauen Funktionsweisen der eben vorgestellten Phasen eingegangen und die Besonderheiten bei ihren Implementierungen im Detail erläutert.

Im Abschnitt über die Initialisierung wird näher darauf eingegangen, wie aus den übergebenen oder erschlossenen Parametern die weiteren essentiellen Parameter bestimmt werden können. Abhängig vom verwendeten Modus werden unterschiedliche Parametermengen benötigt, die in der Konfigurationsdatei oder als mkfs-Standardwerte vorliegen müssen.

Die anschließende Inode-Carving-Phase verwendet Suchmuster, um Inodes für die spätere Rekonstruktion zu finden. In diesem Abschnitt wird erläutert, wie die Vergleiche mit den Suchmustern implementiert wurden und wie die Inode-Datenstruktur ausgelesen wird. Dabei wird auch auf die Formatierung der Inode-Datenstruktur und ihrer Felder eingegangen. Weiterhin wird das Auslesen der gesamten Festplatte mittels zweier, im Wechsel genutzter Puffer beschrieben. Diese asynchrone *Double-Buffering*-Technik hat den Zweck, die Laufzeit zu verringern.

Darauf folgend wird im Abschnitt über die Verzeichnisbaum-Phase das Schema der internen Verzeichnisbaum-Struktur vorgestellt. Der Aufbau dieser Verzeichnisbaum-Struktur wird anhand einiger Beispiele illustriert. Diesem Vorgang liegt das Auslesen von Verzeichniseinträgen auf dem Dateisystem und ihre Übersetzung in Knoten des Verzeichnisbaums zugrunde. Schließlich wird die Rekursion für die Auflösung eines Dateipfads für eine gegebene Inodenummer dargelegt.

Der Abschnitt über die Inhaltsdaten-Phase behandelt im Detail, wie die Rekonstruktion der Inhaltsdaten der regulären Dateien umgesetzt wurde. Dabei wird eine Rekursion vorgestellt, welche die Extent-Hierarchie einer regulären Datei oder eines Verzeichnisses auf eine lineare Verkettung von Datenblöcken abbildet. Es wird zudem auf die Unterschiede zwischen dem Metadaten- und dem Inhaltsdaten-Modus hingewiesen, die die Rekonstruktion der Dateien betreffen.

Schließlich wird im Abschnitt über die Leerdaten-Phase erklärt, wie anhand des internen Verzeichnisbaumes Informationen über bislang nicht erfasste Dateien genutzt werden können. Durch diesen Ansatz ist es möglich, Namen und Pfade leerer Verzeichnisse und nicht rekonstruierter Dateien wiederherzustellen.

3.2.1 Initialisierung

Zu Beginn der Initialisierungsphase wird die Konfigurationsdatei eingelesen. Alle Parameter, die in dieser nicht angegeben wurden, werden automatisiert gesetzt. Ein Standardwert für den Zielordner der rekonstruierten Dateien, der durch den übergebenen Pfad zum Datenträgerabbild definiert wird, wird vom Sleuthkit zur Verfügung gestellt. Die Größe des Datenträgerabbilds wird vom Werkzeug bestimmt, sofern keine Partitionsgröße angegeben wurde, und der Offset zur entsprechenden Partition wird als 0 angenommen. Alle weiteren Variablen der Konfigurationsdatei werden, falls nicht spezifiziert, mit ihren von der Partitionsgröße abhängigen mkfs-Standardwerten initialisiert, wie bereits in Abschnitt 3.1.1 näher erläutert.

Aus den so zur Verfügung gestellten Werten werden im Metadaten-Modus weitere, für die Berechnung von Inodenummern notwendige Parameter abgeleitet. Zu diesen gehören folgende Werte:

- Anzahl der Blöcke pro Blockgruppe
- Anzahl der Blöcke und Blockgruppen des Dateisystems
- Anzahl der Inodes pro Blockgruppe
- Größe des Speicherbereichs für die wachsende Gruppen-Deskriptor-Tabelle

Die Anzahl der Blöcke pro Blockgruppe ergibt sich, wie in Kapitel 2 bereits erwähnt, durch die Anzahl der Bits die ein Ext4-Block umfasst. Das bedeutet, dass bei einer Blockgröße von 4096 Byte, die Blockgruppe 32768 Blöcke beinhaltet. Mit diesem Wert und der Dateisystemgröße kann die Anzahl der Blöcke und Blockgruppen des Dateisystems bestimmt werden. Sei D die Größe des Dateisystems in Byte, b die Blockgröße in Byte und n_{BG} die Anzahl der Blöcke einer Blockgruppe, dann ergeben sich $n_D = \left\lceil \frac{D}{b} \right\rceil$ Blöcke im Dateisystem und dadurch $n_G = \left\lceil \frac{D}{b \cdot n_{BG}} \right\rceil$ Blockgruppen.

Da im Metadaten-Modus für die Berechnung von Inodenummern der Abstand vom Anfang einer Blockgruppe zu ihrer Inode-Tabelle bekannt sein muss, werden die Breiten aller Speicherbereiche der Metadaten benötigt, die vor der Inode-Tabelle platziert sind. Zu diesem Metadaten gehört die Gruppen-Deskriptor-Tabelle, die für jede Blockgruppe einen Gruppen-Deskriptor beinhaltet. Folglich hängt die Größe der Gruppen-Deskriptor-Tabelle direkt von der Anzahl der Blockgruppen im Dateisystem ab.

Dabei muss beachtet werden, dass vom mkfs-Programm bei der Formatierung eines Ext4-Dateisystems zusätzlich Speicher für die Gruppen-Deskriptor-Tabelle angelegt wird, damit das Dateisystem nachträglich wachsen kann. Dieser reservierte Wachstumsspeicher wird dafür verwendet, dass die Gruppen-Deskriptor-Tabelle maximal auf das 1024-fache ihrer ursprünglichen Größe wachsen kann. Die Größe der Gruppen-Deskriptoren hängt von der Adressbreite des Dateisystems ab und kann 32 Byte bzw. 64 Byte umfassen.

Bei der Berechnung der Anzahl der Inodes pro Blockgruppe spielen verschiedene Parameter eine Rolle. Da durch das Inodeverhältnis angegeben wird, wieviele Byte pro Inode im Schnitt verwaltet werden sollen, muss die Dateisystemgröße durch das Inodeverhältnis v geteilt werden, so dass sich für die Gesamtzahl an Inodes $\frac{D}{v}$ ergibt. Teilt man diesen Wert durch die Anzahl der Blockgruppen des Dateisystems, so ergibt sich ein Richtwert für die Anzahl der Inodes pro Blockgruppe $n_{i,BG}$, so dass $n_{i,BG} = \frac{D}{v \cdot n_D}$.

Da die Anzahl der Inodes allerdings ganzzahlig durch die Blockgröße teilbar sein muss, folgt für die Anzahl der Inodes pro Block n_I mit der Inodegröße i, dass $n_I = \left\lceil \frac{b}{i} \right\rceil$.

Unter diesen Annahmen ergibt sich Formel 3.3 für die Berechnung der Anzahl der Inodes pro Blockgruppe $n_{I,G}$. Für den Inhaltsdaten-Modus sind diese Berechnungen allerdings irrelevant, da dieser nur mit der Blockgröße des Ext4-Dateisystems arbeitet.

$$n_{I,G} = \left\lceil \frac{n_{i,BG}}{n_I} \right\rceil \cdot n_I \tag{3.3}$$

Zusätzlich zu den Ext4-Parametern können für einige der in der Inode-Carving-Phase verwendeten Suchmuster, weitere Parameter angegeben werden. Hierfür relevant sind die Suchmuster nach Zeitstempeln und Zugriffsrechten.

Zur Angabe von Zeitgrenzen werden UNIX-Zeitstempel verwendet, deren Standardwerte 0 und $2^{32} - 1$ sind. Wird keine dieser Grenzen explizit vom Nutzer angegeben, findet keine Konsistenzüberprüfung der Zeitstempel statt. Soll lediglich die Konsistenzprüfung stattfinden, kann als eine der beiden Grenzen explizit das Maximum angegeben werden.

Wurden vom Nutzer in der Konfigurationsdatei keine Zugriffsrechte angegeben, akzeptiert die Inode-Carving-Phase Inodes mit beliebigen Zugriffsrechten. Andernfalls kann eine *Whitelist* akzeptierter Zugriffsrechte angegeben werden. Die Zugriffrechte selbst sind als Ganzzahl zwischen 0 und 511 im dezimalen, oktalen (mit führender 0) oder hexadezimalen (mit führendem 0x) anzugeben.

3.2.2 Inode-Carving-Phase

In der Inode-Carving-Phase wird die Festplatte sequentiell nach potentiellen Inodes durchsucht. Aufgrund der Größe der Datenträgerabbilder und der damit verbundenen, hohen Einlesedauer bietet es sich an, mit der sogenannten Double-Buffering-Methode zu arbeiten. Beim Double-Buffering werden zwei Puffer im Speicher verwendet, von denen jeweils einer mit Inhalt von der Datenquelle (das Datenträgerabbild) befüllt wird, während der Andere zum Arbeiten benutzt werden kann. Sind beide Operationen vollständig ausgeführt worden, tauschen die Puffer die Rollen. Auf diese Weise kann parallel gelesen und gearbeitet werden, wobei nur eine Puffer-Füllzeit anfällt. Hierfür werden dem Programm 128 MiB pro Puffer zur Verfügung gestellt. Da keine Performance-Analyse für verschiedene Puffergrößen vorgesehen ist, ist diese konstant als Makro `BLOCKSIZE` realisiert.

Die Double-Buffering-Komponente wurde mittels `std::async()` und Lambda-Funktionen aus dem C++11-Standard in der Methode `readNewBlock()` implementiert, wie in Listing 3.1 zu sehen ist.

```
int InodeFileReader::readNewBlock()
{
  // Lambda function
  auto lambda = [&] ()
  {
    mutex1.lock();
    if (m_File.is_open())
    {
      m_File.seekg(m_CurrAddress + m_ImgOffset);
      m_File.read((char *)m_LoadBuffer, BLOCKSIZE);
      m_Count = m_File.gcount();
      if (m_Count < BLOCKSIZE)
      {
        m_CurrAddress += m_Count;
      }
      else
      {
        m_CurrAddress += (BLOCKSIZE-m_ExtInodeSize);
      }
    }
    mutex1.unlock();
    mutex2.unlock();
  };

  // Swap buffers
  mutex2.lock();
  char *tmp = m_LoadBuffer;
  m_LoadBuffer = m_CurrBuffer;
  m_CurrBuffer = tmp;
  m_Loaded = m_Count;

  // Async reload of buffer
  if (m_CurrAddress < (m_ImgSize+m_ImgOffset))
  {
    std::async(std::launch::async, lambda);
  }
  return m_Loaded;
}
```

Listing 3.1: Methode zur Umsetzung des Double-Bufferings mittels Lambda-Funktion

Da im Falle eines Verschnitts am Ende des Datenträgerabbilds der Puffer nicht vollständig gefüllt ist, dürfen die nicht aktualisierten Daten am Ende des Puffers nicht interpretiert werden. Diese Methode gibt aus diesem Grund die Anzahl der gelesenen Bytes zurück, um den Bereich der vom nächsten Aufruf zu interpretierenden Daten zu definieren.

Der vom Zeiger `m_LoadBuffer` referenzierte Puffer beinhaltet die neu geladenen Daten, die durch die Lambda-Funktion vom Datenträgerabbild gelesen werden. Parallel dazu wird auf dem Puffer, der vom Zeiger `m_CurrBuffer` referenziert wird, gearbeitet und nach potentiellen Inodes mittels Suchmustern gesucht. In `m_CurrAddress` wird die Speicheradresse auf dem Datenträgerabbild, ab der der von `m_CurrBuffer` referenzierte Puffer Daten enthält, gespeichert und nach jedem Lesen um die Anzahl der gelesenen Bytes erhöht. Wenn beide Prozesse vollständig durchgelaufen sind, werden die entsprechenden Synchronisationspunkte in Zeile 21 und 22 freigegeben und anschließend die Puffer ab Zeile 27 getauscht. Dadurch sind im vom `m_CurrBuffer` referenzierten Puffer die neuen Daten vorhanden; die Daten im Puffer, der durch `m_LoadBuffer` referenziert wird, können nicht mehr gebraucht werden, weswegen diese durch einen weiteren Ladevorgang überschrieben werden können.

Um alle potentiellen Inodes zu erfassen, werden die Pufferadressen so gewählt, wie es in Abbildung 3.1 zu sehen ist. Hierbei wird der Versatz zwischen zwei Puffern nicht auf die Puffergröße `BLOCKSIZE` gesetzt, sondern auf `BLOCKSIZE - m_ExtInodeSize`. Auf diese Weise wird gewährleistet, dass erstens alle potentiellen Inodes erfasst und zweitens keine Inodes gelesen werden, die sich nicht vollständig in einem Puffer befinden.

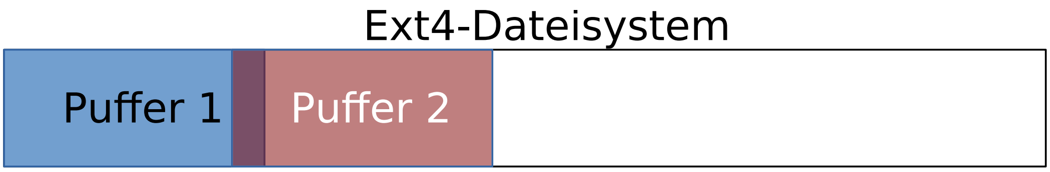


Abbildung 3.1: Überschneidung beim Double-Buffering

Da der Abstand der Anfänge zweier aufeinanderfolgender Pufferfenster kleiner ist als die Pufferfenstergröße, gibt es Überschneidungen zwischen den Puffern. Um keine potentiellen Inodes durch das Lesen über die Puffergrenzen hinaus zu erkennen, wird von einem Pufferfenster w_i der Überschneidungsbereich zum nächsten Pufferfenster w_{i+1} nicht interpretiert. Auf diese Weise vom Fenster w_i nicht erfasste Speicherbereiche werden anschließend von w_{i+1} analysiert. Somit wird der Speicherbereich doppelt geladen, allerdings nicht doppelt gelesen.

Im Inhaltsdaten-Modus wird für die Größe der Inodes kein korrekter Wert vorausgesetzt. Dennoch befinden sich alle für die Suchmuster relevanten Daten in den ersten 128 Byte eines potentiellen Inodes. Folglich entstehen keine Fehler, wenn für die Größe des Überschneidungsbereichs mindestens 128 Byte gewählt werden. Damit ist der Überschneidungsbereich unabhängig von der tatsächlichen Inodegröße; Im Inhaltsdaten-Modus wird an dieser Stelle die durch die Konfigurationsdatei oder die mkfs-Standardwerte angegebene Inodegröße verwendet. Selbst wenn diese falsch ist, entstehen hierdurch keine Fehler.

Aus diesen Pufferfenstern werden diejenigen Daten geladen, die für die Suche nach Inodes mittels Suchmustern benötigt werden. Um beispielsweise die Zugriffsrechte, die sich in den ersten 9 Bit eines Inodes befinden, herauszufiltern, müssen 2 Byte gelesen und die entsprechenden Daten mittels bitweisen Operationen extrahiert werden, wie es in Listing 3.2 zu sehen ist.

```
 1  for (size_t i = 0; i < Buffer.size() - m_InodeSize; ++i)
 2  {
 3    unsigned short lowerFileMode = static_cast<unsigned short>((unsigned char)Buffer[i]);
 4    unsigned short upperFileMode = static_cast<unsigned short>((unsigned char)Buffer[i+1]);
 5
 6    unsigned short bit_9 = upperFileMode & 0x01;
 7    bit_9 = bit_9 << 8;
 8    unsigned short permission = bit_9 | lowerFileMode;
 9    ...
10  }
```

Listing 3.2: Extrahieren der Zugriffsrechte aus dem Puffer

Der Wert in `lowerFileMode` entspricht den unteren 8 Bit der Zugriffsrechte eines potentiellen Inodes. Die Kombination aus Bitmasken und logischen, bitweisen Verschiebeoperationen (Zeile 6 bis 8) trägt der Tatsache Rechnung, dass vom höherwertigen Byte lediglich ein Bit von Interesse ist. Dieses Bit wird in `upperFileMode` gespeichert und an `lowerFileMode` angeknüpft, um einen 2 Byte großen Wert zu generieren, der die Zugriffsrechts-Konfiguration des Inodes angibt.

Die Zugriffsrechte werden als 9-Bit-Zahl repräsentiert, weshalb vom Benutzer übergebene Zugriffsrechte als Ganzzahl zwischen 0 und 511 erwartet werden, wie schon in Abschnitt 3.2.1 erwähnt wurde. Mit den übergebenen Zugriffsrechten, die in einem Container des Typs `std::vector<unsigned short>` gespeichert werden, werden die extrahierten 9 Bit des potentiellen Inodes mit dem Suchmuster abgeglichen, wie in Listing 3.3 zu sehen ist.

```
 1  if (PermissionVec.size() > 0)
 2  {
 3    bool noPermissionFound = true;
 4    for (auto it : PermissionVec)
 5    {
 6      if (it == permission)
 7      {
 8        noPermissionFound = false;
 9        break;
10      }
11    }
12    if (noPermissionFound == true)
13      continue;
14  }
```

Listing 3.3: Vergleichsoperation für Zugriffsrechte

Der Wert für `permission` folgt aus Listing 3.2, ebenso wie die `continue`-Anweisung in Zeile 13. Diese Anweisung bezieht sich auf die äußere `for`-Schleife, die dadurch fortgeführt werden soll. Die Bedingung in Zeile 12 trifft genau dann zu, wenn die gefundenen 9 Bit keinem Zugriffsrecht entsprechen, das durch den Nutzer spezifiziert wurde. In diesem Fall wird die byteweise Suche nach Inodes mit der nächsten Adresse fortgeführt.

Die Auswertung aller weiteren Suchmuster verläuft auf die gleiche Weise: Die entsprechenden Bytes des potentiellen Inodes werden aus dem Arbeitspuffer geladen und gegebebenfalls mit Bitmasken und bitweisen Verschiebungen zu den benötigten Werten zusammengefügt. Bei der Auswertung eines Suchmusters wird mit einer Menge akzeptierter Werte gearbeitet.

Das vorgestellte Programm erlaubt bei der Spezifikation gültiger, sekundengenauer Zeitstempel allerdings keine beliebigen Mengen, sondern ein Zeitintervall. Sollen mehrere Zeitintervalle überprüft werden, muss das Modul demnach mehrfach ausgeführt werden. Zusätzlich zur Intervalleingrenzung findet eine Konsistenzüberprüfung der gefundenen Zeitstempel statt. Im Folgenden wird von `mtime`, `ctime` und `dtime` gesprochen, wie sie auch in Kapitel 2 verwendet wurden. Der `crtime`-Zeitstempel wird hierbei ignoriert, weil er nicht zu den unteren 128 Byte der Inode-Datenstruktur gehört und deswegen nicht allgemeingültig als verfügbar angesehen werden kann. Ebenso wird der `atime`-Zeitstempel nicht mit betrachtet, da seine Aktualisierung nicht konsistent über alle Betriebssysteme gleich geschieht und somit logisch inkonsistente Zustände gültig sein können. Diese Inkonsistenzen werden unter manchen Betriebssystemen in Kauf genommen, um die Anzahl der Schreiboperationen zu reduzieren.

Zur Konsistenzüberprüfung für die Gültigkeit der Zeitstempel gehören folgende Bedingungen:

1. `mtime` muss kleiner gleich `ctime`.
2. `dtime` muss entweder 0 oder größer gleich `mtime` und `ctime` sein.
3. `mtime`, `ctime` und `dtime` müssen zwischen der spezifizierten unteren und oberen Schranke liegen.

Durch die erste Bedingung werden Inkonsistenzen während der Veränderung einer Datei ausgeschlossen. Bei einer Modifikation der Inhaltsdaten einer Datei werden meist auch deren Metadaten verändert, da Zeitstempel, Dateigröße usw. aktualisiert werden müssen. Wenn eine Modifikation der Inhaltsdaten in dem entsprechenden Zeitstempel dokumentiert wird, bewirkt dies die Modifikation der Metadaten, da der `ctime`-Zeitstempel selbst ein Metadatum ist. Eine Datei mit einem `ctime`-Zeitstempel, der weiter in der Zukunft liegt, als ihr `mtime`-Zeitstempel, wird folglich als inkonsistent angenommen.

Die zweite Bedingung betrifft Inkonsistenzen des Löschzeitpunkts einer Datei. Ungelöschte Dateien haben einen Löschzeitpunkt von 0; dies muss gesondert berücksichtigt werden. Im Falle einer gelöschten Datei muss ihr `dtime`-Zeitstempel nach der letzten Änderung der Metadaten und Inhaltsdaten liegen, da an gelöschten Dateien keine Änderungen vorgenommen werden können.

Die Eingrenzung der Zeitstempel potentieller Inodes auf ein Intervall wird durch die dritte Bedingung ausgedrückt. Wenn mindestens eine Schranke des Zeitintervalls angegeben wurde, müssen sich alle Zeitstempel im dadurch definierten Intervall befinden. Für die Erhebung der Werte der Zeitstempel werden jeweils nur die unteren 32 Bit in Betracht gezogen, wodurch nur sekundengenaue Angaben bis ins Jahr 2106 angenommen werden können, jedoch lässt sich dies mit wenigen Änderungen auf nanosekundengenaue Angaben bis ins Jahr 2514 erweitern. Die dafür notwendigen Zusatzinformationen befinden sich jedoch nicht in den unteren 128 Byte eines Inodes. Da die Suchmuster unabhängig von der tatsächlichen Inodegröße gültig sein sollen und die Inodegröße mindestens 128 Byte groß ist, beschränken sich die Muster auf die ersten 128 Byte eines Inodes.

Die Unterscheidung der Dateitypen behandelt gleichzeitig das Dateityp-Suchmuster und ist in Listing 3.4 dargestellt. Hierbei bezeichnet `upperFileMode` die Variable, die bereits in Listing 3.2 definiert wurde.

```
 1  unsigned short type = upperFileMode & 0xF0;
 2  type = type << 8;
 3
 4  if (type == 0x8000)
 5  {
 6    m_RegFiles.push_back(address);
 7  }
 8  else if (type == 0x4000)
 9  {
10    m_Directories.push_back(address);
11  }
12  else if (type == 0x1000 || type == 0x2000 || type == 0x6000 ||
13           type == 0xA000 || type == 0xC000)
14  {
15    ...
16  }
```

Listing 3.4: Mustervergleich nach Dateityp

Die physikalischen Adressen aller potentiellen Inodes, die allen aktivierten Suchmustern genügen, werden in den Zeilen 6 und 10 in zwei separate Container des Typs `std::vector<size_t>` geschrieben. In welchen der beiden Container eine Adresse gespeichert wird, hängt davon ab, ob der Inode ein Verzeichnis (Zeile 8 bis 11) oder eine reguläre Datei (Zeile 4 bis 7) repräsentiert. Würden weitere Dateitypen unterstützt werden, müsste es für diese ebenfalls separate Container geben, da ihre Inhaltsdaten vom Dateityp abhängig rekonstruiert werden müssten.

Am Ende dieser Phase befinden sich die physikalischen Adressen aller akzeptierten, regulären Dateien in `m_RegFiles` und die der akzeptierten Verzeichnisse in `m_Directories`.

3.2.3 Verzeichnisbaum-Phase

In der Verzeichnisbaum-Phase werden die Inodes, die sich an den Adressen aus `m_Directories` befinden, interpretiert. Die Verzeichniseinträge der Inodes von Verzeichnissen, die sich auf diese Weise auslesen lassen, werden in die interne Baumstruktur `m_InodeTree` überführt. Diese Baumstruktur besteht aus Strukturen, deren Datenschema für innere und äußere Knoten identisch ist. Diese Datenstruktur, `directoryTree`, ist in Listing 3.5 definiert.

In dieser Datentruktur befindet sich die Inodenummer des Elternverzeichnisses, der Name des Eintrags und einen Boole'schen Wert, der als Markierung gilt und angibt, wie der Namenseintrag zu interpretieren ist. Ist `complete` (Zeile 4) auf `false` gesetzt, befindet sich in `name` (Zeile 5) lediglich der Name der Datei selbst.

```
1 typedef struct directoryTree
2 {
3   size_t inodeParent;
4   bool complete;
5   std::string name;
6 } directoryTree;
```

Listing 3.5: Struktur für den internen Verzeichnisbaum

Ansonsten wurde bereits die vollständige Namensauflösung durchgeführt und `name` beinhaltet den vollständigen Pfad der Datei bis zum Wurzelverzeichnis. In `std::map<size_t, directoryTree> m_InodeTree` wird mit der Inodenummer als Schlüsselwert diese Struktur für jeden Verzeichniseintrag angelegt. Da die eigene Inodenummer den Schlüssel dieser Abbildung darstellt, enthält die Struktur nicht die Inodenummer.

Weil sich ein Verzeichnis über mehrere Datenblöcke erstrecken kann, wird im ersten Datenblock des Verzeichnisses überprüft, ob sich in ihm die Einträge für '.' und '..' befinden. Da der Eintrag für '.' das Elternverzeichnis aller Dateien des Verzeichnisses ist, dient dieser Wert als `inodeParent` (Zeile 3) aller im Folgenden einzufügenden Baumknoten. Der `name` jedes Verzeichniseintrags kann direkt übernommen werden, `complete` wird zunächst auf `false` gesetzt. Die Einträge für '.' und '..' dürfen hierbei nicht mit übernommen werden: Beide Einträge kommen an anderer Stelle in der Verzeichnishierarchie als Teil eines Verzeichnisses vor und würden sonst mehrfach eingefügt werden; ein solches Konfliktszenario ist in Abbildung 3.2 zu sehen. Die einzige Ausnahme bildet das Wurzelverzeichnis, das nie das Kind eines Knotens und sein eigener Elternknoten ist. Dieses Verzeichnis wird gesondert behandelt.

```
Inodenummer      Name                 Inodenummer      Name
------------------------              ------------------------
25               .                    13               .
13               ..                   2                ..
39               wget                 25               bin
40               cut                  26               lib
...                                   ...
```

Abbildung 3.2: Verzeichnisse mit dazugehörigen Verzeichniseinträgen

Im Beispiel treten in beiden Verzeichnissen Einträge auf, die die Inodenummer 13 und 25 haben. Die Inodenummern bezeichnen zwar stets die selben Dateien, jedoch soll der Name der Datei ausgelesen werden. Um keine Überschreibungskonflikte behandeln zu müssen, werden '.' und '..' nicht in den Baum aufgenommen. Der Wert `complete` wird nur für das Wurzelverzeichnis gesetzt. Dieses wird dadurch erkannt, dass die Inodenummern für '.' und '..' dieselben sind. Für den Fall, dass Verzeichniseinträge beschädigt sind und so mehrere „Wurzelverzeichnisse“ entstehen, generiert das vorgestellte forensische Werkzeug mehrere Wurzelverzeichnisse mit ihren Inodenummern im Namen und somit einen Verzeichniswald.

Das `complete`-Flag dient daher als Abbruchkriterium für eine später definierte Rekursion zur Namensauflösung, da es angibt, ob `name` den kompletten Pfad der Datei enthält. Befindet sich in `name` der komplette Pfad der Datei, muss ein Kindknoten, der über diesen Knoten traversiert, lediglich seinen Teilnamen an diesen Pfad konkatenieren, um seinen Gesamtpfad zu erhalten. Auf diese Weise werden bereits traversierte Verzeichnispfade nicht mehrfach ausgewertet. Teilname und Gesamtpfad sind beim Wurzelverzeichnis identisch, womit es einen universellen Abbruchfall darstellt.

In Abbildung 3.3 ist ein Beispiel für eine mögliche Verzeichnisstruktur abgebildet, welches ab dem Wurzelknoten mit der Inodenummer 2 beginnt. Quadrate stellen hierbei Verzeichnisse dar, wobei Kreise reguläre Dateien repräsentieren.

Im Wurzelverzeichnis befinden sich 2 Unterverzeichnisse, `usr` und `etc` mit den Inodenummern 13 und 14. In diesen Verzeichnissen befinden sich weitere Verzeichnisse und reguläre Dateien, jeweils mit ihrer beispielhaften Inodenummer. Die Verzeichniseinträge aus den Verzeichnissen 2, 13, 14 und 25 sind in Tabelle 3.2 dargestellt, wie sie in der internen Baumstruktur im forensischen Werkzeug gespeichert werden würden.

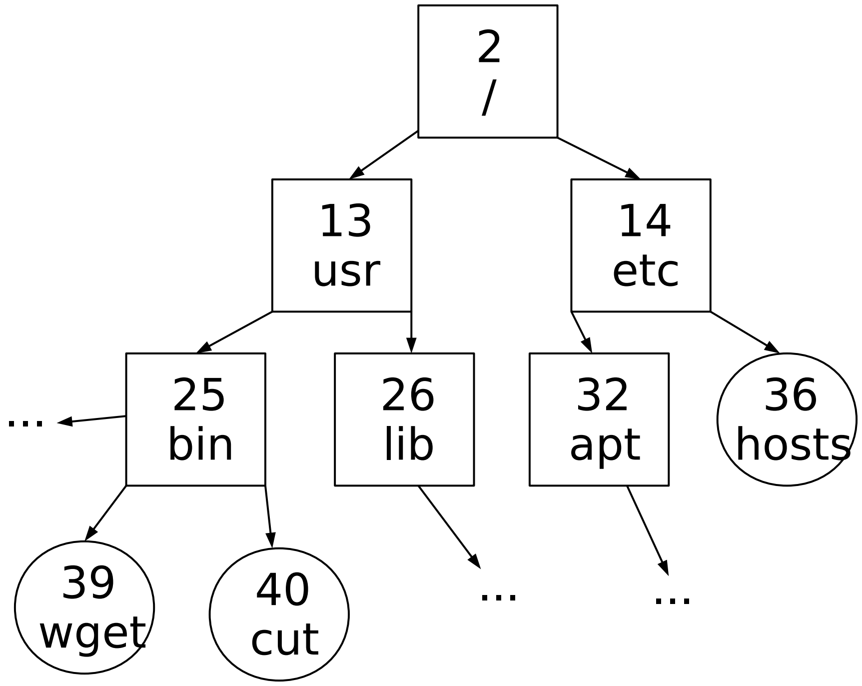


Abbildung 3.3: Verzeichnisstruktur ab Wurzelverzeichnis mit Inodenummern und Namen

Inodenummer	Elternnummer	Name	Pfad vollständig?
2	2	/	Ja
13	2	usr/	Nein
14	2	etc/	Nein
25	13	bin/	Nein
26	13	lib/	Nein
32	14	apt/	Nein
36	14	hosts	Nein
39	25	wget	Nein
40	25	cut	Nein

Tabelle 3.2: Interpretierte Verzeichniseinträge in der inneren Baumstruktur

Die Elternnummer stellt hierbei die Inodenummer des Elternverzeichnisses dar und wurde durch den '.'-Eintrag innerhalb des Verzeichnisses bestimmt. Inodenummer und Name sind durch das Auslesen der Verzeichniseinträge entstanden.

Um ungültige Einträge in der internen Baumstruktur nicht zuzulassen, werden die Ergebnisse zunächst in einer temporären Struktur abgespeichert. Fehler können jederzeit während des Einlesens eines Verzeichnisses auftreten und sie invalidieren das gesamte Verzeichnis. Weiterhin können syntaktisch korrekte Verzeichnisse mit beschädigten Inodenummern zu Konflikten führen, so etwa, wenn zwei Verzeichnisse die selbe Inodenummer für sich beanspruchen. Erst, wenn die temporäre Struktur mit dem gesamten betrachteten Verzeichnis aufgefüllt und vollständig konsistent ist, kann diese in die interne Baumstruktur `m_InodeTree` übernommen werden. Wenn alle Verzeichnisse ausgelesen wurden, befinden sich alle Verzeichniseinträge als `directoryTree`-Knoten in der internen Baumstruktur wieder, wie es auch in Tabelle 3.2 zu sehen ist.

Der Zweck des Baumes ist die Generierung vollständiger Pfadnamen für alle regulären Dateien und Verzeichnisse. Den Einstiegspunkt dieser Namensauflösung bildet die Rekonstruktion regulärer Dateien: Die Adressen der betreffenden Inodes befinden sich in `m_RegFiles`. Nachdem die zugehörige Inodenummer, wie in Sektion 3.1.3 erläutert, berechnet wurde, wird diese in `m_InodeTree` nachgeschlagen und der zugehörige Dateipfad erzeugt, wie es in Listing 3.6 zu sehen ist.

```
1  std::string InodeCarver::directoryTreeRec(size_t a_InodeNum, size_t a_Depth)
2  {
3    std::string ret = "";
4
5    // Loop detection
6    if (a_Depth > MAXTREEDEPTH)
7    {
8      ret = "LoopDir";
9      m_InodeTree[a_InodeNum].inodeParent = 0;
10     m_InodeTree[a_InodeNum].complete = true;
11     m_InodeTree[a_InodeNum].name = ret;
12     return ret;
13   }
14   // Found a point in the tree where the recursion can end
15   if (m_InodeTree[a_InodeNum].complete == true)
16   {
17     ret = m_InodeTree[a_InodeNum].name;
18     return ret;
19   }
20
21   // Continue recursion
22   ret = directoryTreeRec(m_InodeTree[a_InodeNum].inodeParent, a_Depth+1);
23   ret += "/" + m_InodeTree[a_InodeNum].name;
24   m_InodeTree[a_InodeNum].name = ret;
25   m_InodeTree[a_InodeNum].complete = true;
26
27   return ret;
28 }
```

Listing 3.6: Rekursion für das Auslesen des internen Verzeichnisbaums

Ein Abbruchfall wird in Zeile 6 durch das Makro `MAXTREEDEPTH` realisiert. Dieses gibt eine maximale Tiefe des Verzeichnisbaums an. Bei Knoten von einer höheren Tiefe wird von einem Zyklus im Verzeichnisgraph ausgegangen. Verzeichniszyklen werden nicht an das Wurzelverzeichnis, sondern an ein separates Zyklusverzeichnis mit dem Namen `LoopDir` angeknüpft. Das Argument `a_Depth` dient als Tiefenzähler und wird in jedem Rekursionsschritt um eins erhöht. In Zeile 9 bis 12 wird ein Wurzelknoten angelegt, der den Zyklus unterbricht. Durch das parallele Anlegen mehrerer Wurzeln entsteht anstelle eines Verzeichnisbaums ein Verzeichniswald.

Der `complete`-Wert illustriert den nächsten Abbruchfall in Zeile 15. Wenn dieser zuvor für einen Knoten gesetzt wurde, so endet die Rekursion bei diesem und sein Dateipfad wird als Rückgabewert genutzt. Auf diese Weise werden bereits ermittelte Dateipfade nicht mehrfach ausgewertet. Da zu Beginn nur der Wurzelknoten der Verzeichnishierarchie den `complete`-Wert gesetzt hat, ist dies initial der einzige Knoten, der einen fehlerfreien Abbruch ermöglicht.

Bei der Rekursion wird für jeden traversierten Knoten auf dem Dateipfad der regulären Datei, die den Einstieg darstellt, der vollständige Dateipfad aufgelöst. Dies geschieht in den Zeilen 22 bis 24. Hierbei wird für diese Einträge der `complete`-Wert in Zeile 25 gesetzt, da ihre Zwischenergebnisse für weitere Dateien als Abbruchfall genutzt werden können. Da die Dateipfadrekursion nur für reguläre Dateien begonnen wird, werden in diesem Schritt keine leeren Verzeichnisse rekonstruiert; selbst dann nicht, wenn diese als Eintrag in der Verzeichnisstruktur stehen.

Für das Beispiel in Abbildung 3.3 würde nach der Rekonstruktion aller Pfade die innere Struktur wie in Tabelle 3.3 aussehen. In diesem Beispiel wird angenommen, dass keine regulären Dateien in `/usr/lib/` notiert sind. Wenn sich in diesen Verzeichnissen keine regulären Dateien befänden, würden sie nicht von der Rekursion belangt werden. Das Beispiel `apt/` bildet so eines, denn es wird angenommen, dass sich darunter keine regulären Dateien befinden. Somit wurde dieser Eintrag in der Verzeichnisbaum-Struktur nicht aktualisiert.

Inodenummer	Elternnummer	Name	Pfad vollständig?
2	2	/	Ja
13	2	/usr/	Ja
14	2	/etc/	Ja
25	13	/usr/bin/	Ja
26	13	/usr/lib/	Ja
32	14	apt/	Nein
36	14	/etc/hosts	Ja
39	25	/usr/bin/wget	Ja
40	25	/usr/bin/cut	Ja

Tabelle 3.3: Interpretierte Verzeichniseinträge in der inneren Baumstruktur

3.2.4 Inhaltsdaten-Phase

Bei der Rekonstruktion von Inhaltsdaten müssen zwei Eigenschaften der Extent-Struktur der Inodes beachtet werden. Extent-Einträge sind im Allgemeinen ungeordnet und können ein großes Datenvolumen aufspannen.

Sowohl bei der Wiederherstellung von regulären Dateien, als auch von Verzeichnissen, müssen Anforderungen an die logische Ordnung der Datenblöcke nach Dateiinhalt gestellt werden. Im Falle von Verzeichnissen müssen sich im logisch ersten Datenblock die Verzeichniseinträge für '.' und '..' befinden; alle weiteren Blöcke dürfen in beliebiger Ordnung auftreten, solange keine HTree-Suche durchgeführt wird. Reguläre Dateien beinhalten im Allgemeinen Inhaltsdaten, über die keine Aussage getroffen werden kann und müssen daher stets vollständig in logischer Reihenfolge rekonstruiert werden. Der algorithmischen Einfachheit halber werden die Datenblöcke von Verzeichnissen in logischer Reihenfolge eingelesen, da die hierfür benötigte Funktionalität bereits für die Rekonstruktion regulärer Dateien zur Verfügung steht.

Weiterhin kann ein Extent-Baum Dateien mit Größen von bis zu 16 TiB umfassen. Ein derartiges Datenvolumen kann nicht im Allgemeinen im Hauptspeicher vorgehalten werden. Folglich bietet sich eine datenblockweise Ausgabe in eine Zieldatei an.

Im Hinblick auf diese beiden Eigenschaften werden die Extent-Bäume mittels einer Rekursion ausgelesen, in der in jedem inneren Knoten in logischer Reihenfolge abgestiegen wird. Auf diese Weise werden die Blätter in logischer Reihenfolge traversiert und entweder interpretiert oder herausgeschrieben, abhängig davon, ob es sich um ein Verzeichnis oder eine reguläre Datei handelt. Da die Inhaltsdatenblöcke in logischer Reihenfolge verarbeitet werden, können diese ohne Weiteres an die Zieldatei angehängt werden. Der Speicherbedarf überschreitet so zu keinem Zeitpunkt einen Datenblock. Zudem kann so die Anforderung an den Anfang eines Verzeichnisses überprüft werden. Eine Annahme wird jedoch getroffen: Da die gesamte Verzeichnishierarchie in Form der internen Baumstruktur in den Hauptspeicher übernommen wird, muss dieser zu jeder Zeit vollständig in den Hauptspeicher passen.

Verläuft die Extent-Baum-Rekursion frei von Fehlern, werden die Inhaltsdaten regulärer Dateien blockweise in ihre Zieldateien geschrieben. Im Metadaten-Modus sind hierfür die entsprechenden Verzeichnisnamen aus der Verzeichnisbaum-Phase bekannt. Vollständige Dateipfade, und damit die entsprechenden Verzeichnisse, werden beim Rekonstruieren regulärer Dateien erzeugt. Listing 3.7 skizziert vereinfacht den Beginn der Rekursion zum Auslesen der Extent-Baum-Struktur für reguläre Dateien.

Hierbei stellt die Funktion `determineInodeNumber(size_t)` in Zeile 8 die Berechnung der Inodenummer anhand einer gegebenen Adresse dar, wie sie algorithmisch in Abschnitt 3.1.3 beschrieben wurde. Mittels der Funktion `directoryTreeRec(size_t, size_t)` in Zeile 9 werden die Dateipfade, die durch die interne Verzeichnisbaum-Struktur gegeben sind, erzeugt bzw. ausgelesen. Diese Funktion wurde im Detail in Listing 3.6 dargestellt. Die Funktion `convertToString(size_t)` (Zeile 13) dient zur Darstellung einer `size_t`-Ganzzahl als Zeichenkette.

Die hier gezeigte Funktion `recoverData()` iteriert über alle gefundenen potentiellen Inodes regulärer Dateien und bestimmt im Metadaten-Modus ihre Inodenummern und Dateinamen. Im Inhaltsdaten-Modus

```
1  void recoverData()
2  {
3    for (int i = 0; i < m_RegFiles.size(); ++i)
4    {
5      std:string fileName;
6      if (metaDataMode == true)
7      {
8        size_t inodeNum = determineInodeNumber(m_RegFiles[i]);
9        fileName = directoryTreeRec(inodeNum, 0);
10     }
11     else
12     {
13       fileName = convertToString(m_RegFiles[i]);
14     }
15     createFile(fileName);
16     bool check = extentRecursionRoot(m_RegFiles[i]);
17     if (check == false)
18       deleteCurrentFile(fileName);
19   }
20 }
```

Listing 3.7: Vereinfachter Code für das Iterieren über die potentiellen Inodes regulärer Dateien

wird als Dateiname die physikalische Adresse des betreffenden Inodes verwendet (Zeile 6 bis 10), da diese eindeutig ist. Diese Dateien werden angelegt, ehe die Rekursion mittels `extentRecursionRoot(size_t)` (Zeile 16) aufgerufen wird. Zu jedem Zeitpunkt in der Rekursion kann festgestellt werden, dass die Extent-Struktur defekt ist. Bei der Wiederherstellung einer regulären Datei wird in diesem Fall die Rekursion abgebrochen und die partielle Ausgabedatei gelöscht, wie in den Zeilen 17 bis 19 zu sehen.

Tritt ein Defekt beim Rekonstruieren eines Verzeichnisses auf, wird sein temporärer Teilbaum verworfen. Fehler beim Auslesen von Extent-Bäumen können beispielsweise dadurch zustande kommen, dass einer der inneren Knoten des Extent-Baums keinen Extent-Header besitzt oder, dass die angegebene Anzahl der Extent-Einträge eines Knotens die durch die Blockgrenzen vorgegebene Anzahl übersteigt. Auch Fehler beim Lesen eines Datenblocks beenden die Rekursion und invalidieren das jeweilige Teilergebnis.

In Listing 3.8 wird die Funktion `extentRecursionRoot(size_t)` näher erläutert, die mit der Adresse eines Inodes aufgerufen wird.

```
1  bool extentRecursionRoot(size_t address)
2  {
3    readInode(address);
4    if (inode_not_valid)
5    {
6      return false;
7    }
8    else
9    {
10     size_t check = extentRecursion (extent.address, extent.depth, extent.entries);
11     if (check == 0)
12       return false;
13   }
14   return true;
15 }
```

Listing 3.8: Vereinfachter Code für das Auslesen der Inhaltsdaten, beginnend beim Inode

Dieser Inode wird mit seinen Extent-Einträgen in Zeile 3 ausgelesen. Danach wird in Zeile 4 bis 7 überprüft, ob dieser potentielle Inode gültig ist und rekonstruiert werden kann. Mit der Adresse der Extent-Einträge wird die Wiederherstellung der Datei mittels der Funktion `extentRecursion` (Zeile 10) durchgeführt. Anschließend werden in den Zeilen 11 bis 13 in der Rekursion aufgetretene Fehlerfälle an die aufrufende Funktion weitergereicht.

Listing 3.9 beschreibt die Rekursion über die Extent-Struktur. Dabei wird die Tiefe des Extent-Baumes und die Anzahl der Einträge, die sich an der übergebenen Adresse befinden aus dem Extent-Eintrag ausgelesen und übergeben. Mit diesen Werten finden dann Konsistenzüberprüfungen statt; diese werden durch das Feld `extent_not_valid` (Zeile 8) dargestellt.

```
 1 int extentRecursion(size_t address, int depth, int entries)
 2 {
 3   readExtent(address);
 4   int curBlock = 0;
 5   int entriesDone = 0;
 6   int minBlock = INT_MAX;
 7   bool minBlockSet = false;
 8   if (extent_not_valid)
 9     return 0;
10   else {
11     for (int k = 0; k < entries * 2; ++k) {
12       for (int i = 0; i < entries; ++i) {
13         if (extent[i].block == curBlock)
14         {
15           if (depth == 0)
16           {
17             char *data = readDatablocks(extent[i].address);
18             writeToFile(data);
19             curBlock += writtenBlocks;
20             minBlock = INT_MAX;
21             minBlockSet = true;
22           }
23           else
24           {
25             size_t num = extentRecursion (extent[i].address, depth-1);
26             if (num == 0)
27               return 0;
28 
29             curBlock += num;
30             minBlock = INT_MAX;
31             minBlockSet = true;
32           }
33           ++entriesDone;
34         }
35         else
36         {
37           if (extent[i].block > curBlock && extent[i].block <= minBlock) {
38             minBlock = extent[i].block;
39             minBlockSet = false;
40           }
41         }
42       }
43       if (entriesDone == entries)
44         break;
45 
46       if (minBlockSet == true) {
47         minBlockSet = false;
48         continue;
49       }
50       if (minBlock > curBlock) {
51         if (minBlock == INT_MAX)
52           break;
53         char *data = fillWithZero((curBlock-minBlock)*Blocksize);
54         writeToFile(data);
55         curBlock = minBlock;
56         minBlock = INT_MAX
57       }
58     }
59   }
60   return curBlock;
61 }
```

Listing 3.9: Vereinfachter Code für die Rekursion des Auslesens der Inhaltsdaten

Da die referenzierten Extent-Strukturen nicht in sortierter Reihenfolge vorliegen, muss eine Form der sortierenden Suche angewendet werden. Die Funktion `extentRecursion` bedient sich hierzu einer Suche mit zwei verschachtelten Schleifen (Zeile 11 und 12), da die zu sortierenden Elemente häufig, aber nicht immer, in logischer Reihenfolge vorliegen. Der Zähler `curBlock` beinhaltet hierfür die Blocknummer des nächsten erwarteten Blocks. In `minBlock` wird jeweils das Minimum der bekannten, noch nicht ausgelesenen Blocknummern vorgehalten. Bei der Suche des nächsten Extent-Eintrags wird idealerweise ein Eintrag gefunden, dessen umspannte Blöcke bei `curBlock` beginnen (Zeile 13). Wird für ein Blatt (Zeile 15) ein solcher Eintrag gefunden, kann die Datei mit Inhaltsdaten befüllt werden, sonst steigt die Rekursion in Zeile 25 ab.

Diese Suche begründet auch die äußere Schleife in Zeile 11: Für jeden Eintrag müssen alle weiteren Einträge als potenzielle Nachfolger in Erwägung gezogen werden. Die Grenze `entries * 2` kommt dadurch zustande, dass nicht immer ein Extent-Eintrag existieren muss, dessen erster Block `curBlock` ist. Die Abfrage `entriesDone == entries` in Zeile 43 wird verwendet, um die Eintragssuche abzubrechen, wenn für jeden erwarteten Eintrag bereits ein Eintrag behandelt wurde, wodurch die Laufzeit im Idealfall linear statt quadratisch mit der Anzahl der Extent-Einträge skaliert.

Fehlende Speicherbereiche kommen in Ext4-Dateisystemen dadurch zustande, dass längere Nullfolgen innerhalb einer Datei implizit durch das Fehlen von Blöcken in Extents dargestellt werden. Durch die Bedingung `minBlock > curBlock` (Zeile 50) kann auf eine solche Nullfolge geschlossen werden. Mit `minBlockSet` wird verhindert, dass das erneute Setzen von `minBlock` auf `INT_MAX`, nachdem ein mögliches Minimum gefunden wurde, als fehlender Speicherbereich in der Extent-Struktur erkannt wird. Dieser entstandene Bereich zwischen `curBlock` und `minBlock` wird in der Ausgabedatei mit Nullblöcken aufgefüllt und `minBlock` anschließend wieder auf sein Maximum gesetzt, wie es in den Zeilen 53 bis 56 zu sehen ist.

In Abbildung 3.4 wird beispielhaft skizziert, wie fehlende Datenblöcke innerhalb einer Extent-Struktur in der Ausgabedatei zu interpretieren sind.

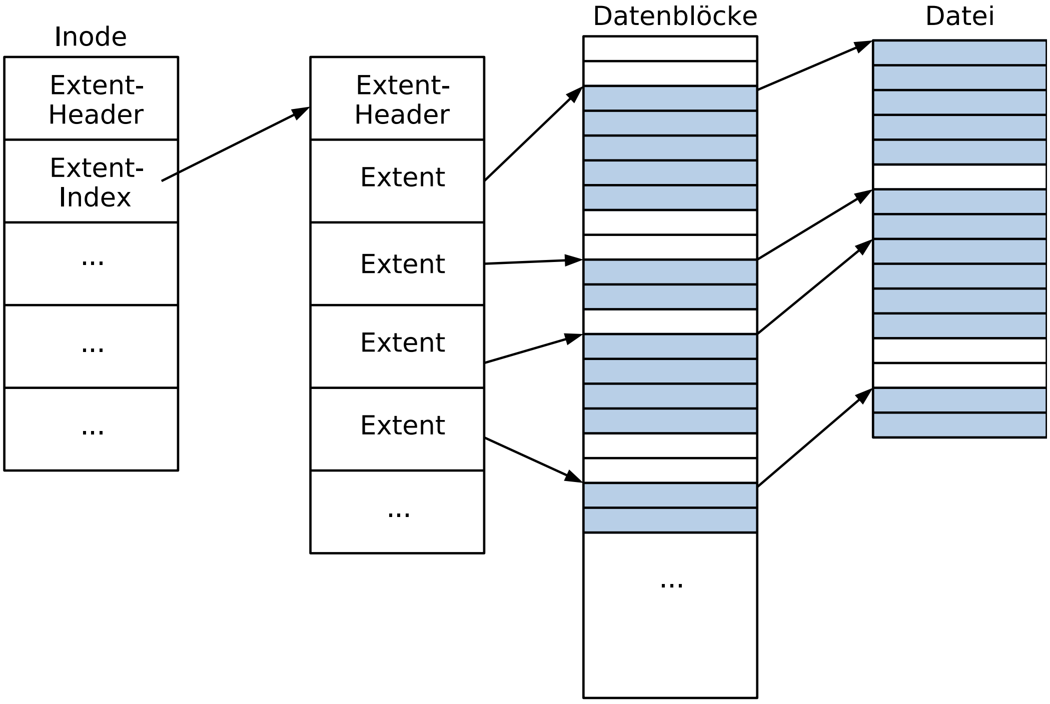


Abbildung 3.4: Extent-Struktur einer Datei, die leere Datenblöcke beinhaltet

Die Extents verweisen hierbei auf Datenblöcke des Datenträgers. Rechts im Bild ist zu sehen, wie die Abbildung der Datenblöcke auf die Datei mithilfe der logischen Blocknummern stattfindet. Wie bereits erklärt, werden Lücken in den logischen Blocknummern hierbei auf implizite, leere Datenblöcke (im Bild in weiß) abgebildet. Aufgrund der Tatsache, dass solche Lücken als Differenzen von Anfangsblocknummern der Extent-Einträge realisiert sind, kann eine Datei nicht mit einer impliziten Nullfolge enden.

Alternativ zur Rekursion, die die logische Lesereihenfolge der Datenblöcke sicherstellt, könnte in physischer Reihenfolge über die Extents iteriert werden. In diesem Fall müsste jeder Inhaltsdatenblock an seine entsprechende Stelle innerhalb der Ausgabedatei geschrieben werden; dies bedingt der vorherigen Erstellung einer leeren Datei entsprechender Größe. Bei dieser Erstellung würden unter Umständen sehr große Dateien alloziert und beschrieben werden; beim gewählten Algorithmus wird stets nur so viel Speicher belegt, wie bis zum jeweiligen Zeitpunkt sicher benötigt wird. Durch das direkte Abbrechen im Fehlerfall wird so nie Speicher alloziert, der wieder verworfen werden muss. Weiterhin ist es auf diese Weise nicht notwendig, erst die gesamte Datei mit Nullen zu beschreiben; dies sorgt bei vollständig beschriebenen Dateien insgesamt für die doppelte Anzahl an Schreiboperationen.

Da im Inhaltsdaten-Modus aufgrund fehlender Metadaten nur Dateiinhalte, anstelle von Dateien, rekonstruiert werden können, soll Redundanz vermieden werden. Durch die Wiederherstellung des Journals kann die selbe Datei von mehreren Inodes gleicher Nummer, jedoch unterschiedlicher Adressen im Dateisystem gefunden werden. Weil in diesem Modus derartige Duplikate eliminiert werden sollen, muss ein anderes Kriterium für die Gleichheit zweier Inodes formuliert werden, als Adressengleichheit.

In Listing 3.10 wird gezeigt, wie die Duplikate während der Rekonstruktion behandelt werden. Hierzu werden die 60 Byte, in der die Wurzel der Extent-Struktur im Inode steht, auf Gleichheit überprüft (Zeile 17). Gleichen sich zwei Inodes in diesen 60 Byte und geben zusätzlich dieselbe Dateigröße an (Zeile 18), würde eine Rekonstruktion das gleiche Ergebnis erzielen. Derartige Inodes werden als Duplikate verstanden.

```
 1  size_t lowerSize = (unsigned char)a_Reader.m_InodeBuffer[4]
 2                   | (unsigned char)a_Reader.m_InodeBuffer[5] << 8
 3                   | (unsigned char)a_Reader.m_InodeBuffer[6] << 16
 4                   | (unsigned char)a_Reader.m_InodeBuffer[7] << 24;
 5
 6  size_t upperSize = (unsigned char)a_Reader.m_InodeBuffer[108]
 7                   | (unsigned char)a_Reader.m_InodeBuffer[109] << 8
 8                   | (unsigned char)a_Reader.m_InodeBuffer[110] << 16
 9                   | (unsigned char)a_Reader.m_InodeBuffer[111] << 24;
10
11  size_t size = (size_t)lowerSize | (size_t)upperSize << 32;
12
13  std:pair<char[60], size_t> tmp;
14  memcpy(tmp.first, m_InodeBuffer+40, 60);
15
16  for (size_t run = 0; run < m_RecoveredInodes.size(); ++run)
17  {
18    if ((std::equal(tmp.first, tmp.first+60, m_RecoveredInodes[run].first) == true) &&
19        (size == m_RecoveredInodes[run].second))
20    {
21      return -6;
22    }
23  }
24  tmp.second = size;
25  m_RecoveredInodes.push_back(tmp);
```

Listing 3.10: Duplikatvergleich im Inhaltsdaten-Modus

Die Werte `lowerSize` und `upperSize` in den Zeilen 1 bis 9 stellen die zweigeteilte Speicherung der Größe der Datei dar, wie sie in Inodes im Ext4-Dateisystem realisiert wird. In `m_InodeBuffer` befindet sich der Inode, dessen Dateiinhalt rekonstruiert werden soll. Da die Extent-Struktur ab Byte 40 beginnt und 60 Byte lang ist, wird dieser Bereich in Zeile 14 herauskopiert und zwischengespeichert.

In der Schleife ab Zeile 16 wird über alle bisherigen, wiederhergestellten Inodes iteriert, um festzustellen, ob es sich beim betreffenden Inode um ein Duplikat handelt. Wenn dies der Fall ist, beendet sich die Funktion und der nächste Inode wird untersucht, denn die Rekonstruktion des Dateiinhalts eines Inodes findet nur dann statt, wenn sein Extent-Dateigröße-Paar nicht bereits in der Menge akzeptierter Extent-Dateigröße-Paare `m_RecoveredInodes` vorkommt.

3.2.5 Leerdaten-Phase

Nach der Rekonstruktion der regulären Dateien im Metadaten-Modus durch die Inhaltsdaten-Phase verbleiben in der Leerdaten-Phase Verzeichniseinträge, die von den Inodes der regulären Dateien nicht erfasst wurden. Folglich können die Dateinamen von Dateien anderer Typen rekonstruiert und zusätzlich leere Verzeichnisse nachträglich erstellt werden.

Hierfür wird, wie in Listing 3.11 zu sehen ist, über den Verzeichnisbaum iteriert und dessen Einträge überprüft. In der Struktur `m_DirectoryInodes` befinden sich alle Inodes der Verzeichnisse, die im Dateisystem gefunden wurden. Die Funktion `directoryTreeRec` wurde in Listing 3.6 dargestellt und ist dafür zuständig, aus einer Inodenummer den dazugehörigen Dateipfad der Datei zu rekonstruieren.

```
 1 for (auto it : m_InodeTree)
 2 {
 3   if (it.second.complete == false &&
 4     m_DirectoryInodes.find(it.first) != m_DirectoryInodes.end())
 5   {
 6     std::string name = directoryTreeRec(it.first, 0);
 7     std::string dir = "mkdir␣-p␣\"" + m_OutputFolder + name + "\"";
 8     if (system(dir.c_str()))
 9     {
10       ...
11     }
12   }
13   else if (it.second.complete == false &&
14            m_DirectoryInodes.find(it.first) == m_DirectoryInodes.end())
15   {
16     std::string n = directoryTreeRec(it.first, 0);
17 
18     size_t pos = n.find_last_of("/");
19     std::string dir = n.substr(0, pos);
20     std::string name = n.substr(pos+1);
21 
22     std::string com = "touch␣\"" + m_OutputFolder + dir +
23                       "/-nonRegular_" + name + "\"";
24     if (system(com.c_str()))
25     {
26       ...
27     }
28   }
29 }
```

Listing 3.11: Iterieren über den Verzeichnisbaum um Einträge zu rekonstruieren, die nicht von der Inhaltsdatenphase erfasst wurden

In der Bedingung in Zeile 3 und 4 wird überprüft, ob für einen Eintrag im Verzeichnisbaum bereits eine Datei existiert. Wenn dies nicht der Fall ist und sich dieser Eintrag unter den gefundenen Verzeichnissen befindet, so wird für dieses Verzeichnis die Inodenummer bestimmt und anschließend ihr Verzeichnis erstellt (Zeile 6 bis 11). Der Befehl `mkdir -p` in Zeile 7 erstellt ein Verzeichnis und zusätzlich alle dafür notwendigen Elternverzeichnisse. Da in diesem Schritt auch für alle Teilverzeichnisse die Pfade gesetzt werden, und somit ihr `complete`-Wert aktualisiert wird, wird somit auch sichergestellt, dass diese Verzeichnisse bereits existieren.

Durch die Bedingung in Zeile 13 und 14 wird der Fall abgefangen, in dem der Eintrag im Verzeichnisbaum noch nicht bearbeitet wurde und dieser kein Verzeichnis ist. Auf diese Weise werden alle Dateitypen behandelt, die nicht rekonstruiert wurden, wie zum Beispiel symbolische Verknüpfungen und Gerätedateien. In dieser Phase erzeugte Dateien werden auf Basis ihrer Verzeichniseinträge, nicht ihrer Inodes, erstellt, wodurch kein Dateiinhalt für sie zur Verfügung steht. Dadurch werden auch Verzeichniseinträge von Dateien behandelt, deren Inodes von den Suchmustern aussortiert wurden. Dazu gehören zum Beispiel reguläre Dateien, die andere Zugriffrechte als die vom Nutzer angegebenen besitzen. Für diese Einträge werden leere Dateien erstellt (Zeile 16 bis 27), da auf Inhaltsdaten nicht zugegriffen werden können.

In den Dateinamen wird bei der Rekonstruktion die Markierung `-nonRegular_` (Zeile 23) eingefügt, damit diese Dateien nicht mit regulären, leeren Dateien verwechselt werden können. Dies ist vor allem wichtig, da im Metadaten-Modus leere Dateien als reguläre Dateien zugelassen und somit rekonstruiert werden. Im Inhaltsdaten-Modus wird diese Phase nicht aufgerufen, wodurch die Aussortierung leerer Dateien diesen Modus nicht betreffen.

Da diese Phase entkoppelt von der Inhaltsdaten-Phase ist und zu einem späteren Zeitpunkt ausgeführt wird, können keine regulären Dateien rekonstruiert werden, die in der Inhaltsdaten-Phase betrachtet wurden. Für ihre Einträge im internen Verzeichnisbaum gilt aufgrund der Rekonstruktion bereits `complete == true`. Schlägt die Rekonstruktion einer regulären Datei jedoch fehl, ist der entsprechende Eintrag im internen Verzeichnisbaum dennoch mit einem vollständigen Pfad versehen. Deshalb werden Dateien dieser Art von der Leerdaten-Phase nicht erfasst.

3.3 Verwendung

Für die Benutzung des Moduls innerhalb des Sleuthkit-Frameworks müssen verschiedene Einstellungen getroffen werden. In der `pipeline_config.xml`, wie sie schon in Kapitel 2 vorgestellt wurde, muss das Modul an der gewünschten Stelle innerhalb der Pipeline eingefügt werden. Da das Programm als Post-Processing-Modul entwickelt wurde, gestaltet sich die entsprechende Konfigurationsdatei wie in Listing 3.12.

```
<PIPELINE_CONFIG>
  <PIPELINE type="FileAnalysis">
    ...
  </PIPELINE>
  <PIPELINE type="PostProcessing">
    ...
    <MODULE order="1" type="plugin" location="tskInodeModule"
            arguments="#MODULE_CONFIG_DIR#/InodeModule_arguments.txt"/>
    ...
  </PIPELINE>
</PIPELINE_CONFIG>
```

Listing 3.12: Konfigurationsdatei für die Framework-Pipeline

In diesem Fall wurde das Modul als erstes in der Nachbearbeitungsphase ausgewählt; dies ist durch das Attribut `order="1"` gekennzeichnet. Das Modul kann ein Plugin oder eine ausführbare Datei sein, was durch das Attribut `type` angegeben wird. Da das Modul mit dem Sleuthkit interne Variablen, Pfade zum zu untersuchenden Datenträger und zum Zielordner austauschen und somit mit ihm kommunizieren soll, wurde sich für ein Plugin entschieden. Als `arguments` wird der Pfad zur Konfigurationsdatei übergeben.

Die Entscheidung, das Modul in das Post-Processing einzugliedern, liegt darin begründet, dass der vorgestellte Ansatz streng genommen ein File-Carving-Ansatz ist, das Sleuthkit-Framework allerdings keine Module in die Extraktionsphase eingliedern lässt. Gegen die Phase der Dateianalyse wurde sich entschieden, da diese Module auf Daten innerhalb der bereitgestellten Datenbank arbeiten und diese aktualisieren. Vom vorgestellten Modul wird dies nicht vorgenommen, deswegen bleibt einzig die Nachbearbeitungsphase übrig. Da alle nicht erwünschten Module aus der Pipeline entfernt werden können, können eventuelle Modulkonflikte in jedem Fall umgangen werden.

Zusätzlich soll in dem vorgestellten Modul auf dem Datenträger selbst gearbeitet werden, selbst wenn dieses korrupt oder beschädigt ist. Um dies unter allen Umständen zu gewährleisten, wurde eine eigene Klasse zum Auslesen des Dateisystems implementiert, anstatt auf die bereitgestellte Methode des Sleuthkit-Frameworks zurück zu greifen.

Entwickler-Module, die in das Sleuthkit-Framework eingebunden werden sollen, müssen die in Listing 3.13 dargestellte Schnittstelle implementieren.

Die Funktion `initialize(const char* arguments)` (Zeile 1) behandelt die Initialisierung des Moduls und wird zu Beginn aufgerufen. Übergeben bekommt diese Funktion eine Zeichenkette, in der die Argumente

```
 1  TskModule::Status TSK_MODULE_EXPORT initialize(const char* arguments)
 2  {
 3    ...
 4  }
 5
 6  TskModule::Status TSK_MODULE_EXPORT report()
 7  {
 8    ...
 9  }
10
11  TskModule::Status TSK_MODULE_EXPORT finalize()
12  {
13    ...
14  }
```

Listing 3.13: Benötigte Funktionen für ein Post-Processing-Modul

aus der `pipeline_config.xml`-Datei übergeben werden, in diesem Fall den Pfad zur Konfigurationsdatei des Moduls. Die `report()`-Funktion in Zeile 6 ist die Hauptfunktion des Post-Processing-Moduls. Falls nach der Ausführung noch Speicherbereiche freigegeben, Seiteneffekte korrigiert oder Prozesse beendet werden müssen, findet dies laut Framework-Spezifikation in der `finalize()`-Funktion (Zeile 11) statt. Diese Funktion wird als letztes, vor der Beendigung des Programms, ausgeführt.

Falls ein Modul für die Phase der Dateianalyse entwickelt wird, so wird die `report()`-Funktion nicht benötigt. Hierfür ist die `run()`-Funktion vorgesehen.

Um das Modul ausführen zu können, muss das Sleuthkit installiert sein, und das Framework mit einem Kompilierer übersetzt werden. Relativ zum Installationspfad des Sleuthkits befindet sich das Framework in `sleuthkit-X.X/framework/`. Durch das Übersetzen wird eine ausführbare Datei erstellt, die sich in `sleuthkit-4.1.3/framework/tools/` befindet und `tsk_analyzeimg` heißt. Dieser Binärdatei können verschiedene Argumente übergeben werden, die innerhalb des Frameworks von den Modulen abgerufen werden können, wie etwa der Dateipfad des zu untersuchenden Datenträgerabbilds. Durch die Ausführung dieser Datei wird die Pipeline gestartet und somit die Module der Reihe nach aufgerufen.

Mittels der Konfigurationsdatei für das vorgestellte forensische Werkzeug stehen weitere Optionen zur direkten Einflussnahme auf seine Wirkungsweise zur Verfügung. Wie diese aussehen kann, wird beispielhaft in Listing 3.14 gezeigt.

Das '-' zu Beginn bedeutet, dass diese Option auskommentiert ist und somit nicht vom Modul beachtet und eingelesen wird. Das Zeichen'$' markiert den Beginn eines Parameternamens. Hierbei sind alle Parameter optional. Für eine Untersuchung muss keiner der aufgelisteten Parameter gesetzt werden, jedoch müssen für eine korrekte Rekonstruktion der Dateien auf dem Dateisystem Werte übergeben werden, wenn diese für das zu untersuchende Dateisystem von den Standardwerten abweichen.

Durch den Pfad, der in `OutputFolder` spezifiziert wird, kann ein alternativer Pfad für die wiederhergestellten Zieldateien bereitgestellt werden. Steht kein vom Nutzer spezifizierter Parameter zur Verfügung, so wird der standardmäßige Ausgabepfad des Sleuthkit-Frameworks für alle Module genutzt. Dieser befindet sich meist im selben Verzeichnis, wie auch das Datenträgerabbild. Mittels `ImageSize` und `ImgOffset` kann eine Partition innerhalb eines größeren Datenträgerabbilds angegeben werden. Im Bereich, der durch die beiden Werte als obere und untere Schranke aufgespannt wird, wird das Datenträgerabbild nach Inodes untersucht. Dabei müssen die Werte in Byte angegeben werden. Wenn keine Werte angegeben wurden, wird als Offset 0 angenommen und als Größe des Dateisystems wird die gesamte Größe des Datenträgerabbilds benutzt.

Die nächsten beiden Werte, `BlockSize` und `InodeSize` müssen ebenfalls in Byte angegeben werden. Falls diese an dieser Stelle nicht spezifiziert werden, werden Standardwerte, wie sie vom mkfs-Programm gesetzt werden, angenommen, welche von der Größe des Dateisystems abhängen. Für die Werte `FlexGroupSize` und `InodeRatio` werden ebenfalls Standardwerte des mkfs-Programms benutzt, wenn an dieser Stelle nichts anderes spezifiziert wurde. Durch diese beiden Werte wird der Speicherort und die Größe der Inode-Tabelle beeinflusst, wodurch sich dies ebenfalls auf die Berechnung der Inodenummern für reguläre Dateien und Verzeichnisse im Metadaten-Modus bezieht.

```
-$OutputFolder              /home/user/output/
$ImageSize                  120000000
$ImgOffset                  0
$BlockSize                  4096
$InodeSize                  256
-$FlexGroupSize             16
-$InodeRatio                16384
-$64BitEnable               false
-$SparseSuperblock          true
$TimeStampMin               1420070400
$TimeStampMax               1451606400
$Permission                 0x124
-$Permission                0x1b4
-$Permission                0x180
-$Permission                0x1e8
-$Permission                0x1f8
-$Permission                420
$Permission                 493
$Permission                 0775
-$Permission                0700
```

Listing 3.14: Konfigurationsdatei für das Inode-Carving-Modul

Der Wert `64BitEnable` gibt an, ob das Dateisystem den 64-Bit-Modus verwendet. Dies hat Auswirkungen auf die Größe des adressierbaren Speicherbereichs wie auch auf die Größe der Gruppen-Deskriptoren, wodurch der Offset zu den Inode-Tabellen beeinflusst wird. Standardmäßig wird der 32-Bit-Modus beim Ext4-Dateisystem benutzt, wodurch dieser Wert intern mit `false` initialisiert wird. Im Gegensatz dazu ist die `SparseSuperblock`-Option auf Ext4-Dateisystemen standardmäßig aktiviert. Durch diese Option werden Superblock und Gruppen-Deskriptor-Tabelle nicht in jeder Blockgruppe redundant abgespeichert. Auch dieser Wert beeinflusst die Berechnungen für den Offset der Inode-Tabelle.

Durch `TimeStampMin` und `TimeStampMax` kann ein weiteres Kriterium für die Suchmuster in der Inode-Carving-Phase angegeben werden. Alle potentiellen Inodes, werden nur dann für die Rekonstruktion in Betracht gezogen, wenn alle definierten Zeitstempel innerhalb dieses Zeitintervalls liegen. Falls nur die Konsistenz der Zeitstempel überprüft werden soll, ohne eine Einschränkung in ein Intervall zu haben, kann `TimeStampMin` einkommentiert und auf 0 gesetzt werden; dementsprechend kann `TimeStampMax` auf `INT_MAX` gesetzt werden, um diese Funktionalität zu erhalten.

Der einzige Parameter in der Konfigurationsdatei, der häufiger als einmal auftreten darf, ist `Permission`. Mit diesen Feldern können Zugriffsrechte spezifiziert werden, die im Laufe der Inode-Carving-Phase als Suchmuster verwendet werden können. An dieser Stelle gibt es verschiedene Möglichkeiten, wie die Zugriffsrechte übergeben werden können. Beispielsweise das Muster `r--r--r--` bildet sich binär auf `100100100` ab. Durch diese Darstellung ist es möglich, diesen Parameter als Ganzzahl zu verschiedenen Basen anzugeben. Mit einem führenden `0x` kann somit eine hexadezimale Zahl angegeben werden, wobei eine führende 0 eine oktale Zahl kennzeichnet. Sind keine sonstigen führenden Ziffern gegeben, so wird der Wert als dezimale Zahl interpretiert.

Dementsprechend kann `r--r--r--` auch hexadezimal als `0x124` dargestellt werden. Ein Beispiel für ein oktal definiertes Zugriffsrecht lautet 0775 und entspricht `rwxrwxr-x`. Die dezimale Zahl 420 dagegen repräsentiert das Zugriffsrecht `rw-r--r--`. Die oktale Notierung erlaubt es dem Nutzer, die Zugriffsrechte im gleichen Format anzugeben, wie im verbreiteten Kommandozeilenprogramm `chmod`, das Änderungen an Zugriffsrechten einer Datei oder eines Verzeichnisses vornehmen kann.

Diese Werte werden als vorzeichenlose Ganzzahlen in einem `std:vector<unsigned short>` gespeichert. In der Inode-Carving-Phase werden Zugriffsrechte potentieller Inodes mit diesen verglichen. Sobald der Inode einem dieser Muster entspricht, wird dieser als korrekt angenommen vom Suchmuster akzeptiert. Jedoch wird der Fehlerfall abgefangen, in dem ungültige Zugriffsrechte übergeben werden. Dabei entspricht die Untergrenze 0 und die Obergrenze für ein gültiges Zugriffsrecht dezimal 511, beziehungsweise 0777 oder `0x1ff`, je nach Basis.

Die Mehrfachnennung von Parametern abgesehen von Zugriffsrechten bewirkt keinen Abbruch des Programms. In einem solchen Fall wird die in der Datei letzte Parameterspezifikation als gültig angenommen. Es wird dennoch empfohlen, keine Mehrfachnennungen anzuwenden, sondern alternative Testparameter durch die Kommentarfunktion in der Datei zu notieren.

4

EVALUATION

Im vorigen Kapitel wurde ein forensisches Werkzeug vorgestellt, mit dem Dateien rekonstruiert werden können, indem mittels verschiedener Suchmuster Inodes in einem Ext4-Dateisystem gesucht werden und die dazugehörigen Dateien anschließend mittels Metadatenanalyse rekonstruiert werden. Dabei wurde unter Anderem auf die Konfigurationsmöglichkeiten der Suchmuster und die verschiedenen Modi zur Rekonstruktion der Dateien eingegangen.

Die folgenden Suchmuster werden vom Programm unterstützt:

- Zugriffsrechte
- Zeitstempel
- Anzahl der Hard Links
- Extent-Flag und Extent-Header
- Dateityp des Inodes

Alle Inodes, die diesen Suchmustern genügen, gelten als potentielle Inodes und werden anschließend anhand ihrer Extent-Baum-Struktur rekonstruiert. Hierfür bietet das Programm zwei verschiedene Modi an: Den Inhaltsdaten- und Metadaten-Modus. Im Inhaltsdaten-Modus werden ausschließlich Dateiinhalte rekonstruiert, unabhängig davon, ob der Inode in einer gültigen Inode-Tabelle oder in einem anderen Datenbereich des Dateisystems aufgefunden wurde. Der Metadaten-Modus bietet die Möglichkeit, die Verzeichnisstruktur, wie sie auf dem Dateisystem vorzufinden war, mittels Verzeichniseinträgen zu rekonstruieren.

In diesem Kapitel wird eingangs der Datensatz vorgestellt, mit dem das Werkzeug evaluiert wurde. Es wurde besonderes Augenmerk darauf gelegt, verschiedene Testfälle zu generieren, die sowohl Real- als auch Spezialfälle überprüfen.

Anschließend wird mittels der vorgestellten Testfälle die Korrektheit überprüft. Anhand eines Realfalles, auf dem ein vollwertiges Betriebssystem installiert wurde, wird die Korrektheit und Selektionsrate der Suchmuster ermittelt. Dabei werden die Effektivität und die Einschränkungen der verschiedenen Suchmuster einzeln untersucht und schließlich eine Kombination von mehreren getestet. Es spielt ebenfalls die Anzahl der potentiellen Inodes eine Rolle, die zu keinen rekonstruierbaren Dateien führen, sogenannte *false positives*.

Ebenso wird die Korrektheit mittels der Vollständigkeit der rekonstruierten Daten aufgezeigt. Zu diesem Zweck wurde eine Auswahl an Suchmustern getroffen, die für alle weiteren Testfälle gilt. Bei der Rekonstruktion werden anhand der gefundenen Inodes die dazugehörigen Dateien, bzw. Dateiinhalte, mittels beider vorhandenen Modi rekonstruiert und ihre Ergebnisse überprüft und miteinander verglichen. Mittels Prüfsummenberechnungen durch MD5 und SHA256 wurde sichergestellt, dass die Dateien vom Werkzeug korrekt rekonstruiert wurden. Ist dies nicht der Fall, wird von einer inkorrekten oder unvollständigen Rekonstruktion gesprochen. Um auch das Anwendungsgebiet des forensischen Werkzeugs zu ermitteln, wurden sowohl Realfälle als auch Spezialfälle getestet.

Schließlich wurde für einige Testfälle die Anwendbarkeit getestet, indem die Laufzeit des Programms für die Suchmuster und die Rekonstruktion gemessen wurde. Im Zuge dessen wurde der Einfluss auf die Laufzeit durch die Größe des Dateisystems und den zu rekonstruierenden Dateien evaluiert.

4.1 Datensatz

Um das forensische Werkzeug im Folgenden unter realistischen Bedingungen zu evaluieren, wurden verschiedene Datenträgerabbilder erstellt. Diese umfassen verschiedene Dateisystem-Größen, damit sie verschiedene Standardfälle beim Formatieren mit dem mkfs-Programm abdecken. Ebenso wurden verschiedene Testfälle, wie etwa das Löschen bestimmter Dateien oder die Einstellung der Parameter, welche einen Einfluss auf die Funktionsweise des vorgestellten forensischen Werkzeugs nehmen sollen, formuliert. In Tabelle 4.1 werden die Datenträgerabbilder für diese Testfälle detailliert aufgezeigt.

Datenträgername	Größe	mkfs-Typ	Beschreibung
floppy.img	2,3 MB	floppy	
small.img	230 MB	small	Beispiele von Dateisammlungen
default.img	12 GB	default	
floppy_deleted.img	2,3 MB	floppy	Dateien vom Image gelöscht
floppy_deletedTrash.img	2,3 MB	floppy	Dateien vom Image mit Papierkorb gelöscht
floppy_newFiles.img	2,3 MB	floppy	
small_newFiles.img	230 MB	small	Gelöschte Dateien durch Andere ersetzt
default_newFiles.img	12 GB	default	
default_sameFiles.img	12 GB	default	
default_withoutJournal.img	6 GB	default	Journal deaktiviert; Dateien gelöscht
Ubuntu_Pics_complete.img	16,0 GB	default	Ubuntu 12.04
small_newExt4.img	230 MB	small	Überformatiert mit gleichen Ext4-Parametern
small_diffExt4.img	230 MB	small	Überformatiert mit anderen Ext4-Parametern
small_fastExt4.img	230 MB	small	Schnell überformatiert mit gleichen Parametern
small_fastdiffExt4.img	230 MB	small	Schnell überformatiert mit anderen Parametern
small_NTFS.img	230 MB	small	Überformatiert mit NTFS
small_fastNTFS.img	230 MB	small	Schnell überformatiert mit NTFS

Tabelle 4.1: Vollständiger Datensatz für die Evaluation

Die ersten drei Datenträgerabbilder der Tabelle beschreiben verschiedene Beispiele für Sammlungen von Dateien, genauer Bild- und Text-Dateien. Zweck dieser drei unterschiedlich großen Fälle ist die Überprüfung der Parameterschätzung. Es wurden keine vom mkfs-Standard abweichenden Ext4-Parameter gewählt, so dass das Programm den mkfs-Typ und damit alle relevanten Parameter des Dateisystems allein anhand der Größe des Datenträgerabbilds schätzen kann.

Die nächsten beiden Abbilder sollen Fälle darstellen, in denen von einem Datenträger eine Sammlung von Dateien gelöscht wurde. Dabei wurde beachtet, dass die Dateien beim Löschvorgang unter Betriebssystemen, in diesem Fall dem Linux-basierten Ubuntu, in den sogenannten Papierkorb verschoben werden, bevor diese vollständig vom Datenträger gelöscht werden. Dadurch ergeben sich zwei verschiedene Testfälle: Das herkömmliche Löschen (`floppy_deleted.img`) und das Löschen mit anschließender Leerung des Papierkorbs (`floppy_deletedTrash.img`). Mit letzterem Testfall soll analysiert werden, ob gelöschte Dateien, deren Metadatenstrukturen in den Inode-Tabellen nicht mehr zu ihrer Wiederherstellung ausreichen, durch Einträge im Journal gefunden werden können.

Von `floppy_deletedTrash.img` ausgehend, wurde ein Fall generiert, in dem der nun aus Nutzersicht leere Datenträger mit anderen Dateien beschrieben wird. Zum Überschreiben der Dateien wurden gänzlich andere Dateien anderer Formate und Bedeutungen gewählt, um die beiden Zustände unterscheiden zu können. Hierbei wurden auch größere Dateien verwendet, um sicherzustellen, dass das Auslesen nicht flacher Extent-Bäumen für gültige Tiefenwerte korrekt funktioniert. Zweck dieses Tests ist die Analyse darüber, ob durch das Journal auch (teils) überschriebene Dateien gefunden werden können. Diesen Fall stellen `floppy_newFiles.img`, `small_newFiles.img` und `default_newFiles.img` dar; alle Datenträgerabbilder in dieser Kategorie wurden analog erstellt.

Der Testfall `default_sameFiles.img` stellt einen Testfall dar, der wie `default_newFiles.img` erstellt wurde. Jedoch wurden auf dieses Image die selben Dateien, wie auf `small_newFiles.img` verwendet, wodurch nur ein kleiner Teil des verfügbaren Speichers mit Daten überschrieben wurde. Dadurch kann in Abschnitt 4.3 ein Vergleich zwischen den Laufzeiten beider Datenträgerabbilder durchgeführt werden.

Da sich die eben genannten Testfälle allein darauf beziehen, dass aus dem Journal Dateien wiederhergestellt werden können, soll mit `default_withoutJournal.img` ein Kontrolltest zur Verfügung gestellt werden. Bei diesem Testfall wurde das Journal bei der Erstellung des Dateisystems explizit deaktiviert und alle Dateien vom Datenträgerabbild gelöscht. Dadurch sollten keinerlei interpretierbaren Metadaten der gelöschten Dateien auf dem Datenträger zu finden sein. Die Wahl der Dateisystemgröße von 6 GB soll sicherstellen, dass das Fehlen wiederhergestellter Dateien auf die eben gestellte Vermutung zurückzuführen ist und nicht auf Zufall basiert.

Zur Bereitstellung eines möglichst realitätsnahen Testfalles wurde auf `Ubuntu_Pics_complete.img` ein vollwertiges Ubuntu-Betriebssystem auf einem USB-Stick installiert. Dieses wurde in Betrieb genommen und trat in den Datenaustausch mit anderen Partitionen außerhalb des eigenen Datenträgers. Es wurde als Dateisystemgröße bewusst kein ganzzahlig Vielfaches von 1024 Byte gewählt, damit Verschnitte von Blöcken und Blockgruppen auftreten. Da bei der Installation des Betriebssystems eine Vielzahl von Dateien verschoben, gelöscht und verändert werden, und durch die Notwendigkeit von symbolischen Links, Gerätedateien und weiteren, nicht regulären Dateitypen, wird durch diesen Testfall ein breites Spektrum von Sonderfällen für das entwickelte forensische Werkzeug abgedeckt. Zweck der Betrachtung dieser Sonderfälle ist, zu überprüfen, wie das vorgestellte Werkzeug diese behandelt.

Schließlich wurde ein Datensatz generiert, in dem ein bestehendes, mit Dateien gefülltes Ext4-Dateisystem mit einem neuen Dateisystem überschrieben wurde. Hierfür wurde `small.img` als Ausgangsfall genutzt und mit Konfigurationen der Ext4- und NTFS-Dateisysteme neu formatiert. Im Fall von NTFS wurden stets Standardparameter gewählt; für Ext4 wurden sowohl Testfälle für das Belassen der Parameter, als auch für deren Veränderung beim Neuformatieren generiert. Dabei wurde die Blockgröße auf 4096 Byte und die Anzahl der Inodes pro Blockgruppe auf 128 gesetzt. Im Standardfall ist die Blockgröße bei einem `small`-Datenträgerabbild 1024 Byte und Anzahl der Inodes pro Blockgruppe auf 8192. Für jeden der genannten Fälle wurde sowohl die Standardformatierung, als auch der schnelle Formatiermodus, bei dem die Datenblöcke nicht mit Nullen überschrieben werden, überprüft. Durch diese Testfälle ist es möglich, zu überprüfen, ob trotz der Formatierung mit einem neuen Dateisystem, die Daten aus dem ursprünglich darunter liegenden Dateisystem gefunden werden können.

4.2 Korrektheit

Mit den beschriebenen Testfällen soll die Korrektheit des forensischen Werkzeugs überprüft werden. Hierbei gilt es, sowohl die Selektionsrate der Suchmuster der File-Carving-Methode, als auch die Vollständigkeit der Rekonstruktion zu betrachten.

In diesem Abschnitt wird zunächst die Selektionsrate der einzelnen Suchmuster evaluiert und diese miteinander verglichen. Diese Untersuchung fand auf dem `Ubuntu_Pics_complete.img`-Datenträgerabbild statt, um ihre Effektivität bei einer großen Auswahl an Inodes verschiedener Dateitypen messen zu können. Die Selektivität entspricht hierbei dem Verhältnis zwischen der Anzahl akzeptierter potentieller Inodes zur Anzahl möglicher Inodes. Je geringer dieser Wert ist, desto restriktiver ist das Suchmuster. Im Zuge der Evaluation der Suchmuster wird auf alle Suchmuster eingegangen, die das forensische Werkzeug zur Verfügung stellt. Im Falle der Zeitstempel und Zugriffsrechte wird eine vordefinierte Auswahl getroffen.

Anschließend wurde sich auf eine Menge von Suchmustern festgelegt, um die Vollständigkeit zu überprüfen. Dabei lautete das Kriterium, dass alle rekonstruierbaren Dateien vom Werkzeug gefunden und korrekt wiederhergestellt werden müssen. Für alle getesteten Fälle wurde ein Vergleich zwischen dem Inhaltsdaten- und Metadaten-Modus anhand der Ergebnisse gezogen. Es wurden sowohl Realfälle – Datenträgerabbilder, die mit gewöhnlichem Inhalt beschrieben wurden – als auch Spezialfälle – Datenträgerabbilder mit überformatiertem Dateisystem – getestet.

4.2.1 Selektionsrate

Anhand des `Ubuntu_Pics_complete.img`-Datenträgerabbilds wurde die Selektionsrate der einzelnen Suchmuster getestet. Da dort eine Vielzahl verschiedener Dateien und damit unterschiedlicher Inodes zu finden ist, bietet dieser Realfall gute Bedingungen, um die Selektion nach Suchmustern zu evaluieren. Dabei befinden sich verschiedene Dateitypen auf dem Image, wie sie in Tabelle 4.2 aufgeschlüsselt sind.

Anzahl	Beschreibung
201.279	Anzahl der reservierten Inodes
201.269	Anzahl der Dateien auf dem Dateisystem
15.760	Anzahl der Verzeichnisse
127.159	Anzahl der regulären Dateien
58.350	Anzahl der Inodes anderer Dateitypen

Tabelle 4.2: Dateitypen auf `Ubuntu_Pics_complete.img`

Mittels des Sleuthkit-Werkzeugs `fsstat` wurde die Anzahl der reservierten Inodes auf dem Ext4-Dateisystem ermittelt. Werden mehr Dateien gefunden, müssen sich unter den Ergebnissen false positives befinden. Wenn nun beachtet wird, dass die ersten 10 Inodes auf einem Ext4-Dateisystem für spezielle Anwendungen reserviert werden, so bleiben 201.269 Dateien für das Dateisystem übrig. Von diesen wurde die Anzahl an Verzeichnissen und regulären Dateien ermittelt. Die übrigen Dateien haben einen Dateityp, der für die Rekonstruktion mit dem in dieser Arbeit vorgestellten Werkzeug nicht rekonstruiert werden können, wie zum Beispiel symbolische Links oder Gerätedateien.

Im Folgenden werden die verschiedenen getesteten Suchmuster evaluiert. Tabelle 4.3 stellt die verschiedenen Funde gegenüber. Dabei wurde das Datenträgerabbild byteweise untersucht, wodurch Suchmustervergleiche 16.011.755.520 mal durchgeführt wurden. Gelöschte Dateien wurden bei der Untersuchung als false positives betrachtet, da deren Inhaltsdaten nicht rekonstruiert werden können.

Muster	Treffer	Tabellenfehler	Adressfehler	Selektivität
Zugriffsrechte	150.927	1.022.673	99.861.246	0,6%
Zeitstempel Intervall	72.310	206.792	237.509	0,003%
Zeitstempel Konsistenz	209.481	5.708.816	1.416.653.929	8,89%
Anzahl Links	201.271	29.777.014	7.590.949.677	47,6%
Extent-Flag	165.830	13.431.604	3.281.240.822	20,6%
Extent-Header	165.830	514.795	53.231	0,004%
Dateityp	150.989	4.507.359	904.728.740	5,7%

Tabelle 4.3: Suchmuster mit ihrer Treffergenauigkeit

Jedes Suchmuster wird für jede betrachtete, physikalische Adresse ausgewertet und kann potentielle Inodes entweder annehmen oder ablehnen. Angenommene Strukturen, die sich innerhalb von Inode-Tabellen an gültigen Adressen befinden und korrekt rekonstruiert werden können, werden in Tabelle 4.3 als Treffer bezeichnet. Hierbei ist die gewünschte Referenz die Anzahl der Dateien auf dem Dateisystem, wie sie in Tabelle 4.2 zu sehen ist. Strukturen, die nach dem Muster zwar ein gültiger Inode sein könnten, sich aber außerhalb von gültigen Inode-Tabellen befinden, werden Tabellenfehler genannt. Da diese Strukturen gültige Journal-Inodes sein können, können diese nicht allgemeingültig abgelehnt werden. Ähnlich verhält es sich mit den sogenannten Adressfehlern. Diese stellen von Suchmustern akzeptierte Adressen dar, die nicht ganzzahlig durch die Größe eines Inodes teilbar sind. Da sowohl die Inode-Tabellen, als auch das Journal blockweise organisiert sind, können gültige Inodes bei bekannten Ext4-Parametern nicht an derartigen Adressen auftauchen.

Um die Suche nach potentiellen Inodes im vorgestellten Programm unabhängig von den Dateisystemparametern zu halten, wird auch über Adressfehler keine allgemeingültige Aussage zur Ablehnung getroffen. Die Summe über Treffer, Tabellenfehler und Adressfehler stellt folglich die Gesamtmenge der von einem Muster angenommenen Inodes dar. Der prozentuale Anteil dieser Gesamt-Trefferzahl von allen möglichen Inode-Adressen wird als Selektivität bezeichnet. Im Folgenden werden die Funktionsweisen und Ergebnisse aller einzelnen Muster diskutiert.

Zugriffsrechte

Bei der Suche mittels Zugriffsrechtsmustern wird für potentielle Inode-Adressen überprüft, ob ihre Zugriffsrechte relevant sind. Hierfür kann in der Konfigurationsdatei spezifiziert werden, welche Zugriffsrechte für die Suchmuster gelten sollen. Die Selektivität dieses Musters hängt hierbei stark von der getroffenen Wahl ab. Für das Beispiel wurden die relevanten Zugriffsrechte wie in Tabelle 4.4 gewählt. Diese wurden in Linux-typischer Schreibweise mit den Symbolen `rwx` für Lese-, Schreib- und Änderungsrechte notiert.

Besitzer	Gruppe	Alle
r - -	r - -	r - -
rw -	r - -	r - -
rw -	rw -	r - -
rwx	r - x	r - x
rwx	rwx	r - x
rwx	- - -	- - -
rw -	- - -	- - -
rwx	r - x	- - -
rwx	rwx	- - -

Tabelle 4.4: Für das Musterexperiment als relevant gewählte Zugriffsrechte

Von den angenommenen, potentiellen Inodes (Treffer) ließen sich 8.072 in der Inhaltsdaten-Phase nicht rekonstruieren; ein Teil hiervon besteht aus gelöschten Dateien, da diese nicht anhand der Zugriffsrechte erkennbar sind. Damit wurden nur 142.855 gültige Inodes von 201.269 mit diesem Suchmuster gefunden. Darunter befanden sich fast alle gesuchten regulären Dateien und Verzeichnisse. Diese hohe Ausschlussquote der Zugriffsrechtsprüfung liegt darin begründet, dass im Realfall-Dateisystemabbild Gerätedateien, symbolische Verknüpfungen und bestimmte Systemdateien Zugriffsrechte besitzen, die durch die Auswahl nicht erfasst wurden. Hiermit werden folglich wiederherstellbare Dateien bewusst nicht in die Auswahl mit aufgenommen; es wird eine semantische Selektion getroffen.

In der Praxis muss daher hohes Augenmerk auf die Wahl der relevanten Zugriffsrechte gelegt werden, da beispielsweise die Namen symbolischer Verknüpfungen, die stets die Nutzerrechte `rwxrwxrwx` besitzen, vom vorgestellten Werkzeug wiederhergestellt werden können. Ist eine Vorauswahl jedoch beabsichtigt, kann die Zugriffsrechts-Einschränkung diese zu einem frühen Zeitpunkt in der Programmausführung vornehmen. Werden keine Einschränkungen über die relevanten Zugriffsrechte getroffen, beträgt die Selektivität 100%, da es keine ungültige 9-Bit-Permutation im betroffenen Inode-Speicherbereich gibt.

Zeitstempel

Die Inode-Suche mittels Zeistempelvergleichen beinhaltet zwei wesentliche Bestandteile. Einerseits kann die innere Konsistenz der Zeitstempel eines potentiellen Inodes überprüft werden, andererseits besteht die Möglichkeit einer Eingrenzung des betrachteten Zeitintervalls.

Für den vorgestellten Test wurde das Zeitintervall vom 01.01.2015 um 00:00:00 GMT bis zum 01.01.2016 um 00:00:00 GMT gewählt, wodurch beispielsweise Systemdateien, deren letzte Änderung im April 2012 liegt, aussortiert wurden. Unter den gefundenen, potentiellen Inodes befanden sich noch 7.140 false positives, wodurch in der in Tabelle 4.3 gezeigte Wert für die Treffer auf 65.170 sinkt. Somit ist die Betrachtung von Zeitstempelgrenzen, wie auch die von Zugriffsrechten, ein semantisches Kriterium und erlaubt ein Vorauswahl zu einem frühen Zeitpunkt im Programmablauf.

Die Analyse der inneren Konsistenz von Zeitstempeln ohne Einschränkung des Zeitintervalls wurde in einem separaten Experiment getestet. Dass die Anzahl der Treffer die der möglichen Dateien übersteigt, liegt darin begründet, dass die Suche nach Zeitstempeln allein noch keine gelöschten Dateien eliminiert; genauer waren 8.213 der gefundenen Treffer false positives. Vier Dateien, zwei reguläre Dateien und zwei symbolische Verknüpfungen, die sich auf dem Datenträgerabbild befanden, wurden von der Konsistenzanalyse aussortiert. Diese wurden genauer überprüft: Ihren Zeitstempeln nach zu urteilen, wurden sie zu einem späteren Zeitpunkt modifiziert als ihre Metadaten, was von der Konsistenzüberprüfung aussortiert wird. Jedoch wurden alle anderen 201.264 gültigen Inodes gefunden.

Anzahl der Hard Links

Bei der Betrachtung der Anzahl der Hard Links wird in einem potentiellen Inode ausgewertet, ob sein interner Referenzzähler für auf ihn verweisende Verzeichniseinträge einen Wert über 0 angibt. Dieses Muster eliminiert gelöschte Dateien, da bei eben diesen die Anzahl der Referenzen null ist.

Abzüglich der speziellen Inodes 7 und 8, die gefunden wurden, allerdings als false positives zählen, werden somit alle 201.269 Dateien vom Muster erfasst. Es ist allerdings zu betonen, dass dieses Anzahlmuster am wenigsten Restriktion über die potentiellen Inode-Adressen mit sich bringt, da der Bytewert `0x00` der Einzige ist, der ausgeschlossen wird, aber gleichzeitig sehr häufig auftritt. Sieht man von der Ausgrenzung gelöschter Dateien ab, kann dieses Muster als syntaktische Ausgrenzung verstanden werden.

Extent-Flag und -Header

Die Suchmuster, bei denen das Extent-Flag bzw. der Extent-Header eines potentiellen Inodes analysiert werden, wurden getrennt betrachtet, ähneln sich aber in ihrer Funktionsweise stark. Während die Selektion anhand des Flags selbst noch keine besondere Restriktion bewirkt, stellt die vollständige Header-Analyse einen der am stärksten restriktiven Ausgrenzungsmechanismen dar. Da sich das Flag auf ein einzelnes Bit beschränkt besitzt es einen erheblich geringeren Informationsgehalt, als die 2 Byte lange Magic Number des Extent-Headers.

Weiterhin ist zu betonen, dass das Extent-Header-Muster das einzige Suchmuster ist, das deutlich mehr Tabellenfehler als Adressfehler verursacht. Dies liegt darin begründet, dass nicht nur jede Wurzel eines Extent-Baums einen Header besitzt, sondern jeder seiner inneren Knoten, wodurch sich dieses Muster auch in Datenblöcken finden lässt, was zu einem Tabellenfehler führt.

Darüber hinaus fällt auf, dass sowohl Extent-Header, als auch -Flag die selbe Anzahl an Treffern produzieren. Dies erklärt sich dadurch, dass genau jene intakten Inodes Extents besitzen, die dies auch in ihrem Flag anzeigen. Symbolische Verknüpfungen besitzen oftmals keine Extents und werden daher von den beiden Mustern ausgeschlossen. Ebenso nicht erfasst werden die meisten Geräte- und sonstige, nicht reguläre Dateien. Dennoch wurden alle regulären Dateien und Verzeichnisse angenommen. Unter den gefundenen potentiellen Inodes befinden sich allerdings auch 8.074 false positives. Beim Löschen von Dateien werden zwar die Extent-Referenzen ausgenullt, jedoch bleiben das Extent-Flag und der Extent-Header davon unberührt, weshalb gelöschte Dateien bei Extent-basierten Mustern angenommen werden.

Dateityp

Die Einstellung, lediglich bestimmte Dateitypen in der Suche nach möglichen Inodes zuzulassen, wurde im Test so gewählt, dass ausschließlich reguläre Dateien und Verzeichnisse angenommen werden, weswegen nur eine geringe Anzahl von Treffern erzielt wird. Neben 8.068 false positives werden hiermit alle 142.919 Inodes gefunden, nach denen gesucht wird.

Es bleibt, zu betonen, dass die zur Angabe des Dateityps bereitgestellten 4 Bit zur Darstellung von 7 verschiedenen Dateitypen es zusätzlich erlauben, 9 ungültige Werte (0, 3, 5, 7, 9, 11, 13, 14, 15) auszugrenzen. Somit könnte auch ohne eine semantische Einschränkung auf Dateitypen, im Beispiel reguläre Dateien und Verzeichnisse, eine Restriktion durch Konsistenzprüfung erreicht werden. Bei einer eventuellen Erweiterung der betrachteten Dateitypen muss der Extent-Flag-Mechanismus (zusammen mit dem Extent-Header-Mechanismus) angepasst werden, da einige Dateitypen selten Extents besitzen, falls diese Muster in Kombination verwendet werden sollen.

Schlussfolgerungen

Bei der Betrachtung der Selektivitätsraten der verschiedenen Suchmuster liegt es nahe, eine geringe Anzahl hochselektiver Kriterien für die folgenden Experimente auszuwählen. Die restriktivsten drei wären etwa die Suchmuster nach Zeitstempel Intervallen, Extent-Headern und Zugriffsrechten. Jedes dieser Muster stellt jedoch semantische Anforderungen an die Suche nach Inodes, insbesondere die Angabe eines Zeitintervalls und einer Menge relevanter Zugriffsrechte.

Da im Folgenden die Korrektheit des Programms analysiert werden soll, sind derartige Einschränkungen zu vermeiden, da sonst fälschlicherweise nicht aufgefundene Dateien nicht von solchen zu unterscheiden wären, die von besagten Einschränkungen ausgegrenzt werden würden. Die einzige Einschränkung, die getroffen werden darf, lautet, dass gefundene Dateien durch das vorgestellte Werkzeug wiederherstellbar sein müssen. Hierunter fallen ausschließlich ungelöschte, reguläre Dateien und Verzeichnisse. Aufgrund dessen ist die Kombination aus der Anzahl der Hard Links, der Analyse des Extent-Flags und -Headers und des Dateityps gewählt worden.

In Abbildung 4.1 ist zu sehen, wie die sukzessive Anwendung der einzelnen Suchmuster sich auf die potentiellen Zwischenergebnisse auswirkt. Hierbei stellt die Ordinate logarithmisch die Anzahl der von einem Zwischenschritt akzeptierten Adressen dar. Die Ergebnisse wurden für alle Dateisystemabbilder eingetragen, die unter den default-mkfs-Typ fallen, da diese vom Datenumfang ähnlich sind. Für die anderen Datenträgerabbilder verhält sich der Verlauf der Selektion analog, weshalb er an dieser Stelle nicht gesondert abgebildet wurde.

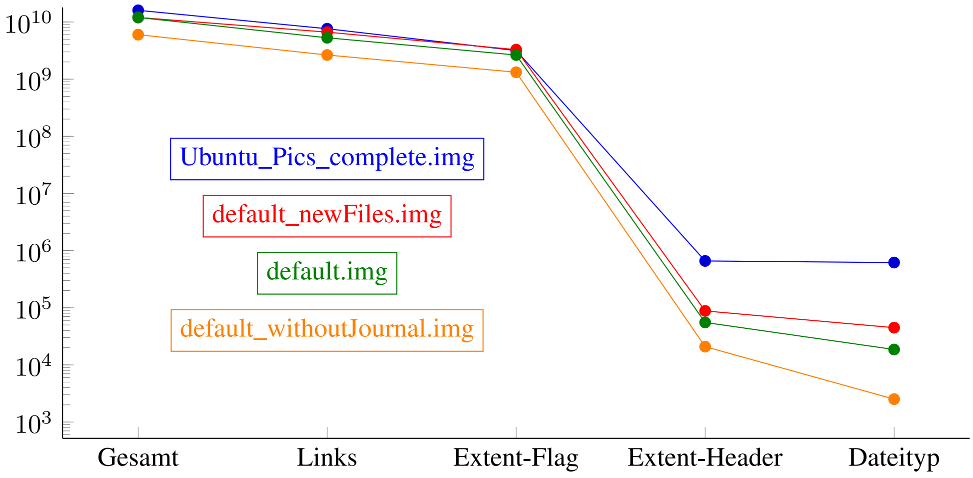


Abbildung 4.1: Sukzessive Reduktion des Suchraums durch die gewählten Vergleichsmuster

Die Abgrenzung durch Hard Links und Extent-Flags reduziert die potentiellen Ergebnisse jeweils um etwa die Hälfte. Dies legt zwar zunächst nahe, dass der Ausschluss von Hard Links keinen Selektivitätsgewinn bringt, jedoch werden auf diese Weise gelöschte Dateien, die in der Rekonstruktion nicht beachtet werden, herausgefiltert. Anschließend findet eine hohe Reduktion des Suchraums durch die Überprüfung der Magic Number der Extent-Header statt. Das Eliminieren irrelevanter Dateitypen sortiert schlussendlich lediglich einen geringen Anteil von Suchergebnissen aus, da es wenige reguläre Dateien und Verzeichnisse gibt, die keine Extents verwenden.

4.2.2 Vollständigkeit

Für die Evaluation der Korrektheit und Vollständigkeit der Ergebnisse durch das vorgestellte forensische Werkzeug, gibt es zwei relevante Messgrößen: Der Anteil gefundener Solldateien und die Anzahl von false positives.

Für die Rekonstruktion wurden beide Modi, die das Programm bietet, getestet und deren Ergebnisse miteinander verglichen. Alle wiederhergestellten Dateien wurden mittels MD5- und SHA256-Prüfsummen auf Korrektheit überprüft. Als korrekt wiederhergestellte Dateien gelten im Metadaten-Modus alle Dateien, die den richtigen Dateinamen und Dateipfad erhalten haben, und auch mit den Prüfsummen der Originaldatei übereinstimmen. Im Inhaltsdaten-Modus wird hierfür nur mittels der Prüfsummen die Korrektheit der Dateiinhalte überprüft, da keinerlei Metadaten dafür zur Verfügung stehen.

Im Folgenden werden Fälle in zwei verschiedenen Kategorien getestet. Als Realfälle gelten Datenträgerabbilder, die gewöhnliche Dateien und mit den mkfs-Standardparametern formatierte Dateisysteme besitzen. Auf diesen soll die Korrektheit und Vollständigkeit überprüft werden, da es durch die bekannte Anzahl tatsächlicher Dateien einen definierten Sollwert gibt, der durch die Rekonstruktion erfüllt werden muss.

Die zweite Testkategorie behandelt Spezialfälle, in denen ein bestehendes Ext4-Dateisystem mit anderen Dateisystemen überschrieben wurde. Auf diese Weise soll untersucht werden, ob sich Daten des darunter liegenden Dateisystems rekonstruieren lassen. Die Datenträgerabbilder, die mit einem anderen Dateisystem überformatiert wurden, werden nicht als Realfälle behandelt, da diese mit keinen neuen Dateien beschrieben wurden und sich somit keine für den Nutzer sichtbaren, wiederherstellbaren Dateien auf dem Datenträger befinden. Somit kann kein Sollwert für diese Testfälle angenommen werden. Eine obere Schranke für die Anzahl rekonstruierbarer Dateien stellen die Werte des `small.img`-Testfalles dar.

Realfälle

Um die Vollständigkeit der Ergebnisse überprüfen zu können, werden in Tabelle 4.5 die Mengen an Dateien aufgelistet, die sich auf den Datenträgern für den Nutzer sichtbar zur Rekonstruktion befinden. Auch die durch die Inode-Carving-Phase akzeptierten Inodes werden in dieser Tabelle für die jeweiligen Testfälle aufgezeigt.

Hierbei ist, wie bereits erwähnt, die Gesamtzahl aller Dateien nicht als Sollwert für das Werkzeug zu verstehen, da ausschließlich reguläre Dateien und Verzeichnisse wiederhergestellt werden. Nur im Falle von `Ubuntu_Pics_complete.img` entspricht die Gesamtzahl der Dateien nicht der Summe der Anzahlen regulärer Dateien und Verzeichnisse, da ausschließlich auf diesem Datenträgerabbild andere Dateitypen auftreten.

In Tabelle 4.6 sind die Endergebnisse der Rekonstruktion für die eben genannten Datenträgerabbilder notiert. Hierbei wird für jedes Datenträgerabbild jeweils angegeben, wie viele reguläre Dateien und Verzeichnisse mit beiden Modi wiederhergestellt worden sind. Die Anzahl der Solldateien wird jeweils fettgedruckt in der Zeile mit dem Namen des Datenträgerabbilds angegeben und in reguläre Dateien und Verzeichnisse unterteilt. Im Folgenden werden die angegebenen Ergebnisse näher erläutert.

Die gegebenen Sollwerte wurden von keinem der beiden Modi unterschritten. Es ist zu betonen, dass alle Solldateien vom vorgestellten Werkzeug wiederhergestellt wurden. Dies wurde ermittelt, indem sowohl für alle Solldateien, als auch für alle rekonstruierten Dateien MD5- und SHA256-Prüfsummen gebildet und gegeneinander abgeglichen wurden. Daher werden die Ergebnisse des Metadaten-Modus nicht näher erläutert, da diese in jedem Testfall exakt der Sollmenge entsprechen.

Datenträgername	alle Dateien	reguläre Dateien	Verzeichnisse
floppy.img	**6**	**4**	**2**
Akzeptiert	34	20	14
small.img	**128**	**121**	**7**
Akzeptiert	444	349	95
default.img	**5.102**	**4.866**	**236**
Akzeptiert	21.238	17.271	3.967
floppy_deleted.img	**13**	**8**	**5**
Akzeptiert	63	39	24
floppy_deletedTrash.img	**2**	**0**	**2**
Akzeptiert	41	24	17
floppy_newFiles.img	**28**	**25**	**3**
Akzeptiert	125	106	19
small_newFiles.img	**349**	**340**	**9**
Akzeptiert	1.229	1.110	119
default_newFiles.img	**9.062**	**7.773**	**1.289**
Akzeptiert	46.389	37.126	9.263
default_withoutJournal.img	**2**	**0**	**2**
Akzeptiert	2.516	1.020	1.496
Ubuntu_Pics_complete.img	**201.269**	**127.159**	**15.760**
Akzeptiert	617.802	493.885	123.917

Tabelle 4.5: Datensatz für die Evaluation mit der Anzahl der gesamten Dateien und akzeptierten Inodes

Im Falle des `Ubuntu_Pics_complete.img`-Abbilds wurden vom Metadaten-Modus leere Verzeichnisse erstellt und für alle nicht regulären Dateien je eine leere Datei mit dem entsprechenden Dateinamen angelegt. Zusätzlich zu den nutzergenerierten Daten auf jedem Datenträgerabbild, befinden sich das Wurzelverzeichnis und das vom Betriebssystem generierte `lost+found`-Verzeichnis, die in die Zählung mit aufgenommen werden. Es ist zu beachten, dass im Inhaltsdaten-Modus Verzeichnisse ignoriert und somit nicht rekonstruiert werden.

Die Datenträgerabbilder `floppy.img`, `small.img` und `default.img` haben im Inhaltsdaten-Modus, zusätzlich zu den regulären Dateien, jeweils den Inhalt des Journals wiederhergestellt, da dieser Inode für den Algorithmus wie eine reguläre Datei aussieht. Darüber hinaus wurden Dateien durch das Journal wiederhergestellt, die allerdings als false positives verstanden werden müssen.

Das Datenträgerabbild `floppy_deleted.img` wurde aus `floppy.img` generiert, indem alle darauf befindlichen Dateien gelöscht wurden. Hierbei wurden die Dateien in den Betriebssystem-spezifischen Papierkorb verschoben und nicht vollständig gelöscht. Im Papierkorb befinden sich die ursprünglichen Dateien, die dorthin verschoben wurden, und zusätzlich Metadaten-Dateien, die die ursprünglichen Pfade und Löschzeitpunkte im Textformat dokumentieren. Dadurch, dass der Papierkorb sowohl für die gelöschten Dateien, als auch für die dazugehörigen Metadaten-Dateien Verzeichnisse anlegt, und der Papierkorb beim ersten Löschvorgang erstellt wird, werden dem Dateisystem drei weitere Verzeichnisse hinzugefügt. Die Datei, die zusätzlich vom Inhaltsdaten-Modus rekonstruiert wurde, ist der Inode des Journals.

Wenn der Papierkorb zusätzlich gelöscht wurde, befinden sich keine nutzersichtbaren Dateien auf dem Datenträger. Dieser Fall wird durch `floppy_deletedTrash.img` dargestellt. In diesem Beispiel können durch den Metadaten-Modus, bis auf die zwei essentiellen Verzeichnisse, keine Dateien gefunden werden. Im Inhaltsdaten-Modus werden jedoch alle Dateien vollständig und korrekt rekonstruiert, die sich auch auf dem Datenträgerabbild `floppy_deleted.img` befinden.

Durch das Löschen der Dateien des Papierkorbs wurden im Journal die Änderungen und damit die entsprechenden Inodes dokumentiert. Durch diese können die Dateien wiederhergestellt werden, da die Datenblöcke nicht von anderen Dateien überschrieben worden sind. Unter den 24 gefundenen Inodes der regulären Dateien befinden sich 15 Duplikate. Diese kommen durch die Dokumentation des Journals der Änderungen am Dateisystem zustande.

Datenträgername mit Modus	alle Inodes	reguläre Dateien	Verzeichnisse
floppy.img	**6**	**4**	**2**
Metadaten-Modus		4	2
Inhaltsdaten-Modus		6	0
small.img	**128**	**121**	**7**
Metadaten-Modus		121	7
Inhaltsdaten-Modus		124	0
default.img	**5.102**	**4.866**	**236**
Metadaten-Modus		4.866	236
Inhaltsdaten-Modus		4.868	0
floppy_deleted.img	**13**	**8**	**5**
Metadaten-Modus		8	5
Inhaltsdaten-Modus		9	0
floppy_deletedTrash.img	**2**	**0**	**2**
Metadaten-Modus		0	2
Inhaltsdaten-Modus		9	0
floppy_newFiles.img	**28**	**25**	**3**
Metadaten-Modus		25	3
Inhaltsdaten-Modus		27	0
small_newFiles.img	**349**	**340**	**9**
Metadaten-Modus		340	9
Inhaltsdaten-Modus		355	0
default_newFiles.img	**9.062**	**7.773**	**1.289**
Metadaten-Modus		7.773	1.289
Inhaltsdaten-Modus		7.664	0
default_withoutJournal.img	**2**	**0**	**2**
Metadaten-Modus		0	2
Inhaltsdaten-Modus		0	0
Ubuntu_Pics_complete.img	**201.269**	**127.159**	**15.760**
Metadaten-Modus		127.159	15.760
Inhaltsdaten-Modus		168.035	0

Tabelle 4.6: Datensatz mit der Anzahl der gesamten und rekonstruierten Dateien

Wie im beschriebenen Fall `floppy_deletedTrash.img`, wurden die drei Datenträgerabbilder mit Dateisammlungen vollständig geleert. Anschließend wurden andere Dateien auf diese Dateisysteme geladen. Mit dem Metadaten-Modus werden alle neu geschriebenen Dateien in allen drei Fällen korrekt wiederhergestellt. Von besonderer Bedeutung ist die Tatsache, dass im Falle von `small_newFiles.img` im Inhaltsdaten-Modus eine Datei aus dem ursprünglichen Dateisystem wiederhergestellt werden konnte. Ihr Dateiinhalt war intakt; der dazugehörige Inode wurde im Journal gefunden. Die Datenblöcke, in denen der Dateiinhalt zu finden war, wurden durch die neuen Dateien nicht überschrieben. Da von diesen drei Datenträgerabbildern lediglich in einem Fall eine einzige Datei auf diese Weise wiederhergestellt werden konnte, ist davon auszugehen, dass die Wahrscheinlichkeit hierfür gering ist.

Weiterhin fällt auf, dass `default_newFiles.img` im Inhaltsdaten-Modus scheinbar weniger Dateien wiederherstellt, als minimal für die Korrektheit notwendig wäre. Da dieser Modus Duplikate im Sinne der Extent-Struktur aussortiert, gelten leere Dateien als identisch. Da sich im neuen Datensatz insgesamt 268 leere Dateien befinden, kann der Sollwert von 7.773 nicht erreicht werden. Allerdings wurden einige fehlerhafte Dateien wiederhergestellt, die aus dem Journal stammen und nicht zur Sollmenge gehören, wodurch der Unterschied der fehlenden Dateien nicht 268 beträgt.

Um zu kontrollieren, ob die Rekonstruktion von nicht nutzersichtbaren Dateien tatsächlich durch das Journal erfolgt, wurde mit `default_withoutJournal.img` ein Testfall formuliert, der kein Journal besitzt. Hier wurden Dateien in das Dateisystem geschrieben und anschließend wieder gelöscht, inklusive Papierkorb. Dadurch befinden sich für beide Modi keine rekonstruierbaren Dateien auf dem Dateisystem.

Schließlich wurde noch der Realfall durch das `Ubuntu_Pics_complete.img` abgedeckt. Der Unterschied der Anzahlen rekonstruierter regulärer Dateien durch den Metadaten- und den Inhaltsdaten-Modus ergibt sich durch verschiedene Faktoren. Einerseits beinhaltet das Journal viele Einträge geänderter Inodes, da durch die Installation viele Dateien erstellt, gelöscht und verschoben werden, die im Inhaltsdaten-Modus rekonstruiert wurden. Andererseits ignoriert der Inhaltsdaten-Modus leere Dateien, die jedoch unter Systemdateien vorkommen. Zwischen den zusätzlichen Dateien, die durch den Inhaltsdaten-Modus erstellt werden, befinden sich jedoch keinerlei interpretierbare Dateien. Unter dem Ubuntu-12.04-Betriebssystem befand sich eine Datei auf dem Datenträger, die zwar intakt, aber vom vorgestellten Werkzeug nicht interpretierbar war. Diese Datei hatte eine leere Extent-Struktur, wie eine leere Datei, allerdings war die Dateigröße auf 1 Byte, statt auf 0 Byte gesetzt. Für diesen Fall wurde eine Sonderbehandlung eingebaut.

Spezialfälle

Die folgenden Tests sollen analysieren, ob das vorgestellte Werkzeug dazu geeignet ist, auf überformatierten Dateisystem die ursprünglichen Dateien wiederherzustellen. Eine obere Schranke für die Menge an Dateien, die durch das forensische Werkzeug wiederhergestellt werden soll, stellen die Dateien auf `small.img` dar. Zunächst wurde überprüft, ob Dateien im allgemeinen Fall für eine Überformatierung mit Ext4 und NTFS überhaupt möglich ist.

Diese Fälle werden durch `small_newExt4.img`, `small_diffExt4.img` und `small_NTFS.img` repräsentiert. Dem Werkzeug wurden hierbei stets die Ext4-Parameter von `small.img` übergeben, da dieses effektiv wiederhergestellt werden soll. Da bei einer Überformatierung die Datenblöcke ausgenullt werden, konnte in diesen Fällen keine Datei vom ursprünglichen Dateisystem rekonstruiert werden. Somit sind die Ergebnisse dieser drei Dateisysteme von keiner besonderen Relevanz.

Sowohl Ext4 als auch NTFS bieten Optionen zur sogenannten schnellen Formatierung an. Hierbei werden ausschließlich die Dateisystemblöcke überschrieben, für die das neue Dateisytem notwendige Metadatenstrukturen vorsieht. Alle anderen Blöcke behalten ihren ursprünglichen Inhalt. Das mkfs-Programm bietet dies mit der Kommandozeilen-Option `-E nodiscard` für das Ext4-Dateisystem und mit `-f` für das NTFS-Dateisystem an.

In Tabelle 4.7 werden die gefundenen Inodes und Dateien für die schnell formatierten Dateisysteme gezeigt. Ebenso werden die gefundenen Ergebnisse mittels beider Modi aufgezeigt. Da sich auf den verwendeten überformatierten Dateisystemen keinerlei Dateien befinden, stammen die dadurch gefundenen Inodes vom ursprünglichen Ext4-Dateisystem. Aus diesem Grund ergibt sich keine definierte Sollmenge, sondern nur eine obere Schranke die durch `small.img` gegeben ist.

Datenträgername mit Modus	alle Inodes	reguläre Dateien	Verzeichnisse
small_fastExt4.img	**128**	**121**	**7**
Akzeptiert		171	77
Metadaten-Modus		118	5
Inhaltsdaten-Modus		119	0
small_fastdiffExt4.img	**128**	**121**	**7**
Akzeptiert		228	90
Metadaten-Modus		0	0
Inhaltsdaten-Modus		125	0
small_fastNTFS.img	**128**	**121**	**7**
Akzeptiert		348	94
Metadaten-Modus		121	7
Inhaltsdaten-Modus		124	0

Tabelle 4.7: Überformatierter Datensatz mit der Anzahl der gefundenen Inodes nach der Selektion

Im Datenträgerabbild `small_fastExt4.img` wurde eine Neuformatierung mit den gleichen Standardparametern wie bei der ursprünglichen Formatierung angewendet. Bei der Betrachtung der Ergebnisse fällt auf, dass im Metadaten-Modus, bis auf 3 Dateien und 2 Verzeichnisse, alle Solldateien korrekt wiederhergestellt werden. Genauer wurden alle Inodes ab Inode 17 intakt vorgefunden. Die Inode-Nummer 11 wird dem Verzeichnis `lost+found` zugewiesen; Die Inodes 12 bis 16 entsprechen den fehlenden Dateien und Verzeichnissen. Durch die Parametergleichheit ist zu erwarten, dass alle Metadatenpositionen gleich ausfallen und durch den schnellen Formatierungsmodus nur an den Stellen überschrieben werden, wo die Neuformatierung neue Daten anlegt. Dies betrifft die reservierten Inodes 1 bis 10, jedoch belegen diese 128 Byte großen Inodes Speicher in zwei der 1024 Byte großen Blöcke. Da der Speicher von mkfs blockweise überschrieben wird, befinden sich somit die ersten 16 Inodes in einem neuen Zustand.

Davon ist auch Inode 2, das Wurzelverzeichnis, betroffen. Dies hat zur Folge, dass das Werkzeug keinen zusammenhängenden Verzeichnisbaum aufstellen kann und dadurch sowohl elternlose Dateien als auch Verzeichnisse erstellt. Da somit auch Verzeichniseinträge überschrieben oder unerreichbar gemacht wurden, konnten einige der Dateinamen und Verzeichniszugehörigkeiten nicht rekonstruiert werden. Der Inhaltsdaten-Modus stellt zusätzlich Inode 8, das Journal, wieder her.

Im Fall von `small_fastdiffExt4.img` lagen keine der ursprünglichen Inode-Tabellen in Bereichen, die durch die Neuformatierung intakt geblieben sind. Aufgrund dessen kann der Metadaten-Modus keine Wiederherstellung durchführen. Das neue Journal hat das ursprüngliche jedoch nicht überschrieben, da durch die unterschiedlichen Ext4-Parameter dieses an eine andere Stelle innerhalb des Dateisystems verschoben wurde. Auf diese Weise konnten durch den Inhaltsdaten-Modus insgesamt 125 Dateien wiederhergestellt werden. Von den gefundenen Dateiinhalten stimmen 52 in ihren MD5- und SHA256-Prüfsummen mit Dateien aus `small.img` überein und sind somit vollständig intakt rekonstruiert worden.

Das im Fall von `small_fastNTFS.img` verwendete NTFS-Dateisystem benötigt für seine Metadaten zu Beginn der Partition weniger Speicher, als bis zum Anfang der ersten Inode-Tabelle veranschlagt ist. Daher sind alle Inodes intakt, wodurch beide Modi genau die Dateien finden, die sie auch im Beispiel von `small.img` wiederherstellen konnten. Allerdings sind einige wenige dieser Dateien teilweise beschädigt; die NTFS-Formatierung hat demnach Teile des ursprünglichen Datenbereichs überschrieben.

Es kann zusammengefasst werden, dass das Werkzeug dazu geeignet ist, schnell überformatierte Partitionen bei Kenntnis der ursprünglichen Parameter nach Dateien zu durchsuchen. Ist ein entsprechendes Dateisystem nicht im schnellen Modus überformatiert worden, stehen keine rekonstruierbaren Dateien zur Verfügung.

4.3 Performance

Zusätzlich zu den Überprüfungen der Korrektheit und Vollständigkeit der Ergebnisse des Programms fanden Analysen der Laufzeit statt. Neben der Rekonstruktion schnell überformatierter Ext4-Dateisysteme ist es ein weiteres Ziel des vorgestellten forensischen Werkzeugs, Datenträgerabbilder mäßiger Größe mit herkömmlichen Desktop-Rechnern in absehbarer Zeit zu analysieren. Zu diesem Zweck wurde die Laufzeit des Moduls anhand einiger Testfälle evaluiert, um die praktische Anwendbarkeit aufzuzeigen. Dabei wurden die Messungen auf folgendem System durchgeführt:

- CPU: Intel®Core™i5-3570; 4 Kerne @ 3,40GHz
- RAM: 16 GiB @ 1,600GHz
- Festplatte angeschlossen mittels SATA-2
- Betriebssystem: Ubuntu 14.04 LTS; 64-Bit-Version

Im Folgenden wurden einige der `small`- und `default`-Datenträgerabbilder getestet. Durch die unterschiedlichen Testfälle kann der Einfluss der Anzahl der Inodes auf der Festplatte und der der Größe des Dateisystems auf die Laufzeit ermittelt werden. Der Testfall `default_sameFiles.img` wurde speziell für die Laufzeitmessung erzeugt; auf ihm befinden sich dieselben Dateien wie auf `small_newFiles.img`. Da so die selbe Anzahl an rekonstruierbaren Inodes auffindbar sind, kann hiermit der Einfluss der Dateisystemgröße und der Größe des Journals überprüft werden. Das Journal beinhaltet allerdings Einträge für Inodes, die zu einem Unterschied in der Menge gefundener Inodes führt, jedoch nicht an der Menge rekonstruierbarer Dateien im Metadaten-Modus.

In Tabelle 4.8 werden die Laufzeiten für die Inode-Carving-Phase aufgezeigt. Hierbei wurden die Suchmuster genauso konfiguriert, wie bereits in den Experimenten zuvor.

Datenträgername	gefundene Inodes	Zeit
small.img	444	4,5s
small_newFiles.img	1.229	5,1s
small_fastNTFS.img	442	4,2s
default.img	21.238	1m20s
default_sameFiles.img	10.389	2m28s

Tabelle 4.8: Laufzeit für die Inode-Carving-Phase für ausgewählte Testfälle

Die Laufzeit verändert sich für die getesteten Fälle mit der Größe des Dateisystems und der Anzahl der Inodes, die darauf gefunden werden. Da auf einem größeren Dateisystem mehr Speicher durchsucht werden muss, und dadurch auch mehr Vergleiche ausgeführt werden müssen, verhält sich die Laufzeit zwischen den `small`- und den `default`-Datenträgerabbildern nicht linear. Vor allem bei den Testfällen `default.img` und `default_sameFiles.img` wird der Unterschied klar, denn beide Datenträger haben das gleiche Volumen und auf letzterem befinden sich sogar weniger potentielle Inodes. Jedoch unterscheidet sich die Laufzeit für das Suchen nach Inodes signifikant, was an der Art der Inodes liegt, die sich auf dem Dateisystem befinden. Bei `default_sameFiles.img` finden sich viele Inodes im Journal wieder, die durch Änderungen und Löschvorgänge dort befinden, durch den Suchalgorithmus allerdings erst spät aussortiert werden. Dies legt die Schlussfolgerung nahe, dass die Anzahl und Beschaffenheit der Inodes auf dem Datenträgerabbild einen wesentlich größeren Einfluss auf die Laufzeit des Programms hat, als die reine Größe des Dateisystems.

In Tabelle 4.9 wird die Laufzeit des Moduls für diese Testfälle aufgeführt. Hierbei wurde die Zeit für die Rekonstruktion mit in die Messung aufgenommen und das rekonstruierte Datenvolumen mit der Anzahl rekonstruierter Dateien aufgelistet. In diesem Fall werden Verzeichnisse und reguläre Dateien zusammengefasst.

Datenträgername	Anzahl Dateien	Datenvolumen	Gesamtzeit
small.img			
Metadaten-Modus	128	128,5 MB	5,6s
Inhaltsdaten-Modus	124	134,1 MB	5,4s
small_newFiles.img			
Metadaten-Modus	349	84,5 MB	5,9s
Inhaltsdaten-Modus	355	111,7 MB	5,3s
small_fastNTFS.img			
Metadaten-Modus	128	128,5 MB	5,3s
Inhaltsdaten-Modus	124	134,1 MB	5,3s
default.img			
Metadaten-Modus	5.102	5,3 GB	2m36s
Inhaltsdaten-Modus	4.868	5,4 GB	2m28s
default_sameFiles.img			
Metadaten-Modus	349	84,5 MB	2m40s
Inhaltsdaten-Modus	2.968	3,1 GB	3m14s

Tabelle 4.9: Laufzeit für die Ausführung des Moduls mit beiden Modi

Durch das unterschiedliche Datenvolumen und die Anzahl an Dateien, die bei der Rekonstruktion angelegt werden müssen, hängt die gemessene Laufzeit stark vom Anwendungsfall ab. Der Unterschied zwischen dem Laufzeiten des Inhaltsdaten- und des Metadaten-Modus besteht darin, dass das Auslesen der Verzeichniseinträge zusätzliche Rechenzeit benötigt, selbst wenn dabei ein geringeres Datenvolumen rekonstruiert wird. Das höhere Datenvolumen im Inhaltsdaten-Modus folgt daraus, dass manche Dateien rekonstruiert werden, die durch das Journal als Inodes erkannt und dadurch rekonstruiert wurden; auch wenn diese meist keine interpretierbaren Inhalte enthalten.

Die ähnliche Laufzeit für `default.img` und `default_sameFiles.img` (hier nur im Inhaltsdaten-Modus) liegt darin begründet, dass, obwohl das rekonstruierte Datenvolumen differiert, die Zeit für die Inode-Carving-Phase bei beiden Fällen unterschiedlich ist. So wird beim `default.img` ca. die Hälfte der Laufzeit für die Rekonstruktion aufgewendet, wo beim anderen Datenträgerabbild ein Großteil der Laufzeit für die Inode-Carving-Phase aufgewendet wird.

Bei `default_sameFiles.img` und `small_newFiles.img` ist zu erkennen, dass die Größe des Dateisystems auch dann einen Einfluss auf die Laufzeit hat, wenn die Menge an rekonstruierten Daten gleich ist. Selbst wenn die Laufzeit für die Inode-Carving-Phase von der Gesamtzeit abgezogen wird, so benötigt das `default`-Dateisystem länger für die Rekonstruktion der Dateien, was sich ebenfalls durch die Größe des Datenträgerabbilds begründen lässt. Der große Unterschied zwischen dem Metadaten- und dem Inhaltsdaten-Modus in `default_sameFiles.img` folgt daraus, dass viele Dateien aus dem Journal rekonstruiert werden, was einem großen rekonstruierten Datenvolumen entspricht.

Allgemein lässt sich folgern, dass die Laufzeit für die Rekonstruktion von Dateien mittels des vorgestellten forensischen Werkzeugs von der Größe des Dateisystems abhängt. Die Menge an wiederhergestellten Dateien beeinflusst die Laufzeit ebenfalls, jedoch ist deren Einfluss auf die Laufzeitänderung verhältnismäßig gering. Darüber hinaus muss beachtet werden, dass im Metadaten-Modus die Rekonstruktion von Verzeichnissen wenig zum Datenvolumen beiträgt, jedoch die Komplexität der Rekonstruktion erhöht, was sich in der Laufzeit niederschlägt. Ebenso beeinflusst die Menge an potentiellen Inodes auf dem Dateisystem die Laufzeit für die Suchmustervergleiche.

Auch, wenn das rekonstruierte Datenvolumen minimal ist und die überprüften Dateisysteme bezüglich ihrer Größe keinem Realfall entsprechen, so ist doch zu erkennen, dass die Rekonstruktion der Dateien stark vom Anwendungsfall abhängt. Da bei dieser Arbeit allerdings die Korrektheit und Vollständigkeit der Ergebnisse im Vordergrund steht, wurde kein weiteres Augenmerk auf Laufzeitoptimierung gelegt.

5

ZUSAMMENFASSUNG UND AUSBLICK

In den vorigen Kapiteln wurde ein Ansatz präsentiert, der für die Rekonstruktion von Dateien auf Ext4-Dateisystemen entwickelt wurde. Dieser Ansatz verzichtet dabei auf das Auslesen zentraler Metadatenstrukturen, wie etwa des Superblocks oder der Gruppen-Deskriptor-Tabelle. Mittels einer Kombination aus einer File-Carving-Methode und einer Form der Metadatenanalyse werden Inodes mittels Suchmustern gefunden und anschließend die dazugehörigen Dateien rekonstruiert. Die Implementierung erfolgte als Modul für das Sleuthkit-Framework.

Der Ansatz eignet sich für verschiedene Anwendungsszenarien. Sowohl bei überschriebenen, als auch bei überformatierten Dateisystemen können Daten rekonstruiert werden. Dabei hängt es allerdings stark vom Anwendungsfall ab, wie viele Dateien rekonstruiert werden können und welchen Grad von Intaktheit sie aufweisen. Die Rekonstruktion der Dateien basiert auf den gefundenen Inodes des Ext4-Dateisystems.

Es existieren Einschränkungen, die nicht nur vom Ansatz, sondern ebenso von der Implementierung abhängen. Durch einige Anpassungen der Suchmuster und der Rekonstruktionsverfahren können manche Einschränkungen behoben werden. Eine dieser Einschränkungen ist die Abhängigkeit von der Standardformatierung des Ext4-Dateisystems. Diese lässt sich durch eine manuelle Übergabe der Parameter oder eine neue Parameterschätzungskomponente behandeln.

In diesem Kapitel wird zuerst eine Zusammenfassung der vorgestellten Arbeit gegeben. Anschließend werden die aus den gewonnenen Erkenntnissen gezogenen Schlussfolgerungen gegenübergestellt. Danach werden einige Einschränkungen aufgezeigt, die dieser Ansatz und das entwickelte Modul aufweisen. Schließlich wird noch ein Ausblick auf die mögliche zukünftige Arbeit am vorgestellten Ansatz gegeben.

5.1 Zusammenfassung

Die vorgestellte Arbeit behandelt einen Algorithmus zur Wiederherstellung von Dateien aus Ext4-Dateisystemen, der mittels einer Kombination aus einer Variante des File-Carvings und Methodiken der Metadatenanalyse vorgeht. Die zentralen Metadaten, wie der Superblock und die Gruppen-Deskriptor-Tabelle, werden dabei nicht betrachtet. Inodes aus dem Ext4-Journal können im Inhaltsdaten-Modus interpretiert werden, ohne, dass hierfür Kenntnisse über das Journal vonnöten sind. Alle für die Analyse des Dateisystems notwendigen Parameter werden vom vorgestellten Ansatz anhand von Standardwerten geschätzt, oder manuell an das forensische Werkzeug übergeben.

Mittels ausgewählter Suchmuster wird auf dem Datenträger nach Inode-Datenstrukturen gesucht, um anschließend ihre Inhaltsdaten zu rekonstruieren. Auch die Option, einige semantische Suchmuster, wie beispielsweise Zeitintervall-Einschränkungen, manuell zu konfigurieren, wurde vorgestellt. Mit den so ermittelten Inodes wird anschließend eine vom Dateityp abhängige Metadatenanalyse betrieben. Lediglich die Intaktheit der Inodes und der dazugehörigen Datenblöcke muss für die Rekonstruktion gefordert werden, wobei teils überschriebene Datenblöcke nicht die Wiederherstellung, sondern die wiederhergestellten Dateien beeinflussen.

Für diesen Ansatz wurde ein forensisches Werkzeug bereitgestellt, das als Modul für das Sleuthkit-Framework entwickelt wurde. Mittels verschiedener Optionen kann die Ausführung des Moduls an die Gegebenheiten des zu untersuchenden Datenträgers angepasst werden. Ebenso bietet es die Möglichkeit, die Rekonstruktion von Dateien mit zwei unterschiedlichen Modi durchzuführen. Der Inhaltsdaten-Modus kann mit minimalen Informationen über das Dateisystem – ausschließlich die Ext4-Blockgröße muss bekannt sein – reguläre Dateien rekonstruieren. In diesem Modus wird keine Unterscheidung über den Fundort der Inodes getroffen, wodurch Inodes aus dem Journal mitbetrachtet werden. Dabei muss weder die Inodenummer, noch die Position des Journals bekannt sein; die Journal-Metadaten selbst müssen nicht ausgelesen werden. Inodes aus dem Journal sind nicht speziell formatiert und können auf gewöhnliche Weise interpretiert werden. Im Gegensatz dazu stellt der Metadaten-Modus eine Variante des Ansatzes dar, mit dem durch die Interpretation von Verzeichnissen die Dateinamen und -pfade rekonstruiert werden können, so dass die komplette Verzeichnishierarchie aufgebaut werden kann. Für diesen Modus müssen allerdings Dateisystemparameter bekannt sein, aus denen sich weitere essentiellen Werte errechnen lassen. Zu diesen Werten zählen zum Beispiel die Inodenummern gefundener Inodes.

In Kapitel 1 wurde die Motivation für diesen Ansatz dargelegt. Dabei wurde auf bestehende Ansätze zur Rekonstruktion von Dateien von Ext4-Dateisystemen eingegangen und ihre Schwächen aufgezeigt, die durch den vorgestellten Ansatz ausgeglichen werden. Auch die Notwendigkeit von forensischen Open-Source-Werkzeugen wurde erklärt. Der derzeitige Stand der Technik bezüglich des betrachteten Felds wurde ebenso vorgestellt wie die Beiträge der vorliegenden Arbeit.

In dieser Arbeit wurde für das Verständnis des entwickelten Ansatzes in Kapitel 2 eine Einführung in verschiedene Themen gegeben. Dafür wurde auf die Ext-Dateisystem-Familie eingegangen und die neueste Version (Ext4) detailliert auf technischer Ebene erklärt. Dabei wurden auch die Vorgängerversionen Ext2/3 erläutert, da viele der Datenstrukturen des Ext4-Dateisystems bezüglich ihrer Vorgänger unverändert geblieben sind. Dazu gehört ein Großteil der Inode-Datenstruktur und der Gruppen-Deskriptoren. Zusätzlich wurden u.A. die indirekten Blockzeiger der Ext2/3-Dateisysteme und die Extent-Struktur von Ext4 beschrieben. Auch verschiedene Ansätze zur Dateirekonstruktion wurden vorgestellt, wie der File-Carving-Ansatz und die Metadatenanalyse. Es wurde hierbei auch auf die Möglichkeiten und Einschränkungen beider Ansätze eingegangen. Da der Ansatz in einem Modul für das Sleuthkit-Framework implementiert wurde, wurde weiterhin ein Überblick über die Funktionsweise des Frameworks und den Funktionsumfang des Sleuthkits gegeben.

Anschließend wurde der vorgestellte Ansatz in Kapitel 3 sowohl algorithmisch, als auch technisch detailliert erklärt. Dies wurde in zwei Abschnitten, die jeweils nach den Programmablaufphasen des Werkzeugs gegliedert sind, abgehandelt. Neben der Erläuterung der Auswahl der Suchmuster wurde die Berechnung der essentiellen Parameter aufgezeigt. Diese basieren auf den von mkfs verwendeten Standardwerten und dem Standardaufbau für das Ext4-Dateisystem. Weiterhin wurden die Implementierungsdetails des forensischen Werkzeugs näher beleuchtet. Im Zuge dessen wurde das rekursive Auslesen von Verzeichnissen und regulären Dateien erläutert. Auch die Rekursion zur Ermittlung der Dateipfade wurde in diesem Kapitel beleuchtet. Die Einbindung in das Sleuthkit-Framework wurde, zusammen mit dem Aufbau und der Benutzung des entwickelten Werkzeugs, näher beschrieben.

Schlussendlich wurde der Ansatz mittels eines ausgewählten Datensatzes in Kapitel 4 evaluiert. Dabei wurden die Selektionsraten der einzelnen Suchmuster gemessen und darauf aufbauend einige Suchmuster für die anschließenden Tests ausgewählt. Auch die Korrektheit der Ergebnisse wurde für beide bereitgestellten

Modi mittels MD5- und SHA256-Prüfsummenvergleichen ermittelt. Dabei wurde die Vollständigkeit der durch die Suchmuster erfassten Dateien, als auch die Intaktheit der rekonstruierten Dateien überprüft. Da sich die Ergebnisse beider Modi unterscheiden, wurden beide getrennt betrachtet. Ebenso wurde der Fall analysiert, in dem ehemalige Ext4-Datenträgerabbilder mit anderen Dateisystemen schnell überformatiert wurden. Schließlich wurde die Laufzeit des Moduls für verschiedene Anwendungsfälle ermittelt.

5.2 Schlussfolgerungen

Aus den Ergebnissen dieser Arbeit lässt sich folgern, dass sich mit dem vorgestellten Ansatz Dateien von Ext4-Dateisystemen rekonstruieren lassen, selbst ohne Kenntnisse über den speziellen Aufbau des Dateisystems. Durch die Trennung der Inode-Suche von der -Rekonstruktion ist es möglich, auch dann Inodes zu finden, wenn keinerlei Informationen über das Dateisystem bekannt sind. Weder die Größe des Dateisystems, noch der korrekte Offset zum Beginn der Partition des zu untersuchenden Ext4-Dateisystems sind notwendig, um Inodes aufzufinden und somit erste Anhaltspunkte für potentielle Dateien zu erhalten. Dies wird durch die byteweise Suche nach Inodes erreicht.

Um die gefundenen Inodes rekonstruieren zu können, ist die Kenntnis der Blockgröße des Dateisystems und des Offsets der Partition jedoch notwendig, damit Blockadressen korrekt aufgelöst werden können. Dieser Parametersatz genügt für die Rekonstruktion im Inhaltsdaten-Modus; für die korrekte Wiederherstellung mittels des Metadaten-Modus sind weitere Parameter vonnöten. Weichen diese von den mkfs-Standardwerten ab, ist eine korrekte Angabe seitens des Nutzers durch die Konfigurationsdatei notwendig.

Die durch die verschiedenen Suchmuster durchgeführte Selektion liefert stark anwendungsfallabhängige Ergebnismengen. So können beispielsweise keine Dateien von Datenträgern wiederhergestellt werden, deren Dateisysteme kein Journal besitzen und deren Dateien alle gelöscht sind. Die Menge wiederherstellbarer Dateien und ihr Zustand auf überformatierten und überschriebenen Datenträgern hängt ebenso stark vom genauen Fall ab, etwa vom Grad der Fragmentierung oder der Art der Neuformatierung. Weiterhin muss bei Suchmustern unterschieden werden, ob ihre Einschränkungen semantischer oder syntaktischer Natur sind. Eine klare Trennung ist hierbei nicht möglich. Die Überprüfung der inneren Konsistenz von Zeitstempeln ist nicht mit einer Einschränkung auf intakte Inodes behaftet. Bei der Einschränkung auf ein Zeitintervall jedoch, werden auf semantischer Basis gültige Inodes aussortiert.

Auch ohne die Interpretation des Journals kann dieses zur Rekonstruktion mittels des vorgestellten Ansatzes genutzt werden. Im Inhaltsdaten-Modus gibt es keine Möglichkeit, zwischen Inodes aus Inode-Tabellen und Inodes aus dem Journal zu unterscheiden. Dadurch werden zwar viele Duplikate gefunden, diese können allerdings aussortiert werden. So können gelöschte Dateiinhalte rekonstruiert werden, wenn sich der ungelöschte Inode im Journal befindet. Im Metadaten-Modus ist die Rekonstruktion mittels des Journals nicht möglich, da die Einschränkung gilt, dass alle betrachteten Inodes aus einer Inode-Tabelle stammen müssen. Durch die unterschiedlichen Herangehensweisen beider Modi lässt sich kein abschließendes Urteil über ihre Ergebnisqualität fällen. Der Inhaltsdaten-Modus rekonstruiert potentiell mehr Dateien, der Metadaten-Modus genauer beschriebene.

Durch die Kombination aus File-Carving und Metadatenanalyse ist der vorgestellte Ansatz nicht abhängig vom Dateiformat, sondern nur vom Dateisystem. Somit kann jede Art von Datei rekonstruiert werden. File-Carving-Ansätze beschränken sich meist auf Dateiformate, die regulären Dateien entsprechen. Durch den vorgestellten Ansatz können allerdings alle regulären Dateien rekonstruiert werden, da diese unabhängig vom Dateiformat gespeichert werden. Im Metadaten-Modus ist es gar möglich, anhand der Verzeichniseinträge das Vorkommen von Dateien anderer Typen zu erkennen und an ihrer Stelle leere Dateien zu erzeugen.

Auch die Größe der zu rekonstruierenden Dateien stellt kein Hindernis für den Rekonstruktionsalgorithmus dar. Durch das blockweise Herausschreiben der Dateiinhalte können diese beliebige Größen annehmen, solange die Festplatte, auf der die rekonstruierte Datei gespeichert werden soll, hierfür die notwendige Kapazität bereitstellt.

5.3 Einschränkungen

Der entwickelte Ansatz birgt einige Einschränkungen. So beeinflusst etwa die Auswahl der Suchmuster stark das Ergebnis der Rekonstruktion. Wie durch die Selektionsrate der Suchmuster beschrieben, wird eine Vorauswahl an relevanten Dateien getroffen, die bei der Rekonstruktion beachtet werden. Vor allem bei semantischen Suchmustern ist die Selektionsrate hoch, jedoch können durch diese intakte Dateien ignoriert werden. Werden beispielsweise nur Inodes mit einer Dateigröße > 0 akzeptiert, können keine leeren Dateien wiederhergestellt werden. Auf eine ähnliche Weise führt die Nutzung jedes Suchmusters zu weiteren Einschränkungen des Raums auffindbarer Inodes.

Zusätzlich kann die Intaktheit eines Inodes nicht daraus erschlossen werden, dass diese in der Inode-Carving-Phase anhand der Suchmuster akzeptiert worden sind. Es ist bei jeder Kombination von Suchmustern möglich, dass Inodes, die zum Teil überschrieben wurden, als valide eingestuft werden. Dem könnte durch zusätzliche Konsistenzüberprüfungen entgegengewirkt werden.

Weiterhin sind mit dem vorgestellten Ansatz gelöschte Dateiinhalte ausschließlich dann rekonstruierbar, wenn der Inhaltsdaten-Modus auf ein Dateisystem mit einem Journal angewendet wird. Auch in diesem Fall können nur gelöschte Dateiinhalte wiederhergestellt werden, deren letzte Modifikation noch im Journal dokumentiert ist. Die Verknüpfung des Dateiinhalts mit seinem Inode ist mit dem vorgestellten Ansatz nicht möglich.

Falls Abwärtskompatibilität zu Ext2/3-Dateisystemen gegeben sein soll, muss eine Validierung der Inodes stattfinden, da es beim Mechanismus der indirekten Blockzeiger keine Header oder ähnliche Merkmale gibt, die diese kennzeichnen. Suchmuster, die anhand der Blockzeiger-Struktur einen Inode erkennen, müssen heuristisch vorgehen, da fast jeder Blockzeiger-Inhalt gültig ist. Da die Blockzeiger-Struktur 4-Byte-Adressen speichert, kann als Konsistenzüberpüfung nur ermittelt werden, ob die Adresse auf einen Speicherbereich außerhalb des Dateisystems zeigt.

Eine weitere Einschränkung des Ansatzes stellt die Anordnung der Metadatenstrukturen innerhalb einer Blockgruppe dar. Da der Ansatz die Offsets zu den Metadatenstrukturen selbst errechnet, muss in diesem Fall von einer Standardanordnung ausgegangen werden. In allen vorgestellten Testfällen wichen die Inode-Tabellen und die Inode- und Block-Bitmaps nicht von ihrer Standardreihenfolge ab. Allerdings bietet die Gruppen-Deskriptor-Tabelle die Möglichkeit, Änderungen an dieser Reihenfolge durchzuführen. Da die Gruppen-Deskriptor-Tabelle nicht ausgelesen wird, können diese Offsets nicht ermittelt werden. Hierfür müssten weitere Suchmuster oder Validierungsmechanismen angewendet werden, um die tatsächliche Reihenfolge der Metadatenstrukturen zu ermitteln.

Nicht nur durch den beschriebenen Ansatz, sondern auch durch das entwickelte forensische Werkzeug ergeben sich Einschränkungen. Beispielsweise beschränkt sich die Dateirekonstruktion auf reguläre Dateien. Durch den Metadaten-Modus können zwar Verzeichnisse gelesen und interpretiert werden, jedoch kann ein Dateisystem nicht vollständig rekonstruiert werden, da andere Dateitypen nicht unterstützt werden, wie etwa symbolische Verknüpfungen oder Gerätedateien.

Eine weitere Einschränkung liegt in der Größe der Verzeichnisse. Da im Metadaten-Modus ein interner Verzeichnisbaum aufgebaut wird, müssen alle Verzeichniseinträge in den internen Baum eingefügt werden. Bei der momentanen Implementierung werden diese Informationen vollständig im Hauptspeicher gehalten. Dadurch ergibt sich eine obere Schranke für die Größe der Verzeichnisse, bzw. für die Komplexität der Verzeichnishierarchie.

Auch die Rekonstruktion von Inhaltsdaten mittels indirekter Blockzeiger müsste implementiert werden, falls Kompatibilität zu Ext2/3-Dateisystemen gefordert wird. Weiterhin werden einige spezielle Erweiterungen des Ext4-Dateisystems, die in der Arbeit nicht beachtet wurden, wie etwa Meta-Blockgruppen oder Inline-Dateien und -Verzeichnisse, vom Werkzeug nicht unterstützt.

5.4 Ausblick und weiterführende Arbeit

Eine Möglichkeit zur Weiterentwicklung des vorgestellten Ansatzes ist die Erweiterung und Verbesserung der Suchmuster. Um Abwärtskompatibilität zu Ext2/3 zu bieten, müssen neue Suchmuster entwickelt werden. Da bei den bisherigen Tests eines der restriktivsten, nicht semantischen Suchmuster der Extent-Header ist, muss hierfür eine Alternative gefunden werden, wenn Ext2/3-Dateisysteme vom vorgestellten Werkzeug unterstützt werden sollen.

Ein Beispiel wäre die Kombination aus der Anzahl der reservierten Blöcke einer Datei und die Dateigröße. Beide Werte stehen im Inode und können mittels Konsistenzprüfung verglichen werden, da die Datei nicht größer sein darf, als durch die reservierten Blöcke vorgegeben wird. Da unter Ext2/3 keine erweiterten Attribute zur Verfügung stehen, die abhängig vom Betriebssystem behandelt werden müssen, könnte dieser Vergleich weitere Inodes aussortieren.

Ebenso müssen, für die Kompatiblität mit von Ext3 auf Ext4 migrierten Systemen, für die Rekonstruktion weitere Möglichkeiten der Abspeicherung von Inhaltsdaten in Betracht gezogen werden. Nicht nur indirekte Blockzeiger, die durch eine Migration von Ext2/3 auf Ext4 zustande kommen können, sondern auch Inline-Dateien und -Verzeichnisse müssten etwa betrachtet werden.

Bei der Rekonstruktion von Dateien kann außerdem die Erweiterung um weitere Dateitypen vorgenommen werden. Damit könnten beispielsweise Gerätedateien und symbolische Verknüpfungen rekonstruiert werden. Bei letzteren könnte zusätzlich unterschieden werden, ob ihr roher Inhalt rekonstruiert oder ihre logische Verknüpfung bezüglich des Zielverzeichnisses wiederhergestellt werden soll.

Unabhängig vom Dateityp wäre es denkbar, neben dem Inhalt einer Datei auch ihre Inode-Metadaten zu rekonstruieren. Der Inhalt des rekonstruierten Inodes würde dann in den Inode der rekonstruierten Datei kopiert werden. Dies würde beispielsweise Flags, Zeitstempel und Zugriffsrechte betreffen.

Die Verzeichnishierarchie, die auf einem Ext4-System mittels der Verzeichniseinträge realisiert wird, wird intern mit einer Rekursion aufgelöst. Dabei gibt es zwar einen Abbruchfall, um Endlosschleifen zu erkennen, doch beruht dieser darauf, dass es eine konfigurierbare, maximale Tiefe der Verzeichnisstruktur gibt. Wenn die Rekursion diese Schranke übersteigt, bricht die Rekursion ab. Um mögliche Endlosschleifen besser zu erkennen, könnte hierbei während der Rekursion, unter Nichtbeachtung symbolischer Verknüpfungen, eine Sammlung bereits besuchter Knoten geführt werden. Falls ein Knoten eingefügt werden soll, der bereits in der Sammlung vorkommt, müsste abgebrochen werden, weil die Verzeichnishierarchie per Definition azyklisch ist.

Da der Metadaten-Modus und der Inhaltsdaten-Modus unterschiedlich vorgehen, bringen ihre Ergebnisse unterschiedliche Vorteile mit sich. In der momentanen Implementierung sind die beiden Vorgehen nicht kombinierbar, allerdings wäre es wünschenswert, in einem einzigen Programmdurchlauf Ergebnisse mit den Vorteilen beider Modi zu erzeugen. Damit wäre es möglich, sowohl die Verzeichnishierarchie durch den Metadaten-Modus zu erzeugen, als auch, Dateien zu rekonstruieren, deren Inodes sich ausschließlich im Journal befinden.

Verwandt mit dieser Idee ist die Abbildung von Inodes aus dem Journal auf ihre korrespondierenden Inodes aus den Inode-Tabellen. Dies könnte genutzt werden, um gelöschte Dateien – deren Extent-Wurzel bei ihrer Löschung ausgenullt wurde – rekonstruieren zu können. Da Inodes mit ihrer Extent-Struktur im Journal dokumentiert werden, wenn sie gelöscht werden, stehen so die nötigen Informationen über die Inhaltsdaten der Datei zur Verfügung, können aber momentan einander nicht zugewiesen werden. Durch diese Zuweisung wäre es möglich, gelöschte Dateien im Bewusstein zu rekonstruieren, dass sie gelöscht sind.

6

LITERATURVERZEICHNIS

[1] *2GB Filesize Limit.* `http://linuxmafia.com/faq/VALinux-kb/2gb-filesize-limit.html`. Version: 2015. – Letzter Zugriff: 18.07.2015

[2] *Ext4 Disk Layout.* `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout`. Version: 2015. – Letzter Zugriff: 13.07.2015

[3] *List of file signatures.* `https://en.wikipedia.org/wiki/List_of_file_signatures`. Version: 2015. – Letzter Zugriff: 20.07.2015

[4] CARRIER, Brian: *File system forensic analysis*. Bd. 3. Addison-Wesley Reading, 2005

[5] CARRIER, Brian: The sleuth kit. In: *TSK). http://www. sleuthkit. org/sleuthkit/.[Online]* (2010)

[6] CARRIER, Brian: *Framework*. `http://www.sleuthkit.org/sleuthkit/framework.php`. Version: 2015. – Letzter Zugriff: 02.03.2015

[7] CARRIER, Brian: *The Sleuthkit TSK - Overview*. `http://www.sleuthkit.org/sleuthkit/`. Version: 2015. – Letzter Zugriff: 17.07.2015

[8] CASEY, Eoghan: *Digital evidence and computer crime: forensic science, computers and the internet*. Academic press, 2011

[9] CRAIGER, Philip: Recovering digital evidence from Linux systems. In: *Advances in Digital Forensics*. Springer, 2005, S. 233–244

[10] DIGITAL FORENSICS WIKI: *Extended File System*. `http://wiki.digital-forensic.org/index.php/Extended_File_System`. Version: 2015. – Letzter Zugriff: 09.07.2015

[11] FAIRBANKS, Kevin D.: An analysis of Ext4 for digital forensics. In: *Digital investigation* 9 (2012), S. S118–S130

[12] FAIRBANKS, Kevin D. ; LEE, Christopher P. ; OWEN III, Henry L.: Forensic implications of ext4. In: *Proceedings of the sixth annual workshop on cyber security and information intelligence research* ACM, 2010, S. 22

[13] FOSTER, Kristina: *Using distinct sectors in media sampling and full media analysis to detect presence of documents from a corpus*, Monterey, California. Naval Postgraduate School, Diss., 2012

[14] GARFINKEL, Simson L.: Carving contiguous and fragmented files with fast object validation. In: *digital investigation* 4 (2007), S. 2–12

[15] HAND, Scott ; LIN, Zhiqiang ; GU, Guofei ; THURAISINGHAM, Bhavani: Bin-Carver: Automatic recovery of binary executable files. In: *Digital Investigation* 9 (2012), S. S108–S117

[16] HOFHERR, Matthias: Post Mortem Analysen mit OpenSource Software. In: *IMF*, 2003, S. 0

[17] JAY SMITH, KLAYTON MONROE, ANDY BAIR: *Digital Forensics File Carving Advances*. `https://www.korelogic.com/Resources/Projects/dfrws_challenge_2006/DFRWS_2006_File_Carving_Challenge.pdf`. Version: 2006. – Letzter Zugriff: 16.07.2015

[18] JONES, Tim: Anatomy of ext4. In: *Retrieved at* (2009), S. 7

[19] 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*. Springer, 2012, S. 435–446

[20] KIM, Wook-Hee ; NAM, Beomseok ; PARK, Dongil ; WON, Youjip: Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split. In: *FAST*, 2014, S. 273–285

[21] LEE, Seokjun ; SHON, Taeshik: Improved deleted file recovery technique for Ext2/3 filesystem. In: *The Journal of Supercomputing* 70 (2014), Nr. 1, S. 20–30

[22] MAES, Bart: Comparison of contemporary file systems. (2012)

[23] MANSON, Dan ; CARLIN, Anna ; RAMOS, Steve ; GYGER, Alain ; KAUFMAN, Matthew ; TREICHELT, Jeremy: Is the open way a better way? Digital forensics using open source tools. In: *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on* IEEE, 2007, S. 266b–266b

[24] MATHUR, Avantika ; CAO, Mingming ; BHATTACHARYA, Suparna ; DILGER, Andreas ; TOMAS, Alex ; VIVIER, Laurent: The new ext4 filesystem: current status and future plans. In: *Proceedings of the Linux Symposium* Bd. 2 Citeseer, 2007, S. 21–33

[25] NARVÁEZ, Gregorio: Taking advantage of Ext3 journaling file system in a forensic investigation. In: *SANS Institute Reading Room* (2007)

[26] NELSON, Bill ; PHILLIPS, Amelia ; STEUART, Christopher: *Guide to computer forensics and investigations*. Cengage Learning, 2015

[27] PHILLIPS, Daniel: A directory index for ext2. In: *5th Annual Linux Showcase and Conference*, 2001, S. 173–182

[28] POISEL, Rainer ; TJOA, Simon ; TAVOLATO, Paul: Advanced file carving approaches for multimedia files. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 2 (2011), Nr. 4, S. 42–58

[29] POMERANZ, Hal: *EXT3 File Recovery via Indirect Blocks*

[30] POMERANZ, Hal: *SANS Digital Forensics and Incident Response Blog*. `https://digital-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems`. Version: 2008. – Letzter Zugriff: 14.07.2015

[31] POMERANZ, Hal: *SANS Digital Forensics and Incident Response Blog*. `https://digital-forensics.sans.org/blog/2011/03/14/digital-forensics-understanding-ext4-part-2-timestamps`. Version: 2011. – Letzter Zugriff: 14.07.2015

[32] POMERANZ, Hal: *SANS Digital Forensics and Incident Response Blog*. `https://digital-forensics.sans.org/blog/2011/03/28/`

digital-forensics-understanding-ext4-part-3-extent-trees. Version: 2011. – Letzter Zugriff: 07.07.2015

[33] RICHARD III, Golden ; ROUSSEV, Vassil ; MARZIALE, Lodovico: In-place file carving. In: *Advances in Digital Forensics III.* Springer, 2007, S. 217–230

[34] SAJJA, Abhilash: Forensic reconstruction of fragmented variable bitrate mp3 files. (2010)

[35] SLEUTHKIT WEBPAGE: *TSK Tool Overview*. http://wiki.sleuthkit.org/index.php?title=TSK_Tool_Overview. Version: 2015. – Letzter Zugriff: 17.07.2015

[36] SPENNEBERG, Ralf: *Carving-Tools spüren Files auf, ohne das Dateisystem zu kennen kennen [sic].* http://www.linux-magazin.de/Ausgaben/2008/06/Selbst-geschnitzt. Version: 2008. – Letzter Zugriff: 20.07.2015

[37] TS'O, Theodore: *The Linux ext2/3/4 Filesystem: Past, Present, and Future*

Lebenslauf

Persönliche Daten

Name:	Sabine Seufert
Anschrift:	Lohhofer Straße 4 90453 Nürnberg
Geburtsdatum:	14.01.1989
Geburtsort:	Nürnberg
Familienstand:	ledig

Ausbildung

1995 - 1999	Volksschule Reichelsdorf in Nürnberg - Grundschule -
1999 - 2009	Gymnasium Stein in Fürth - Allgemeinabitur -
2009 - 2013	Friedrich-Alexander-Universität - Studium der Informatik (Bachelor of Science)-
ab 2013	Friedrich-Alexander-Universität - Studium der Informatik (Master of Science)-

Berufstätigkeit

2011 - 2014	studentischer Hilfswissenschaftler Friedrich-Alexander-Universität

Erlangen, 1. September 2015