Technische Berichte in Digitaler Forensik

Herausgegeben vom Lehrstuhl für Informatik 1 der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) in Kooperation mit dem Masterstudiengang Digitale Forensik (Hochschule Albstadt-Sigmaringen, FAU, Goethe-Universität Frankfurt am Main)

Untersuchungen am Dateisystem HFS Plus

Lars Mechler

22.03.2016

Technischer Bericht Nr. 7

Zusammenfassung

Dieser Bericht enthält eine Analyse des Dateisystems HFS Plus unter Verwendung des Referenzmodells aus dem Buch "File System Forensik Analysis" von Brian Carrier. Die Dateisystemstrukturen von HFS Plus, die für die forensische Analyse bedeutsam sind, werden unter Einordnung in die Datenkategorien von Carrier detailliert behandelt. Anschließend wird für jede Kategorie ein Analyseszenario mitsamt den erforderlichen Analysetechniken vorgestellt.

Entstanden als Masterarbeit im Rahmen des Studiengangs Digitale Forensik unter der Betreuung von Felix Freiling und Victor Völzow.

Inhaltsverzeichnis

Abkürzungsverzeichnis

ACL	Access Control List
APM	Apple Partition Map
B-Tree	Balance-Tree
CNID	Catalog Node ID
DMG	Disk Image
DNS	Domain Name System
EFI	Extensible Firmware Interface
FTK	Forensic Toolkit
GMT	Greenwich Mean Time
GPT	GUID Partitionstabellen
GUID	Globally Unique Identifier
HFS	Hierarchical File System
LBA	Logical Block Addressing
LVM	Logical Volume Management
MBR	Master Boot Record
MDB	Master Directory Block
MFT	Master File Table
NTFS	New Technology File System
PLIST	Property List
PMBR	Protective Master Boot Record
TSK	The Sleuth Kit
UEFI	Unified Extensible Firmware Interface

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Seit Apple 2006 von Power PC-Technologie auf Intel umgestiegen ist und das iPhone 2007 eingeführt wurde, haben sich weltweit Apple Geräte auch im Consumer-Bereich stark verbreitet. Apple ist vom Nischenprodukt zu einem Marktführer avanciert.

Für die digitale Forensik bedeutet dies einen starken Anstieg von Untersuchungsaufträgen, bei denen Mac Systeme Untersuchungsgegenstand sind. HFS Plus als das von Apple bereits 1998 eingeführte Dateisystem ist das Standard-Dateisystem bei Mac Geräten mit Mac OS X sowie iOS. Die Anforderung für IT-Forensiker, HFS Plus Dateisystemstrukturen analysieren zu können, steigt stetig.

Brian Carrier hat 2005 mit dem Buch „File System Forensic Analysis“ ein Standardwerk geschaffen, indem er ein Referenzmodell entwickelt hat, das Dateisystemstrukturen in Kategorien einteilt, um die Dateisystemstrukturen von Dateisystemen wie FAT, NTFS und Ext verstehen und forensisch analysieren zu können. Das HFS Plus Dateisystem wurde hierbei jedoch nicht behandelt.

1.1 Zielsetzung

Ziel dieses Berichts ist es, das Kategorienmodell von Brian Carrier auf das HFS Plus Dateisystem anzuwenden, um so eine Technik für die forensische Analyse von HFS Plus spezifischen Spuren analog des Referenzmodells zu entwickeln.

1.2 Abgrenzung

Im Rahmen des Referenzmodells hat Brian Carrier die Unterscheidung von essentiellen und nicht-essentiellen Daten eingeführt.

Diese Abgrenzung ist in der Fachwelt nicht unumstritten. So haben Freiling und Dewald diese Begrifflichkeiten erweitert.[1] Im Rahmen ihrer Forschungsarbeit haben Freiling und Gruhn dies auf der *9th International Conference on IT Security Incident Management and IT Forensics* 2015 erneut thematisiert.[2]

Es wird hier nicht auf die Unterscheidung zwischen essentiellen und nicht-essentiellen Daten bei den Dateisystemstrukturen eingegangen, da es den Rahmen dieser Arbeit sprengen würde.

1.3 Vorgehensweise

Bevor näher auf das Dateisystem HFS Plus und das Referenzmodell von Brian Carrier zur Dateisystemanalyse eingegangen werden kann, sollen einige Grundlagen angesprochen und

[1] Dewald und Freiling 2011, S. 42–43
[2] Freiling und Gruhn 2015, S. 40–48

Begriffe definiert werden, die als Basis für die digitale Forensik und diese Arbeit von Bedeutung sind.

Im darauffolgenden Abschnitt wird auf die Datenträgeranalyse eingegangen, die Voraussetzung für das Verständnis einer Partitionsstruktur bei einem HFS Plus Dateisystem ist.

Im weiteren Verlauf wird das Referenzmodell der Dateisystemkategorien vorgestellt und die generellen Analysetechniken für jede Kategorie nach Brian Carrier ausführlich dargestellt.

Als Schwerpunkt werden die technischen Grundlagen und für die forensische Analyse relevanten Datenstrukturen von HFS Plus behandelt. Diese werden in das Referenzmodell nach den Dateisystemkategorien eingeordnet. Anhand von erstellten Testdaten werden Analyse-Szenarien für Dateisystemstrukturen jeder Kategorie detailliert aufgezeigt, für die anschließend forensische Analysetechniken erarbeitet wurden.

Die Erstellung von Beispielszenarien und deren Untersuchungen und Analyse wurden anhand folgender Testumgebung durchgeführt:

Als Mac Plattform diente ein MacBook Pro Retina, i7, 16 GB RAM, 256 GB SSD, Mac OS X 10.11 (El Capitan) mit für die Analyse genutzten Komponenten:

- Xcode 6.4
- Paragon NTFS for Mac 10.10.2
- Mac Fuse for Mac 2.8.0
- MacPorts 2.3.3
- The Sleuthkit 4.1.3
- FileXray 1.5.0
- BlackLight 2015 Release 3.1

Weiterhin wurde ein Windows 10 Pro 64 Bit System in einer virtuellen Maschine mit VMWare Fusion 8.0.0 ausgeführt. Hier wurden folgende Komponenten für die Tests und Analyse genutzt:

- EnCase 7.11.01
- X-Ways Forensics 18.4
- Forensic Toolkit 6.0.1
- Triforce AHJP HFS+ Journal Parser

Für die Erstellung der Analyse-Szenarien wurden Dateimanipulationen mit dem selbstgeschriebenen Python Skript „data.py“ durchgeführt. Der Quelltext befindet sich im Anhang A.1.

Für die Analyse der Beispielszenarien und für die Veranschaulichung der Datenstrukturen wurden Übungsimages erstellt. Eine Übersicht von allen Images können Anhang A.2 entnommen werden.

1.4 Stand der Technik

Grundlage für die Ausarbeitung der HFS Plus Dateisystemstrukturen legte Amit Singh. Diese Thesis stützt sich auf das umfangreiche Werk „Mac OS X Internals - A Systems Approach“[3] von Singh aus dem Jahr 2007. In der Apple Entwicklerdokumentation „Technical Note 1150“[4] sind die grundlegenden Dateisystemstrukturen ebenfalls gut dokumentiert.

Aktuellere Fachliteratur, die sich mit Mac OS X Komponenten auseinandersetzt, ist unter anderem das Buch „Mac OS X and iOS Internals – To the Apple‘s Core“[5] von Jonathan Levin aus dem Jahr 2013.

Es existiert jedoch kaum Fachliteratur, die sich aus forensischer Sicht mit HFS Plus intensiv auseinandersetzt. Aaron Burghardt und Adam J. Feldman haben mit ihrer Veröffentlichung von „Using the HFS+ journal for deleted file recovery“[6] bei Elsevier 2008 die Wiederherstellung von gelöschten Dateien anhand der Analyse des HFS Plus Journals wissenschaftlich erörtert. Dieses White Paper wurde hier für die Ausarbeitung der Dateikategorie Anwendung herangezogen.

Ryan Kubasiak ist einer der führenden Autoren eines der ersten Forensik Fachbücher, das sich ausschließlich mit Mac Forensics beschäftigt.[7] Kubasiak hat bis Anfang 2015 eine der populärsten Mac Forensics Internetseiten betrieben.[8]

Das Thema Mac Forensics rückte in jüngster Vergangenheit beim SANS Institute[9] immer mehr in den Fokus. So wurde dort von Sarah Edwards der Mac Forensics Kurs „FOR 518 – Mac Forensic Analysis“ entwickelt. Sie ist außerdem Autorin des Blogs „mac4n6“[10].

Steve Whalen ist einer der Pioniere in Sachen Mac Forensics. Er ist Firmengründer von Sumuri LLC[11] und Entwickler der erfolgreichen Trainingsserie „Mac Forensics Survival Course“. Zudem hat Whalen die automatisierte Mac Analysesoftware RECON und die auf Linux basierende Boot-CD „Paladin“ mit zahlreichen Funktionen für die Mac Analyse entwickelt.

Erwähnenswert ist außerdem die Firma BlackBag Technologies[12], die sich ebenfalls schwerpunktmäßig mit Mac Forensics auseinandersetzt. Das Unternehmen hat Analyse- sowie Sicherungssoftware für Mac Systeme herausgebracht und bietet Fortbildungen in diesem Bereich an.

Auch Guidance Software[13] als quasi Standard kommerzieller Forensik-Software hat bei der Implementierung der Analyse von Mac Artefakten nachgelegt - nicht zuletzt durch den Guidance Entwickler Simon Key, der den Guidance Macintosh Kurs „EnCase Examinations of

[3] Singh 2007
[4] Apple: *Technical Note TN1150*
[5] Levin 2013
[6] Burghardt und Feldman 2008
[7] Varsalone et al. 2009
[8] www.appleaxaminer.com, aufgerufen am 02.02.2016
[9] www.sans.org, aufgerufen am 02.02.2016
[10] www.mac4n6.com, aufgerufen am 02.02.2016
[11] www.sumuri.com, aufgerufen am 02.02.2016
[12] www.blackbagtech.com, aufgerufen am 02.02.2016
[13] www.guidancesoftware.com, aufgerufen am 02.02.2016

Macintosh Operating Systems“ entwickelte und viele EnScripts für die Analyse von Mac Artefakten programmierte.

An dieser Stelle soll ein Überblick über forensische Software, die auch Mac Dateisystem Artefakte unterstützt, gegeben werden.

EnCase 7.11.01 von Guidance Software

EnCase Forensics ist vor allem in Nordamerika weit verbreitet, aber auch in Deutschland sehr bekannt. Eine Besonderheit ist die integrierte Skriptsprache EnScript, mit der eine Automatisierung und Anpassung von Arbeitsabläufen möglich ist. EnCase kann HFS Plus Dateisystemstrukturen interpretieren und bietet durch die Anwendung von EnScripts detaillierte Analysemöglichkeiten für HFS Plus Artefakte. Es enthält eine Fallverwaltung mit Backupmöglichkeiten und eine komplexe Reportfunktionalität.

X-Ways Forensics 18.4 von X-Ways Technology AG

X-Ways Forensics ist ein von X-Ways Software Technology AG entwickeltes Werkzeug, das auf dem Hex-Editor Winhex basiert. Es enthält z. B. eine Fallverwaltung, automatische Protokollierung der Arbeitsschritte und Erstellung von Berichten im HTML-Format.

Es unterstützt unter anderem auch HFS Plus Dateisysteme und kann viele Datenstrukturen dieses Dateisystems interpretieren.

Forensic Toolkit (FTK) 6.0.1 von AccessData

FTK bietet eine komplette Analyseumgebung für Unix- und Windows-Dateisysteme. Es kann HFS Plus Dateisystemstrukturen weitestgehend analysieren. Analyseergebnisse werden in einer zentralen Datenbank gespeichert. Eine Fall- und Benutzerverwaltung ist ebenfalls enthalten.

BlackLight R3.1 von BlackBag Technologies

BlackLight R3.1 ist eine Forensik-Plattform mit dem Schwerpunkt Mac Forensics. Es sind Mac- sowie Windowsversionen von BlackLight erhältlich. Neben einer umfangreichen Unterstützung von HFS Plus Dateisystemstrukturen ist ebenfalls eine Fallverwaltung integriert.

Triforce AHJP HFS+ Journal Parser

Das von G-C Partners LLC entwickelte Triforce AHJP HFS+ Journal Parser sucht im HFS Plus Journal nach Catalog Dateieinträgen und speichert die Ergebnisse in CSV-Dateien und in einer SQLite Datenbank.

FileXray 1.5.0

FileXray wurde von Amit Singh, Autor des Buches „Mac OS X Internals – A Systems Approach“, entwickelt. Es ist kommandozeilenbasierend und ermöglicht die forensische Analyse von HFS Plus Dateisystemstrukturen. Der letzte Versionsstand ist aus dem Jahr 2011.

The Sleuthkit (TSK) 4.1.3

The Sleuthkit[14] ist eine Open Source-Entwicklung von Brian Carrier für die Analyse verschiedener Dateisysteme. Es unterstützt neben FAT, NTFS und Ext unter anderem auch HFS Plus. Durch eine Sammlung kommandozeilenbasierender Tools können auf unterschiedlichen Ebenen, die dem von Brian Carrier entwickelten Kategorienmodell entsprechen, die Dateisysteme analysiert werden.[15]

Die wesentlichen Tools werden nachfolgend unter Angabe der zugehörigen Kategorie aufgelistet:

- Kategorie Datenträger
  - *mmls*
    - ➔ analysiert die Partitionsstruktur
- Kategorie Dateisystem
  - *fsstat*
    - ➔ analysiert Dateisysteminformationen
- Kategorie Inhalt
  - *blkcat*
    - ➔ analysiert den Inhalt eines Blocks
  - *blkstat*
    - ➔ analysiert den Status der Allokation eines Blocks
- Kategorie Metadaten
  - *istat*
    - ➔ analysiert Metadateneinträge einer Datei oder eines Verzeichnisses
  - *icat*
    - ➔ analysiert den Inhalt von allokierten Blöcken anhand der Metadateninformationen einer Datei
  - *ifind*
    - ➔ zeigt Metadaten die mit einem analysierten Block verknüpft sind
- Kategorie Dateinamen
  - *fls*
    - ➔ listet Datei- und Verzeichnisnamen
  - *ffind*
    - ➔ findet den mit den analysierten Metadaten verknüpften Dateinamen

Autopsy Version 2

Bei Autopsy Version 2 handelt es sich um eine forensische Open Source-Forensik-Plattform, die ebenfalls von Brian Carrier entwickelt wurde und das grafische Interface für TSK darstellt. Es hat einen erweiterten Funktionsumfang, wie zum Beispiel eine Fallverwaltung, Mehrbenutzer- und Reportfähigkeiten.

[14] www.sleuthkit.org, aufgerufen am 02.02.2016
[15] Geschonneck und Kraus 2014, S. 257

2. Begriffsdefinitionen

2.1 Datenanalyse

Ein Computersystem kann in unterschiedliche Schichten untergliedert werden, die beginnend bei dem physikalischen Aufbau bis hin zur Anwendungsschicht unterschiedliche Aspekte des Mediums berücksichtigen. Dieses Schichtenmodell kann dafür verwendet werden, die unterschiedlichen Analysemethoden aufzuzeigen.

Auf der niedrigsten Schicht ist die physikalische Analyse von Speichermedien eingetragen, also beispielsweise das Auslesen eines magnetischen Datenträgers mittels spezieller Einrichtungen in einem Reinraum. Im Weiteren wird davon ausgegangen, dass das betrachtete Speichermedium auf physischer Ebene einen Datenstrom liefert, sodass diesbezüglich keine Analysemethoden benötigt werden.[16]

2.2 Datenträgeranalyse

Datenträger, die für nicht-flüchtige Datenspeicherung verwendet werden (z.B. Festplatten, Flashspeicher) sind in Volumes und Partitionen organisiert. Diese Begrifflichkeiten werden von Betriebssystemen unterschiedlich verwendet und können mitunter für Verwirrung sorgen. So ist bei Wikipedia der Begriff „Partition“ als ein zusammengesetzter Teil des Speicherplatzes eines geeigneten physischen oder logischen Datenträgers definiert.[17]

Volumes können in mehrere Partitionen geteilt oder auch aus mehreren Volumes zusammengesetzt und anschließend partitioniert werden.[18] Mit logischen Volume Managern, wie zum Beispiel bei Microsoft dem Logical Volume Management (LVM), können mehrere Partitionen zu logischen Volume Gruppen zusammengefasst werden. Innerhalb dieser logischen Volume Gruppen können dann neue Partitionen (logical Volumes) angelegt werden. Auch bei Apple existiert dieses Konzept unter der Bezeichnung CoreStorage. Es wird für die Nutzung von *FileVault II* als Datenträgervollverschlüsselung[19] und für die Umsetzung von *FusionDrive*[20] verwendet.

Bei der Datenträgeranalyse wird ein Datenträger beispielsweise daraufhin untersucht, wo sich welche Dateisysteme befinden, wo sich der Boot-Code befindet und welche versteckten Datenbereiche auf dem Datenträger existieren.

2.3 Dateisystemanalyse

Ein Dateisystem ist eine Sammlung von Datenstrukturen, die es ermöglicht, Daten zu erstellen, zu lesen und zu schreiben. Ziel der Dateisystemanalyse ist es, Dateien zu finden, gelöschte

[16] Carrier 2005, S. 10
[17] Wikipedia: *Partition (Datenträger)*
[18] Carrier 2005, S. 10
[19] Die Bezeichnung Datenträgervollverschlüsselung ist nicht ganz korrekt, da hier lediglich das logische Volume, das mit CoreStorage erstellt wurde, verschlüsselt wird.
[20] Bei FusionDrive wird ein logisches Volume aus einer SSD und einer HDD erstellt. Diese Technik führte Apple 2012 ein, um Performance und Speicherkapazität zu erhöhen.

Dateien wiederherzustellen und versteckte Daten zu lokalisieren. Im Ergebnis können Dateiinhalte, Dateifragmente und Metadaten mit Dateien assoziiert werden.[21]

Nachfolgende Abbildung zeigt das Schichtenmodell der Datenanalyse von Carrier. Im Rahmen dieser Arbeit liegt der Schwerpunkt auf Dateisystemanalyse, wobei auch auf die Schicht Datenträgeranalyse eingegangen wird.

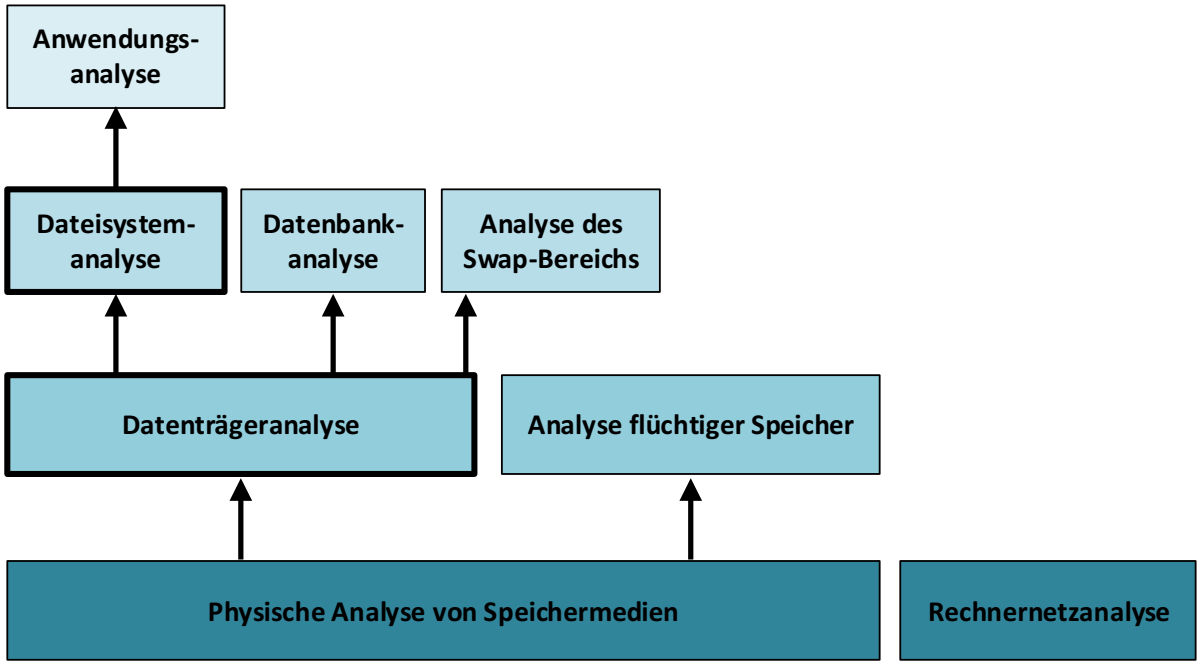


Abbildung 1: Schichtenmodell der Datenanalyse nach Carrier 2005

[21] Carrier 2005, S. 11

3. Datenträgeranalyse

Grundsätzlich ist das konventionelle Speichermedium für Computersysteme die Festplatte. Auf der Hardware Ebene sind diese Speichermedien in Sektoren unterteilt. Eine typische Festplatte enthält logische Sektoren mit der jeweiligen Größe von 512 Bytes. Um die Sektoren adressieren zu können, wird zum einen die Cylinder-Head-Sector Adressierung (CHS-Adressierung) und das zunehmend bevorzugte Logical Block Addressing (LBA) verwendet, bei dem adressierbarer Speicher des Speichermediums in linearen zusammenhängenden Sektoren abgebildet wird.

Bei der LBA Adressierung beginnt die Adresse des ersten logischen Sektors einer Festplatte bei 0. Also hätte der erste logische Sektor dieser Festplatte die physische Adresse 0. Zu unterscheiden ist zwischen der physischen Adresse mit LBA Adressierung und der logischen Partitionsadresse. Die LBA Adresse ist relativ zum Beginn des gesamten Volumes, die logische Partitionsadresse dagegen ist relativ zum Beginn der Partition.

Liegen logische Sektoren außerhalb von Partitionen, zum Beispiel in Bereichen von Partitionslücken, haben sie lediglich eine LBA Adresse.

In einem Mac System werden Volumes grundsätzlich als Blockgerät mit *disk* dargestellt und im Verzeichnis */dev/* gelistet. *disk* bezeichnet dabei das gesamte Volume. Partitionen werden als „slice“[22] vom Volume aufgeführt: */dev/disk#s#.* Folgende Abbildung zeigt die Auflistung aller Volumes und Partitionen im Terminal. Diese Informationen können mit dem Befehl *diskutil list* erhoben werden. Es werden drei Volumes aufgelistet. Die Volumes *disk0* und *disk2* haben jeweils mehrere Partitionen. Besonderes Augenmerk gilt dem Volume *disk1*. Hier gibt das Terminal die Information aus, dass es sich um ein logisches Volume handelt und verweist auf die Partition *disk0s2*. Hier ist erkennbar, dass es sich um ein Apple CoreStorage Volume handelt, den zuvor erwähnten Logischen Volume Manager von Apple. *disk1* ist also ein logisches Volume, dass aus der zweiten Partition des Volumes *disk0* erstellt wurde. Physikalisch sind zwei Datenträger am System angeschlossen: *disk0* und *disk2*.

```
MacBook:images_final Lars$ diskutil list
/dev/disk0 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        *251.0 GB   disk0
   1:                        EFI EFI                     209.7 MB   disk0s1
   2:          Apple_CoreStorage Macintosh HD            250.1 GB   disk0s2
   3:                 Apple_Boot Recovery HD             650.0 MB   disk0s3
/dev/disk1 (internal, virtual):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:                  Apple_HFS Macintosh HD           +249.8 GB   disk1
                                 Logical Volume on disk0s2
                                 5C22D09A-8F93-4489-8EE3-842E2CB0713B
                                 Unlocked Encrypted
```

Abbildung 2: Übersicht Volumes und Partitionen

Exkurs:

Bei der Datenträgeranalyse im Zusammenhang mit Apple CoreStorage ist zu beachten, dass die Windows-basierten Forensik-Programme CoreStorage Volumes nicht interpretieren können. Deshalb ist es vor der Analyse notwendig, das CoreStorage Volume in einem Mac OS

[22] In Unix-Systemen wird die Bezeichnung „slice“ für Partition verwendet.

X System mit CoreStorage Treiber zusammenzusetzen und eine Sicherung des Volumes anzufertigen oder eine Sicherung in einem CoreStorage System einzubinden und dann die Analyse durchzuführen.

3.1 Partitionsschemen

Das aktuelle Apple Betriebssystem Mac OS X unterstützt drei Partitionstypen: Master Boot Record (MBR), Apple Partitionen und GUID Partitionstabellen (GPT).

3.1.1 Master Boot Record

Apple Mac Systeme mit OS X können nicht von MBR-Partitionen booten, können aber einen MBR enthalten. Sie werden aus Kompatibilitätsgründen mit der Bezeichnung „Protective MBR“ auf HFS PLUS Dateisystemen im ersten Sektor erstellt, wenn GPT als Partitionsschema genutzt wird. DOS Partitionen sind nicht Inhalt dieser Arbeit, weshalb im Weiteren nicht darauf eingegangen wird.

3.1.2 Apple Partitionen

Das Partitionsschema der Apple Partition Map (APM) wurde mit dem Macintosh II im Jahr 1987 eingeführt.[23] [24] Es unterscheidet sich erheblich von DOS oder GPT Partitionen.

APM ist ein 32 Bit Schema und war bis zur Einführung von Intel-basierten Mac Rechnern das Standard-Partitionsschema bei Macintosh Systemen.

Die bis dahin Power PC basiert entworfenen Macintosh Rechner können nur von Partitionen mit dem Apple Partitionsschema booten. Mit der Einführung von Intel-basierten Macs wurde APM von GPT als Standard für die Partitionierung von Mac Speichermedien abgelöst. Jedes Mac Betriebssystem kann jedoch Speichermedien mit APM lesen. Intel basierte Macintosh Rechner mit Mac OS X können theoretisch auch von Apple Partitionen booten, unterstützt wird von Apple jedoch nur das Booten von Mac OS X von Volumes mit GPT.

Apple Partitionen können so auf aktuellen Systemen (z. B. Mac OS X El Capitan) oder auf älteren Mac OS 9 sowie auf iPods und Apple Disk Images vorgefunden werden.[25]

[23] Wikipedia: *Apple Partition Map*
[24] Wikipedia: *Macintosh II*
[25] Carrier 2005, S. 101

Abbildung 3 zeigt das Layout von Volumes mit Apple Partitionen:

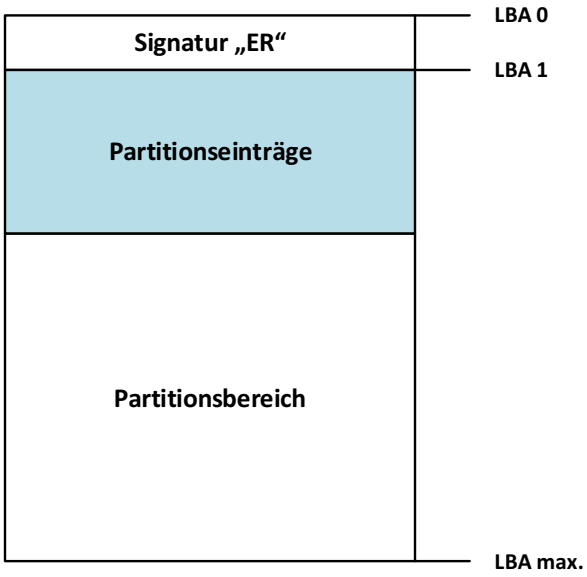


Abbildung 3: Apple Partition Map Struktur

Datenstrukturen in Apple Partitionstabellen werden im Big-Endian-Format gespeichert. In Abbildung 4 wird die Disk-Struktur eines Beispielimages „image_1“[26] mit dem Mac Terminal-Befehl *hdiutil* analysiert:

```
MacBook:Images Lars$ hdiutil partition image_1.dmg
scheme:     Apple
block size: 2
_ ## Type_____________________ Name_____________________ Start___ Size____
+    DDM                      Driver Descriptor Map        0         1
    1 Apple_partition_map     Apple                        1        63
    2 Apple_HFS               disk image                  64     19528
+    Apple_Free                                        19592         4

+ synthesized
```

Abbildung 4: hdiutil-Ausgabe eines Disk Images mit APM Partitionierung

Die Ausgabe zeigt alle Partitionen inkl. der Position der Apple Partitionstabelle. Noch vor der Partitionstabelle im Sektor 0 befindet sich eine Device Descriptor Tabelle mit einer Länge von 18 Byte. Sie beginnt immer mit der Signatur „ER“ (0x4552) und beschreibt unter anderem die Sektorengröße und die Anzahl der Sektoren auf dem Volume.[27]

Folgende Tabelle zeigt die Struktur der Device Descriptor Tabelle:

Offset (Byte)	Länge (Byte)	Beschreibung
0	2	Signatur „ER“ (0x4552)
2	2	Sektorengröße in Byte
4	4	Anzahl der Sektoren
8	8	reserviert
16	2	Anzahl der noch folgenden Driver Descriptor

Tabelle 1: Device Descriptor Tabellen Einträge

[26] Das Image „image_1“ wurde für die Analyse mit *xmount* als Apple Disk Image (dmg) eingebunden.
[27] Apple: *IOApplePartitionScheme.h*

Mit *dd* lässt sich im Mac Terminal die Device Descriptor-Tabelle auswerten:

```
MacBook:Images Lars$ dd if=image_1.dmg | xxd -l 18
0000000: 4552 0200 0000 4c8c 0000 0000 0000 0000  ER....L.........
0000010: 0000                                     ..
```

Abbildung 5: APM ER-Signatur, dd im Mac Terminal

Im Offsetbereich 0-1 des Volumes steht die *ER*-Signatur und signalisiert damit den Beginn eines Volumes mit APM. Die nächsten zwei Bytes repräsentieren die Sektorengröße von 512 Bytes (0x0200), die darauffolgenden vier Bytes geben hier die Anzahl der Sektoren des Volumes in Höhe von 19596 (0x00004C8C) an.

Im zweiten Sektor des Volumes beginnen dann die Partitionseinträge mit jeweils einer Länge von 512 Bytes.[28] Die Abbildung 6 zeigt die ersten 112 Bytes des ersten Partitionseintrags vom Beispielimage. Es ist zu erkennen, dass die Apple Partitionstabelle selbst eine Partition ist. Jeder Partitionseintrag beginnt mit der Signatur *PM* (*0x504d)* am Offset 0 mit einer Länge von 2 Bytes. Offset 4 bis 7 enthält die Anzahl der Partitionen. Im dargestellten Disk Image existieren 2 Partitionen. Die darauffolgenden 4 Bytes geben den Start Sektor der Partition an, der hier bei Sektor 1 liegt. Im Offset 12 bis 15 kann die Größe der Partition in Sektoren entnommen werden, die hier 63 (0x003f) beträgt. Den anschließenden 32 Bytes ist die Partitionsbezeichnung *Apple* zu entnehmen. Die nächsten 32 Bytes sind für den Partitionstyp reserviert. In diesem Fall ist es *Apple_partition_map*.

```
MacBook:Images Lars$ dd if=image_1.dmg bs=512 skip=1 | xxd -l 112
0000000: 504d 0000 0000 0003 0000 0001 0000 003f  PM.............?
0000010: 4170 706c 6500 0000 0000 0000 0000 0000  Apple...........
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000030: 4170 706c 655f 7061 7274 6974 696f 6e5f  Apple_partition_
0000040: 6d61 7000 0000 0000 0000 0000 0000 0000  map.............
0000050: 0000 0000 0000 003f 0000 0003 0000 0000  .......?........
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Abbildung 6: Auszug aus Apple Partition Map Eintrag, dd

Die vollständigen Datenstrukturen können den folgenden Tabellen 2 und 3 entnommen werden.[29]

Offset (Byte)	Länge (Byte)	Beschreibung
0	2	Signatur „PM“ (0x504D)
2	2	reserviert
4	4	Anzahl der Partitionen
8	4	Start Sektor der Partition
12	4	Größe der Partition in Sektoren
16	32	Name der Partition in ASCII
48	32	Partitionstyp in ASCII
80	4	Start Sektor des Datenbereichs der Partition
84	4	Größe des Datenbereichs der Partition in Sektors
88	4	Partitionsstatus (siehe nächste Abbildung)
92	4	Start Sektor des Boot Codes
96	4	Größe des Boot Codes Sektors

[28] Carrier 2005, S. 103
[29] Carrier 2005, S. 103–104

100	4	Adresse des Boot Loaders
104	4	reserviert
108	4	Eintrag Boot Code
112	4	reserviert
116	4	Prüfsumme Boot Code
120	16	Prozessortyp
136	376	reserviert

Tabelle 2: APM Partitionseinträge

Am Offset 88 der Partitionseinträge kann der Partitionsstatus erhoben werden. Folgende Werte sind möglich:

Typ	Beschreibung
0x00000001	Eintrag ist gültig (A/UX only)
0x00000002	Eintrag ist allocated (A/UX only)
0x00000004	Eintrag wird benutzt (A/UX only)
0x00000008	Eintrag enthält Bootinformationen (A/UX only)
0x00000010	Partition ist lesbar (A/UX only)
0x00000020	Partition ist beschreibbar (Macintosh & A/UX)
0x00000040	Boot Code ist positionsunabhängig ((A/UX only)
0x00000100	Partition enthält kompatible (chain) Treiber (Macintosh only)
0x00000200	Partition enthält reale Treiber (Macintosh only)
0x00000400	Partition enthält chain Treiber (Macintosh only)
0x40000000	Partition wird automatisch eingebunden beim Systemstart (Macintosh only)
0x80000000	Startpartition (Macintosh only)

Tabelle 3: Werte für den Partitionsstatus

3.1.3 GPT Partitionen

Als Nachfolger des MBR Partitionsschemas löste das Globally Unique Identifier Konzept (GUID) MBR als modernen Standard ab. Es wurde als Teil der Extensible Firmware Interface (EFI) Spezifikation von Intel entwickelt und ist nun in Unified Extensible Firmware Interface (UEFI) übergegangen. UEFI wird jetzt vom UEFI Forum organisiert.[30] [31]

Standard Partitionsschema ist auch beim Apple Dateisystem HFS Plus GPT.

GPT unterstützt bis zu 128 Partitionen und nutzt eine 64 Bit LBA Adressierung.[32]

3.1.3.1 GPT Disk Layout

Der erste Sektor eines GPT Volumes (LBA 0) enthält einen Master Boot Record mit einem Partitionseintrag, der sich am Offset 446 befindet und 16 Bytes lang ist. Der Eintrag hat als Partitionstyp-Signatur den Wert *0xEE*. Es handelt sich hierbei um einen Protective MBR (PMBR), der verhindern soll, dass Systeme, die nicht mit GPT umgehen können, das GPT Volume beschädigen.[33]

[30] Intel: *UEFI*
[31] Unified Extensible Firmware Interface Forum: *UEFI*
[32] Carrier 2005, S. 139
[33] Apple: *Technical Note TN2166*

Nach dem PMBR folgt der primäre GPT Header in LBA 1. Ein sekundärer GPT Header ist als Backup im letzten LBA des GPT Volumes gespeichert.

Von LBA 2 bis LBA 33 befinden sich die jeweils 128 Bytes großen Partitionseinträge. Ab LBA 34 beginnt der Datenbereich für die jeweiligen Partitionen. Jeweils eine Kopie der Partitionseinträge befindet sich am Ende des Volumes vor dem sekundären GPT Header.

Folgende Abbildung veranschaulicht den Aufbau eines Volumes mit GPT Struktur:

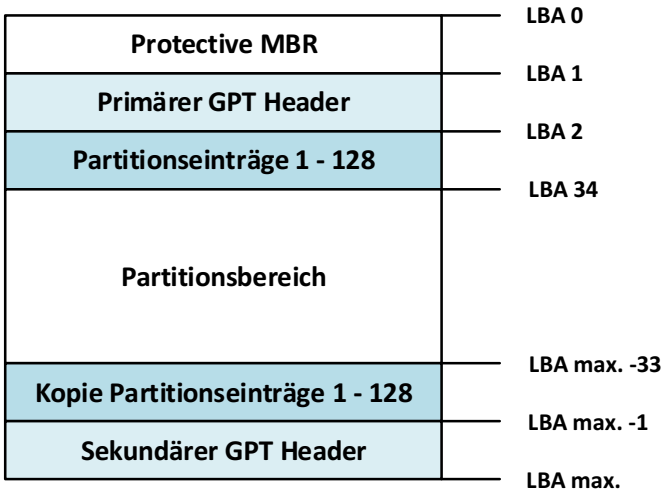


Abbildung 7: GUID Partitionsstruktur

Die folgenden Tabellen beinhalten den Aufbau des GPT Headers, den Aufbau der Partitionseinträge, eine Übersicht über GUIDs, die für Apple Systeme relevant sind sowie mögliche Attribute für Partitionseinträge.[34] [35]

Aufbau des GPT Headers:

Offset (Byte)	Länge	Beschreibung
0	8 Bytes	Signatur „EFI PART“ (0x4546492050415254)
8	4 Bytes	Version (0x00000100)
12	4 Bytes	GPT-Header Größe in Bytes (0x5C000000) (Little-Endian[36])
16	4 Bytes	GPT-Header-CRC32 Prüfsumme
20	4 Bytes	Reserviert
24	8 Bytes	LBA des primären GPT Headers (Little-Endian)
32	8 Bytes	LBA des sekundären GPT Headers (Little-Endian)
40	8 Bytes	LBA vom Beginn des Partitionsbereichs (Little-Endian)
48	8 Bytes	LBA vom Ende des Partitionsbereichs (Little-Endian)
56	16 Bytes	GUID des Datenträgers (Die ersten drei Blöcke der GUID werden in Little-Endian gespeichert.)
72	8 Bytes	Start LBA der Partitionstabelle (Little-Endian)
80	4 Bytes	Anzahl der Partitionseinträge (Little-Endian)

[34] Apple: *Technical Note TN2166*
[35] Carrier 2005, S. 141–143
[36] Bei Little-Endian wird das kleinstwertige Byte an der Anfangsadresse gespeichert, d. h. das höchstwertigste Byte an der höchstwertigsten Adresse.

84	4 Bytes	Größe der einzelnen Partitionseinträge (grundsätzlich 128 Bytes)
88	4 Bytes	CRC32 Prüfsumme der Partitionstabelle
92	bis LBA Ende	Reserviert

Tabelle 4: GPT Header Struktur

Aufbau der GPT Partitionseinträge:

Offset (Byte)	Länge	Beschreibung
0	16 Bytes	GUID vom Partitionstyp (Die ersten drei Blöcke der GUID sind in Little-Endian gespeichert.)
16	16 Bytes	Einzigartige Partitions-GUID (Die ersten drei Blöcke der GUID sind in Little-Endian gespeichert.)
32	8 Bytes	Erster LBA der Partition (Little-Endian)
40	8 Bytes	Letzter LBA der Partition (Little-Endian)
48	8 Bytes	Partitionsattribute
56	72 Bytes	Partitionsname in Unicode

Tabelle 5: Struktur der GPT Partitionseinträge

Format und Werte für den Partitionstyp, der durch einen GUID Wert bestimmt wird, kann der Tabelle 6 entnommen werden:

GUID Wert	Beschreibung
48465300-0000-11AA-AA11-00306543ECAC	HFS (Plus)
52414944-0000-11AA-AA11-00306543ECAC	Apple Raid Partition
52414944-5F4F-11AA-AA11-00306543ECAC	Apple Raid Partition offline
53746F72-6167-11AA-AA11-00306543ECAC	Apple CoreStorage Partition
5265636F-7665-11AA-AA11-00306543ECAC	Recovery Partition von Apple TV der ersten Generation (wurden mit Mac OS X Tiger betrieben)
C12A7328-F81F-11D2-BA4B-00A0C93EC93B	EFI System
55465300-0000-11AA-AA11-00306543ECAC	Apple UFS
4C616265-6C00-11AA-AA11-00306543ECAC	Apple Label

Tabelle 6: GPT Partitionstypen

Mögliche Attribute, die im Offsetbereich 48-55 der Partitionseinträge aus Tabelle 5 gesetzt sein können:

Attribut-Wert	Beschreibung
0	Systempartition
1	vor EFI verstecken
2	Legacy BIOS bootfähig
60	Nur lesen
62	Versteckt
63	Nicht automatisch einbinden

Tabelle 7: Attribute für GPT Partitionseinträge

Apple dokumentiert die Implementierung von GPT in der *Apple Technical Note TN2166*. Wichtig zu wissen ist, dass Apple für alle Partitionen eine 4096 Byte Grenze vorsieht, um den Anforderungen von HFS Plus gerecht werden zu können.[37]

Zum Beispiel beginnt grundsätzlich die erste Partition bei einer EFI Partitionierung mit HFS Plus im LBA 40. Wie zuvor dargestellt, nehmen die GPT Partitionseinträge LBA 2 bis LBA 33 ein (Ein Partitionseintrag von 128 Bytes * 128 Einträge / 512 LBA-Größe = 32). Zusammen mit dem PMBR und GPT Header sind lediglich die ersten 34 LBA belegt. Theoretisch könnte man annehmen, dass die erste Partition im LBA 34 beginnen könnte. Multipliziert man LBA 34 mit der Sektorengröße von 512 Bytes kommt man auf 17408 Bytes. Teilt man nun das Ergebnis durch 4096, bekommt man als Resultat 4,25. Das bedeutet, dass LBA 34 nicht an der 4096 Byte Grenze beginnt. Um den ersten LBA zu bestimmen, der nach LBA 34 an der 4096 Byte Grenze startet, berechnet man 4096*5/512=40. LBA 40 ist der erste LBA an der 4096 Byte Grenze und der Start Sektor der HFS Plus Partition.

Apple macht das Partitionslayout für GPT von der Volume Größe abhängig. Ist das Volume kleiner als 1 GB, werden keine extra Partitionen erstellt. Liegt die Größe zwischen 1 GB und 2 GB, werden nach jeder Partition 128 MB frei gelassen. Ab einer Größe von 2 GB wird eine 200 MB große EFI Systempartition als erste Partition erstellt und nach jeder Partition außer der EFI Partition 128 MB freier Speicher eingerichtet.[38]

Nach der *Apple Technical Note 2166* wird die EFI Systempartition von Mac Systemen nicht genutzt.

3.1.4 Analysemöglichkeiten

Die *hdiutil*-Ausgabe vom Beispielimage „image_2“ zeigt Informationen über ein Layout mit GPT:

```
MacBook:Images Lars$ hdiutil partition image_2.dmg
scheme:     GUID
block size: 512
_ ## Type_____________________ Name_____________________ Start___ Size____
+    MBR                      Protective Master Boo        0         1
+    Primary GPT Header       GPT Header                   1         1
+    Primary GPT Table        GPT Partition Data           2        32
+    Apple_Free                                           34         6
   1 Apple_HFS                disk image                   40    195280
+    Backup GPT Table         GPT Partition Data       195320        32
+    Backup GPT Header        GPT Header               195352         1

+ synthesized
```

Abbildung 8: hdiutil-Ausgabe eines Volumes mit GPT

Mit *dd* kann der LBA 1 aufgerufen werden, um den Header zu analysieren, der direkt nach dem PMBR folgt:

```
MacBook:mnt Lars$ dd if=image_2.dmg bs=512 skip=1 | xxd -l 512
0000000: 4546 4920 5041 5254 0000 0100 5c00 0000  EFI PART....\...
0000010: 7742 60e0 0000 0000 0100 0000 0000 0000  wB`.............
0000020: 18fb 0200 0000 0000 2200 0000 0000 0000  ........".......
0000030: f7fa 0200 0000 0000 8275 0c55 620a 3447  .........u.Ub.4G
0000040: 9035 944e dc38 77ea 0200 0000 0000 0000  .5.N.8w.........
0000050: 8000 0000 8000 0000 516d 6aa1 0000 0000  ........Qmj.....
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

[37] Apple: *Technical Note TN2166*
[38] Apple: *Technical Note TN2166*

Abbildung 9: Auszug der Terminal-Ausgabe vom GPT Header

Es können erste Informationen über die Partitionsstruktur erhoben werden, wie zum Beispiel die Signatur „EFI PART“ in den ersten acht Bytes des Headers, die Partitionseintragsgröße von 128 Bytes, die Disk GUID und weitere Informationen, die in der Tabelle 4 beschrieben sind.

Der erste und im Beispiel einzige Partitionseintrag beginnt im LBA 2 und umfasst eine Größe von 128 Bytes:

```
MacBook:mnt Lars$ dd if=image_2.dmg bs=512 skip=2 | xxd -l 128
0000000: 0053 4648 0000 aa11 aa11 0030 6543 ecac  .SFH.......0eC..
0000010: c5f1 17f9 ee70 5744 9901 3e1d adea 8209  .....pWD..>.....
0000020: 2800 0000 0000 0000 f7fa 0200 0000 0000  (...............
0000030: 0000 0000 0000 0000 6400 6900 7300 6b00  ........d.i.s.k.
0000040: 2000 6900 6d00 6100 6700 6500 0000 0000   .i.m.a.g.e.....
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Abbildung 10: Partitionseintrag des Volumes mit GPT

Nach den Informationen aus dem Partitionseintrag hat die Partition den Partitionstyp „HFS Plus“, da die GUID im Offsetbereich 0-16 den Wert 48465300-0000-11AA-AA11-00306543ECAC beinhaltet. Die ersten drei GUID-Paare sind in Little-Endian gespeichert. Die Partition beginnt im LBA 40 und endet im LBA 195319 (Offset 32-39 und Offset 40-47 des Partitionseintrags). Die Partitionsbezeichnung lautet „disk image“ (Offset 56) und ist in Unicode codiert.

Die im Rahmen dieser Arbeit verwendeten forensischen Werkzeuge TSK 4.1.3, X-Ways Forensics 18.4, EnCase 7.11.01, FTK 6.0.1 und BlackLight R3.1 können die Partitionsstruktur von APM und GPT interpretieren. X-Ways Forensics bietet hier auch die Möglichkeit, mit Schablonen detailliert die einzelnen Einträge eines GPT Schemas zu analysieren.

4. Dateisystemanalyse nach dem Kategorienmodell von Carrier

Unter einem **Dateisystem** versteht man die Ablageorganisation auf einem Datenträger eines Rechners. Mit Unterstützung des Dateisystems können Dateien gelesen, gespeichert oder gelöscht werden. Das Dateisystem stellt die Zuordnung von dem vom Anwender vergebenen Dateinamen zu der rechnerinternen Dateiadresse her. Das Dateisystem stellt ein Ordnungs- und Zugriffssystem des Dateisystems unter Berücksichtigung der Geräteeigenschaften zur Verfügung und ist normalerweise Bestandteil des Betriebssystems.[39]

So ist das Dateisystem für die Erzeugung bzw. die Löschung von und den Zugriff auf Daten in Form von Dateien und Verzeichnissen verantwortlich. Das Abspeichern, der Zugriff bzw. die Verwaltung der Daten des freien und belegten Speichers werden vom Dateisystem durch zuvor festgelegte Zuteilungs-, Lokalisierungs- und Zugriffsstrategien realisiert. Dabei speichert das Dateisystem für jede Datei und jedes Verzeichnis zusätzliche Informationen, sogenannte Attribute und Metadaten. Beispiele für Attribute und Metadaten sind Ablageorte oder Größe sowie Zeitstempel. Einige Dateisysteme verfügen über Zugriffskontrolllisten (Access Control List (ACL)), die eine Form des Zugriffschutzes der Daten bei Schreib- und Leseoperationen darstellen.

Nach Carrier ist Gegenstand der **Dateisystemanalyse** in der von ihm definierten dritten Schicht (siehe Abbildung 1) das Finden oder Wiederherstellen von Dateien und Verzeichnissen sowie das Lokalisieren von verborgenen Daten.

Die Dateisystemanalyse befasst sich mit der Untersuchung der Daten von logischen Volumes mit dem Ziel der Erkennung des Dateisystems und der Gewinnung der darin gespeicherten Informationen. Dies beinhaltet die Auflistung der enthaltenen Daten in Form von Dateien und Verzeichnissen, den Einblick in die Informationen der damit verbundenen Dateneinheiten sowie die potentielle Wiederherstellung gelöschter Inhalte.

Brian Carrier hat die in einem Dateisystem zu analysierenden Daten in Kategorien unterteilt und darauf basierend eine Methode der forensischen Analyse von Dateisystemdatenstrukturen entwickelt.

Es wird nach diesem Referenzmodell in folgende Kategorien unterschieden:

<u>Kategorie Dateisystem:</u>

Die Kategorie Dateisystem beinhaltet allgemeine Informationen über das Dateisystem. Aufgrund der einzigartigen Struktur jedes Dateisystems können im Bereich dieser Kategorie Informationen beispielsweise über den Aufbau des Dateisystems oder die Größe von Dateneinheiten gewonnen werden.

<u>Kategorie Inhalt:</u>

Diese Kategorie beschreibt den Inhalt von Dateien, die im Dateisystem gespeichert sind. Diese sind in Dateneinheiten mit fester Größe abgelegt. Dateisystemspezifische Bezeichnungen für diese Dateneinheiten sind Cluster oder Allocation Blocks.

[39] Tanenbaum 2009, Abschnitt Dateisysteme

Kategorie Metadaten:

Die Kategorie der Metadaten umfasst Informationen über die Dateien selbst. Es handelt sich um Daten, die Daten beschreiben. Zum Beispiel sind dies Informationen über den Speicherort des Dateiinhalts, die Größe der Datei, die Erstellungs- und Zugriffszeiten oder Zugriffsrechte.

Kategorie Dateinamen:

Die Kategorie der Dateinamen enthält die Namen jeder Datei, die im Dateisystem gespeichert ist. In den meisten Dateisystemen sind die Namen Bestandteil vom Inhalt eines Verzeichnisses mit den korrespondierenden Metadatenadressen. Über den Dateinamen werden die Metadaten „gefunden".

Kategorie Anwendung:

Daten der Anwendungskategorie werden nicht benötigt, um im Dateisystem Daten zu schreiben oder zu lesen. Sie enthalten Informationen, die auch nur optionaler Bestandteil einer Dateisystemspezifikation sein können. Das können Dateisystemjournale oder Benutzer-Quota-Statistiken sein.

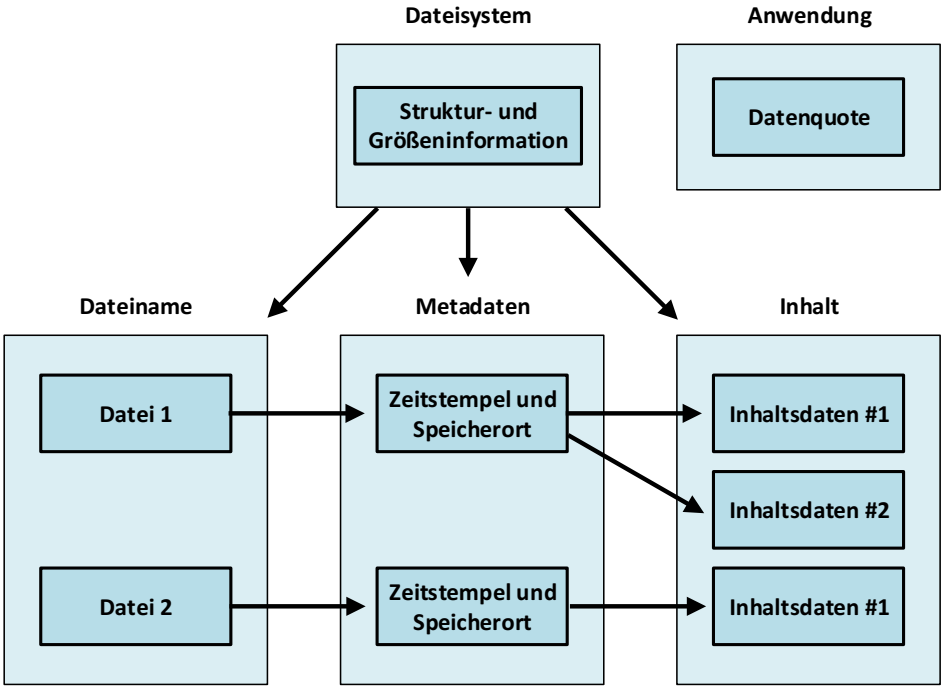


Abbildung 11: Zusammenhang der fünf Dateisystemkategorien nach Brian Carrier

Das Referenzmodell soll die Identifizierung von digitalen Spuren anhand der Kategorien und die Anwendung von den jeweiligen Analysetechniken ermöglichen.

Die folgenden Abschnitte beschreiben basierend auf den Ausführungen Brian Carriers in „File System Forensic Analysis" das Kategorienmodell und anschließend die darauf ausgerichtete Analysetechniken.[40]

[40] Carrier 2005, S. 173–208

4.1 Kategorie Dateisystem

Wie zuvor vorgestellt, enthält das Dateisystem allgemeine Daten, die es ermöglichen, das Dateisystem und dessen Layout zu identifizieren. In der Regel sind diese Daten in einer Standard Datenstruktur im ersten Sektor des Dateisystems gespeichert. Dies sind die ersten Schritte einer Dateisystemanalyse, um die Datenstrukturen der anderen Dateisystemkategorien zu finden. Es können Informationen zur Dateisystemversion, zur Volume ID, zum Label oder Erstellungsdatum in dieser Kategorie ermittelt werden.

Bei der Analyse in diesem Rahmen sind vor allem auch unbenutzte Speicherbereiche zu beachten, die als „Versteck“ für Daten benutzt werden können. Ein Vergleich der Größe des Dateisystems mit dem Volume, auf dem das Dateisystem eingerichtet wurde, decken diese unbenutzten Bereiche auf. Wenn das Volume größer als das Dateisystem ist, werden die Sektoren nach dem Dateisystem als *Volume Slack* bezeichnet.

4.2 Kategorie Inhalt

Eine Analyse dieser Kategorie dient der Suche nach den Speicherorten von Dateien und Verzeichnissen im Dateisystem. Die Daten dieser Kategorie sind in gleichgroßen Dateneinheiten organisiert, die sich entweder in einem allokierten oder nicht-allokierten Zustand befinden. Die Bezeichnung für diese Dateneinheiten variiert je nach Dateisystem (z. B. Cluster oder Allocation Block). Für die Zustandsverfolgung/-verwaltung hat jedes Dateisystem eine bestimmte Datenstruktur (z. B. $Bitmap oder FAT aus den populären Windows-Dateisystemen).

Wenn eine neue Datei erzeugt oder eine bestehende Datei vergrößert wird, werden Dateneinheiten, die noch nicht vom Dateisystem allokiert sind, vom Betriebssystem gesucht und der entsprechenden Datei zugewiesen. Wird eine Datei gelöscht, werden die mit der Datei allokierten Dateneinheiten wieder „freigegeben“. Sie befinden sich dann im nicht-allokierten Zustand und können so wieder mit einer neuen Datei allokiert werden. Bis zur neuen Allokation bleiben die digitalen Inhalte der vorher allokierten Dateneinheiten in der Regel bestehen, da Betriebssysteme bei einem Löschvorgang in der Regel nur die Zuordnung der Dateneinheit aufheben und nicht den Inhalt der Dateneinheit überschreiben.[41]

Gegenstand einer Analyse in der Inhaltskategorie ist die Wiederherstellung gelöschter Daten und die Durchführung einer Low-Level-Suche. Dies erfolgt aufgrund der Speichergrößen und Datenmengen von forensischen Werkzeugen überwiegend automatisiert.

Für eine Analyse dieser Kategorie ist daher wichtig zu wissen, wie Dateneinheiten adressiert und allokiert werden.

Wie oben bereits dargestellt, werden zur Adressierung physische Adressen und logische Volume-Adressen, die relativ zum Anfang eines logischen Volumes beginnen, verwendet. Dateisysteme nutzen die logische Volume-Adressierung aber auch logische

[41] Ein sicheres Löschen von Dateiinhalten wird durch Drittanbieter Software angeboten oder auch immer mehr durch Funktionen/Einstellungsmöglichkeiten in modernen Betriebssystemen ermöglicht.

Dateisystemadressen, die ebenfalls wie die logische Volume-Adressierung relativ zum Anfang des Volumes adressiert werden. Die aufeinanderfolgenden Sektoren bilden eine Dateneinheit.

Für die Allokation von Dateneinheiten existieren unterschiedliche Strategien. Grundsätzlich werden zusammenhängende Dateneinheiten allokiert. Dies ist jedoch nicht immer möglich, so dass Dateneinheiten einer Datei nicht zusammenhängend der Datei zugeordnet sein können. Ist dies der Fall, spricht man von Fragmentierung.

Die *First-Fit* Strategie sucht nach der ersten verfügbaren Dateneinheit beginnend vom Anfang des Dateisystems. Nachdem eine Dateneinheit gefunden und allokiert wurde und eine weitere Dateneinheit für die zu speichernde Datei benötigt wird, sucht das Betriebssystem erneut vom Anfang des Dateisystems nach einer zweiten nicht-allokierten Dateneinheit. Die Wahrscheinlichkeit, dass das Dateisystem mit dieser Allokationsstrategie stark fragmentiert ist, ist sehr hoch. Für die Analyse bedeutet dies, dass nicht-allokierte Dateneinheiten mit noch nicht überschriebenen Daten vorheriger Dateien eher am Ende des Dateisystems wiederhergestellt werden können.

Eine ähnliche Strategie ist die *Next-Fit* Strategie. Der Unterschied zur *First-Fit* Strategie liegt jedoch darin, dass hier nicht vom Anfang des Dateisystems nach der nächsten verfügbaren Dateneinheit gesucht wird, sondern ausgehend von der zuletzt allokierten Dateneinheit. Dies führt zu einer ausgewogeneren Datenwiederherstellung, da das Dateisystem erst wieder vom Beginn anfängt, nicht-allokierte Dateneinheiten zu belegen, nachdem das gesamte Dateisystem bis zum Ende „abgearbeitet“ wurde.

Eine weitere Strategie ist die *Best-Fit* Strategie. Nach dieser Strategie sucht das Betriebssystem im Dateisystem nach zusammenhängenden Dateneinheiten, die für eine Datei benötigt werden. Wenn nicht ausreichend zusammenhängende Dateneinheiten gefunden werden, muss auf die *First-Fit* oder *Next-Fit* Strategie ausgewichen werden.

Auch wenn einige Dateisysteme spezifizieren, welche Allokationsstrategie eingesetzt werden soll, wählen die Betriebssysteme die Allokationsstrategie für ein Dateisystem selbst.

Zu beachten sind ebenfalls die Anwendungen, die den Inhalt einer Datei erstellen. Wenn zum Beispiel der Inhalt einer Datei aktualisiert werden soll, kann eine Anwendung das Original öffnen und nach dem Aktualisieren an der ursprünglichen Stelle speichern. Eine andere Anwendung könnte hingegen eine zweite Kopie vom Original anfertigen, diese dann aktualisieren und umbenennen. Im zweiten Fall würde die Datei mit neuen Dateneinheiten allokiert werden, weil eine neue Datei, nämlich eine zweite Kopie, erstellt wurde.

Beschädigte Dateneinheiten können in der Regel von den Dateisystemen als *beschädigt* gekennzeichnet werden, um zu verhindern, dass diese allokiert werden. Moderne Datenträger übernehmen heute diese Aufgabe, erkennen beschädigte Sektoren und verhindern deren Allokation. Diese Funktionalität kann von einem Anwender genutzt werden, um Daten in von ihm als beschädigt gekennzeichnete Sektoren zu speichern. Gängige Forensik Werkzeuge können als beschädigt gekennzeichnete Sektoren (die s.g. Bad Sectors) erkennen.

4.2.1 Analysetechniken

Folgende Analysetechniken werden auf der Ebene der Kategorie Inhalt durchgeführt.

4.2.1.1 Analyse der Dateneinheiten

Bei dieser Technik geht es um die Einsicht auf den Dateiinhalt anhand von Adressen. Sie ermöglicht das Auffinden von relevanten Daten anhand der Analyse bestimmter allokierter Dateneinheiten, die zu einer Datei gehören. Dazu ist dem Forensiker die logische Dateisystemadresse dieser Dateneinheiten bekannt, die er bei der Untersuchung aufruft und an dieser Adresse den Inhalt der Dateneinheiten analysiert.

4.2.1.2 Logische Dateisystemsuche

Ist der zu analysierende Dateiinhalt bekannt, kann die logische Suche nach dem Inhalt durchgeführt werden, um den Speicherort der Dateiinhalte für die weitere Untersuchung zu finden. Hierzu wird jede Dateneinheit des Dateisystems nach einem Wert oder String durchsucht. Wird eine Datei mit mehreren Dateneinheiten allokiert, die nicht zusammenhängen, dann kann der gesuchte Wert oder String mit dieser Suchtechnik allein nicht gefunden werden. In der logischen Dateisuche kann dieses Problem behoben werden (siehe Kategorie Metadaten). In der Praxis kombinieren forensische Werkzeuge in der Regel beide Suchen.

4.2.1.3 Allokationsstatus der Dateneinheiten

Dateneinheiten, die nicht allokiert sind, können ebenfalls noch Daten einer vorherigen Allokation enthalten. Diese Dateneinheiten können extrahiert werden. Der aus Rohdaten bestehende Inhalt kann analysiert werden. Da der Rohdaten-Export keine Dateisystemstrukturen enthält, können auch keine Dateisystemanalyse-Werkzeuge für die Untersuchung sinnvoll eingesetzt werden.

4.2.1.4 Konsistenzüberprüfung

Konsistenzüberprüfungen von Dateisystemen sind für jede Datenkategorie wichtig, da sie Auskunft darüber geben, ob ein Dateisystem sich in einem normalen oder in einem verdächtigen Zustand befindet.

Für die Kategorie Inhalt wird überprüft, ob jede zugewiesene Dateneinheit genau einen allokierten Metadaten Eintrag besitzt. Dies soll sicherstellen, dass auch nur Dateneinheiten allokiert werden können, wenn eindeutige Metadaten zu diesen Dateneinheiten zugewiesen sind (wie zum Beispiel Dateiname). Besitzen allokierte Dateneinheiten keine zugewiesenen Metadaten, bezeichnet man sie als *verwaiste Dateneinheiten*. Eine doppelte Zuweisung von Metadaten für eine allokierte Dateneinheit ist in der Regel von den meisten Dateisystemen nicht erlaubt.

Eine weitere Möglichkeit, die Konsistenz zu überprüfen, besteht darin, die forensische Sicherung (bitweises Kopieren und Speichern in ein Image-Format) mit den Originaldaten des Dateisystems zu vergleichen. Da forensische Datensicherungssoftware in der Regel beschädigte Sektoren mit Nullen ausfüllen, kann eine Auflistung mit beschädigten Sektoren auf versteckte Daten hinweisen, wenn diese keine Nullen beinhalten.

4.2.2 Löschtechniken

Ein Betriebssystem löscht in der Regel Dateien, indem es die entsprechenden Dateneinheiten deallokiert. Bis die Dateneinheiten neu allokiert werden, ist der ursprüngliche Inhalt in der Dateneinheit gespeichert und so wiederherstellbar. Über moderne Betriebssysteme oder Drittanbietersoftware wird daher ein „sicheres Löschen“ immer mehr implementiert. Hier werden die allokierten oder alle ungenutzten Dateneinheiten mit Nullen oder Zufallswerten überschrieben.

Sollte bei der Analyse ein Löschprogramm von Drittanbietern für sicheres Löschen festgestellt werden, sollten die Zugriffszeiten und die Benutzung dieses Löschprogramms überprüft werden. Je nach Funktionsweise des Löschprogramms und des Verhaltens des Betriebssystems könnten temporäre Kopien der zu löschenden Dateien gefunden werden.

4.3 Kategorie Metadaten

Die Kategorie Metadaten behandelt Daten, die andere Daten beschreiben, zum Beispiel Zeitstempel des letzten Zugriffs, Erstellungsdatum und die Adresse der zugehörigen allokierten Dateneinheiten.

Die meisten forensischen Werkzeuge analysieren Daten dieser Kategorie zusammen mit der Kategorie der Dateinamen.

Metadaten werden in festen oder dynamischen tabellarischen Datenstrukturen gespeichert, wobei jeder Eintrag adressierbar ist. Wird eine Datei gelöscht, wird der zugehörige Metadateneintrag auf *nicht-allokiert* gesetzt. Manche Werte können auch sofort vom Betriebssystem überschrieben werden.

Mit der Analyse von Daten der Metadatenkategorie können Informationen über eine bestimmte Datei erhoben oder nach gewissen Kriterien in dieser Kategorie gesucht werden.

Bei der Analyse von Metadaten ist zu beachten, dass diese in der Regel leicht manipulierbar sind und deshalb nur im Kontext weiterer Informationen wertvolle Details liefern.

4.3.1 Logische Dateiadressen

Wie zuvor bereits angesprochen, verfügen Dateneinheiten über eine logische Dateisystemadresse. Ist eine Dateneinheit mit einer Datei allokiert, hat sie auch eine logische Dateiadresse, welche relativ zum Start der entsprechenden Datei ist, zu der die Dateneinheit zugeordnet ist.

4.3.2 Slack Space

Für die Analyse von Daten in der Kategorie Metadaten spielt der Slack Space eine wichtige Rolle. In einem Dateisystem stellt eine Dateneinheit die kleinste allokierbare Einheit dar, in der Daten gespeichert werden können. Wie zuvor dargestellt, werden abhängig vom Dateisystem logische Sektoren fester Größe zu einer Dateneinheit zusammengefasst. Die meisten Dateien sind oft aber kein Vielfaches einer Dateneinheit, so dass in der Regel eine Datei eine gesamte Dateneinheit allokieren muss, obwohl sie kleiner ist als die Dateneinheit. Der übrige Teil der mit dieser Datei allokierten Dateneinheit wird *Slack Space* genannt. Zum

Beispiel allokiert eine 100 Byte große Datei eine Dateneinheit mit 4096 Bytes, dann sind nach den 100 Bytes die übrigen 3996 Bytes Slack Space.

Dieser Slack Space gehört zu allokierten Dateneinheiten. Dieser Bereich könnte Daten vorheriger Dateien enthalten, wenn er nicht vom Betriebssystem überschrieben wurde.

Der Slack Space lässt sich in zwei Bereiche unterteilen, Datei-Slack und Ram-Slack. Am folgenden Beispiel lässt sich dies gut veranschaulichen. [42]

Nehmen wir an, eine Datei hat eine Größe von 100 Bytes. Das Dateisystem ist so organisiert, dass eine Dateneinheit mit zwei logischen Sektoren mit je 512 Bytes zusammengefasst wird.

Für die Allokation dieser Datei genügt eine Dateneinheit, da die Datei mit 100 Bytes nur einen Bruchteil eines logischen Sektors einnimmt. Der erste Slack Space umfasst den restlichen Teil des mit 100 Bytes belegten Sektors, also 412 Bytes. Dieser Slack Space wird auch Ram-Slack genannt, da ältere Betriebssysteme diesen Bereich mit Daten aus dem Hauptspeicher gefüllt haben. Aktuelle Betriebssysteme füllen diesen Bereich in der Regel mit Nullen. Der zweite und letzte unbenutzte logische Sektor dieser allokierten Dateneinheit stellt den zweiten Slack Space dar. Je nach Betriebssystem ist dieser Bereich gelöscht oder unangetastet und somit mit den Daten einer vorherigen Allokation belassen.

4.3.3 Metadatenbasierte Dateiwiederherstellung

Für die Wiederherstellung gelöschter Dateien existieren zwei wichtige Methoden: die metadatenbasierte und die anwendungsbasierte Wiederherstellung. Für Dateien, die anhand ihrer Metadaten wiederhergestellt werden sollen, ist es notwendig, dass die Metadaten nach dem Löschen der Datei erhalten bleiben. Werden die Metadaten ebenfalls gelöscht oder wird die Metadatenstruktur mit einer neuen Datei allokiert, kann nur noch versucht werden, die Datei anwendungsbasiert wiederherzustellen.

Werden Metadaten einer gelöschten Datei gefunden, kann diese anhand ihrer Metadateneinträge wiederhergestellt werden, wenn sie nicht überschrieben wurde. Dies ist insbesondere zu beachten, wenn die Dateneinheit, auf die die Metadaten zeigen, neu allokiert oder auch zwischenzeitlich neu allokiert und wieder gelöscht wurde. Die Dateneinheit kann somit einen neuen Dateninhalt und bei aktueller Allokation neue Metadaten besitzen.

4.3.4 Analysetechniken

Die Analyse von Metadaten dient der Betrachtung von Dateiinhalten, der Suche nach Werten und dem Auffinden von gelöschten Dateien. Nachfolgend werden Analysetechniken für die Kategorie Metadaten nach Brian Carrier vorgestellt.

4.3.4.1 Metadaten Lookup

Eine gute Methode, Metadaten einer Datei zu analysieren, ist es, anhand von Dateinamen die Metadatenstruktur dieser Datei zu untersuchen. Der Dateiname zeigt auf die Metadatenstrukturen. Diese können ausgewertet und dadurch Informationen aus den Metadaten zur entsprechenden Datei gewonnen werden.

[42] Kuhlee und Völzow 2012, S. 58–59

Beim Auflisten von Dateien in Verzeichnissen führen die gängigen Forensik-Programme diesen Prozess automatisiert durch. Die Prozedur ist hierbei abhängig vom jeweiligen Dateisystem, da die Metadaten je nach Dateisystem an unterschiedlichen Stellen gespeichert sind.

4.3.4.2 Logische Dateibetrachtung

Nach Analyse der Metadatenstruktur können die allokierten Dateneinheiten der entsprechenden Datei aufgesucht und so kann der Dateiinhalt gelesen werden. Hierbei muss der Slack Space, also die Differenz zwischen der logischen und physischen Dateigröße, beachtet werden. Diese Informationen lassen sich aus den Metadaten erheben.

4.3.4.3 Logische Dateisuche

Die logische Dateisuche nutzt die gleiche Technik wie die logische Dateibetrachtung. Der Unterschied besteht darin, dass hier nach einem Wert in den Dateneinheiten, also im Dateiinhalt, gesucht wird. Diese Technik ähnelt auch sehr der vorher bereits behandelten logischen Dateisystemsuche. Im Gegensatz dazu werden bei der logischen Dateisuche jedoch die Dateneinheiten anhand der Metadateninformationen in der Reihenfolge, wie sie von der Datei genutzt werden, analysiert. Auf dieser Weise können auch Dateiinhalte gefunden werden, die auf nicht zusammenhängenden Dateneinheiten verteilt sind. Diese Suche kann nur auf allokierte Dateneinheiten angewandt werden, wobei darauf geachtet werden muss, ob das verwendete forensische Werkzeug Slack Spaces bei der Suche berücksichtigt.

Eine Variante der logischen Dateisuche ist die Suche einer Datei mit einer bestimmten kryptografischen Prüfsummen (MD5, SHA1 etc.).

4.3.4.4 Analyse von nicht allokierten Metadaten

Metadaten, die nicht mehr allokiert sind, können noch relevante Daten beinhalten, wenn zum Beispiel der Dateiname gelöscht und die Dateneinheit noch vor den Metadatenstrukturen neu allokiert wurde.

4.3.4.5 Analyse von Metadaten-Attributen

Für die Analyse von Metadaten kann die Untersuchung von Dateiaktivitäten, die auf bestimmte Gegebenheiten oder Umstände eines Systems schließen lassen, von Bedeutung sein. Diese Analyse wird anhand der Metadatenwerte, wie zum Beispiel Zeitstempel einer oder mehrerer Dateien, durchgeführt. Um Veränderungen oder Inkonsistenzen in einem System festzustellen, kann eine Timeline Analyse durchgeführt werden.

Viele forensische Werkzeuge besitzen Timeline-Funktionen. Brian Carrier hat *mactime* in TSK integriert. Mit *mactime* lassen sich

- m-time = Schreibzugriff / inhaltliche Änderung
- a-time = Zugriff
- c-time = Metadatenänderung
- b-time = Erzeugt (nur bei FAT & NTFS)

von Dateien und Verzeichnissen erheben.

Es ist zwingend notwendig, das zu analysierende Dateisystem interpretieren zu können, um die dort gespeicherten Zeitstempel in Korrelation bringen zu können. Des Weiteren muss das

vom Dateisystem verwendete Zeitsystem beachtet werden. Wird zum Beispiel Universal Time Coordinated (UTC) verwendet, so muss aus den Betriebssystemdaten die eingestellte Lokalzeit erhoben werden und die im Dateisystem gespeicherten Zeitstempel von UTC in Lokalzeit umgerechnet werden.

Carrier führt im Zusammenhang mit der Analyse von Metadaten-Attributen die Möglichkeit an, nach Berechtigungen bestimmter Benutzer für bestimmte Dateien zu suchen. So kann zum Beispiel nach dem Schreibzugriff, dem Ersteller einer Datei oder der Besitzer ID gesucht werden.

Wurde eine logische Dateisystemsuche durchgeführt und wurden relevante Daten einer Dateneinheit gefunden, ist es möglich, anhand der Adresse der Dateneinheit die zugehörigen Metadateneinträge zu ermitteln, welche Datei diese Dateneinheit allokiert, um anschließend alle zur Datei gehörenden Dateneinheiten zu finden.

4.3.4.6 Allokationsreihenfolge von Datenstrukturen

Wenn es notwendig ist, herauszufinden, in welcher Reihenfolge Dateneinheiten allokiert wurden, ist dies abhängig von der Allokationsstrategie des jeweiligen Betriebssystems. In der Regel werden Metadaten mit der *First-Available* oder *Next-Available* Strategie allokiert. Dies ist aber anwendungsabhängig und wird von Carrier im Zusammenhang der Dateisystemkategorien nicht weiter behandelt.

4.3.4.7 Konsistenzüberprüfung

Ein Konsistenzcheck mit Metadaten könnte versteckte Daten aufdecken oder Dateisystemfehler entdecken, die verhindern, dass bei der Untersuchung des Dateisystems alle Daten analysiert werden können.

Eine Überprüfung kann durch eine Verifikation, ob jeder allokierte Metadateneintrag eine zugehörige Dateneinheit hat, die ebenfalls allokiert ist, durchgeführt werden. Die Anzahl der Dateneinheiten sollte mit der Dateigröße konsistent sein. Unter der Überschrift Konsistenzüberprüfung in der Kategorie Inhalt wurde bereits angeführt, dass grundsätzlich jede allokierte Dateneinheit exakt auf einen Metadateneintrag zeigt. In diesem Zusammenhang sollten Metadateneinträge für spezielle Dateitypen keine allokierten Dateneinheiten besitzen (z. B. Sockets, die von Systemprozessen für die Netzwerkkommunikation genutzt werden).

Eine weitere Möglichkeit der Konsistenzüberprüfung besteht darin, im Bereich der Kategorie Dateinamen zu verifizieren, ob jeder allokierte Verzeichniseintrag einen allokierten Namen hat, der zu ihm zeigt.

4.3.5 Löschtechniken

Wird eine Datei gelöscht, können die zugehörigen Metadaten ebenfalls gelöscht werden, wodurch eine Wiederherstellung der Datei erschwert wird. Durch spezielle Software können gelöschte Metadaten auch mit Nullen oder Zufallswerten überschrieben werden. In dieser Konstellation ist zwar eine Wiederherstellung der Datei anhand von Metadaten nicht möglich, es kann jedoch auf eine Datenmanipulation hinweisen, wenn die überschriebenen Metadateneinträge zwischen zwei gültigen Metadateneinträgen gespeichert sind.

Werden Metadaten mit realen Metadatenwerten überschrieben, die auf andere Dateneinheiten einer logisch vorhandenen Datei zeigen, ist eine Wiederherstellung in der Regel nicht möglich, da keine Korrelation zur vorherigen Datei mehr existiert. In diesem Fall kann eine Wiederherstellung anwendungsbasierend versucht werden.

4.4 Kategorie Dateinamen

Die Kategorie Dateinamen beinhaltet den Namen einer Datei in einem Dateisystem und ermöglicht den Zugriff auf den Inhalt der Datei über den Dateinamen an Stelle der Metadatenadresse, da der Dateiname eine Referenz zur Metadatenadresse darstellt. Abhängig vom Dateisystem können in der Kategorie Dateinamen neben dem Dateinamen und der Metadatenadresse auch weitere Informationen vorhanden sein (zum Beispiel Dateityp).

Wichtig für die Analyse von Daten in der Dateinamen-Kategorie ist die Position des Stammverzeichnisses im Dateisystem, da nur so der gesamte Pfad der Datei ermittelt werden kann.

4.4.1 Dateinamenbasierte Wiederherstellung von Dateien

Für die Wiederherstellung von Dateien in der Kategorie Dateinamen werden ein gelöschter Dateiname und die zugehörige Metadatenadresse identifiziert, um den Dateiinhalt basierend auf Metadaten wiederherzustellen.

Es sind verschiedene Szenarien vorstellbar, bei denen eine Untersuchung von gelöschten Dateien aus der Dateinamensperspektive eine Rolle spielen könnte. So können zum Beispiel Metadateneinträge und Dateneinheiten mit einer neuen Datei allokiert werden. Nicht mehr allokierte Metadatenstrukturen sind nicht mehr mit Dateinamen verknüpft.

4.4.2 Analysetechniken

4.4.2.1 Auflistung der Dateinamen

Da Dateinamen Dateien zugeordnet sind, ist die Auflistung von Dateien und Verzeichnissen anhand ihrer Namen und den jeweiligen Metadatenadressen eine sehr populäre Methode für die Untersuchung von Dateisystemen. Beispiele für die Analyse anhand der Kategorie Dateinamen sind die Suche nach dem Dateinamen, den Dateipfaden oder der Dateiendung einer Datei.

Für die Analyse wird im ersten Schritt das Stammverzeichnis im Dateisystem ermittelt, um die Datei- und Verzeichnisnamen aller allokierten Dateneinheiten des Stammverzeichnisses und weiter Verzeichnisse aufzufinden. Anhand der Metadateneinträge dieser Verzeichnisse können die allokierten Dateneinheiten des Stammverzeichnisses und weiterer Verzeichnisse ausgelesen und so alle Dateinamen gefunden und gelistet werden.

Anschließend können mit Analysetechniken der Kategorie Metadaten über die zum Dateinamen zugehörige Metadatenadresse die Metadateneinträge ausgelesen und die Inhalte der allokierten Dateneinheiten der Datei aufgesucht werden.

4.4.2.2 Suche nach Dateinamen

In Konstellationen, bei denen nur Teile von Dateinamen oder der Speicherpfad einer Datei nicht bekannt sind, können Suchen im Dateisystem nach den bekannten Namensbestandteilen durchgeführt werden, zum Beispiel nach bestimmten Dateiendungen.

In diesem Zusammenhang sollten eventuell durchgeführte Verschleierungshandlungen beachtet werden, wie zum Beispiel das Verwenden anderer Dateiendungen. In solchen Fällen können anwendungsbasierte Analysetechniken verwendet werden, um alle Dateien eines Typs zu identifizieren.

Diese Verfahrensweise ähnelt der des Auflistens von Dateinamen. Es wird der Inhalt eines Verzeichnisses analysiert und jeder Verzeichniseintrag mit den Suchbegriffen verglichen. Gefundene oder übereinstimmende Verzeichnisse werden dabei rekursiv durchsucht.

Wird eine relevante allokierte Dateneinheit gefunden, die einer Datei zugehörig ist, deren Dateinamen noch nicht bekannt ist, kann anhand der zur Dateneinheit allokierten Metadateneinträge die gesamte Metadatenstruktur analysiert und so auch der Dateiname ermittelt werden.

4.4.2.3 Allokationsreihenfolge von Datenstrukturen

Die Allokationsreihenfolge von Dateinamen kann, wie zuvor bereits in der Metadatenkategorie angesprochen, wichtige Informationen über den Erstellungszeitpunkt von Dateinamen liefern. Dies ist vom jeweiligen Betriebssystem abhängig.

4.4.2.4 Konsistenzüberprüfung

Die Konsistenzüberprüfung in dieser Kategorie dient der Verifizierung, ob jeder allokierte Dateiname auf eine allokierte Metadatenstruktur zeigt. Für einige Dateisysteme ist dies auch gegeben, wenn mehrere Dateinamen für eine Datei und eine Metadatenadresse existieren.

4.4.2.5 Löschtechniken

Ein Löschvorgang in dieser Kategorie wird durch das Löschen des Dateinamens und der zugehörigen Metadatenadresse von der Metadatenstruktur realisiert. Dafür gibt es verschiedene Techniken. Eine besteht darin, die Werte der Dateinamensstruktur zu überschreiben. Durch eine Analyse kann so ein existierender Eintrag ermittelt werden, der nicht mehr valide ist. Schwieriger wird dieses Vorgehen, wenn ein Betriebssystem mit der *Next-Available* Strategie den neuen Dateinamen am Ende einer Liste platziert.

Eine weitere Löschtechnik für das Löschen von Dateien geschieht durch das Reorganisieren der Dateinamensliste, so dass ein existierender Dateiname einen gelöschten Dateinamen überschreibt.

4.5 Kategorie Anwendung

Einige Daten im Dateisystem gehören zur Kategorie Anwendung. Das üblichste Feature der Kategorie Anwendung ist das Dateisystem Journal und wird zur Protokollierung von sämtlichen Dateimanipulationen verwendet. Diese Funktion wird aus Performancegründen eingesetzt und ist nicht für die Funktion des Dateisystems notwendig.

4.5.1 Dateisystem Journale

Ein Dateisystem Journal wird von entsprechenden Dateisystemen mit dieser Funktion verwendet, um zu gewährleisten, dass sich das Dateisystem in einem konsistenten Zustand befindet. Wird zum Beispiel die Größe einer Datei durch Manipulationen verändert, dann müssen die allokierten Metadaten und Dateneinheiten aktualisiert werden. Kommt es zu einem Systemabsturz oder unsachgemäßem Herunterfahren des Betriebssystems, bevor die Aktualisierung der Metadaten und Dateneinheiten durchgeführt wurde, ist das Dateisystem nicht mehr konsistent. Um sicher zu gehen, dass dies auch tatsächlich geschehen ist, führt ein Dateisystem einen Konsistenzcheck durch, um dies zu überprüfen. Ein Dateisystem mit Journal protokolliert jede Dateimanipulation, bevor die Metadaten und Dateneinheiten aktualisiert werden. So kann auch nach einem Systemabsturz die Konsistenz sehr effizient vom Dateisystem wiederhergestellt werden, da die im Journal gespeicherten Informationen ein Auffinden der betroffenen Dateien sehr schnell ermöglicht, ohne dass das gesamte Dateisystem überprüft werden muss.

Carrier behandelt Informationen aus dem Dateisystem Journal in der Kategorie Anwendung, da diese nicht für den Betrieb des Dateisystems notwendig sind und nur aus Effizienzgründen das Dateisystem Journal genutzt wird.

Dateisystem Journale geben letztendlich nützliche Informationen über die letzten Aktivitäten im Dateisystem und über den Zeitpunkt, an dem die protokollierten Dateimanipulationen noch nicht umgesetzt wurden.

4.5.1.1 Anwendungsbasierte Dateiwiederherstellung (Data Carving)

Bei der anwendungsbasierten Wiederherstellung von Dateien wird grundsätzlich nach Signaturen von bekannten Dateitypen gesucht, um die nicht mehr allokierten Dateneinheiten einer Datei wiederherzustellen. Anwendung findet diese Art der Datenwiederherstellung bei gelöschten Dateien, deren nicht mehr allokierten Dateneinheiten keine Verknüpfung zu den zugehörigen Metadaten besitzen oder die Metadaten nicht mehr existent sind.

Die meisten Forensik Werkzeuge verfügen über die Data Carving Funktionalität. Die Signaturen für das Data Carving lassen sich grundsätzlich anpassen und so auch um neue Dateitypen erweitern. Die Signaturen enthalten in der Regel die Start- und Ende-Signatur des bekannten Dateityps, die Dateiendung und die maximale Größe in Byte, falls die Ende-Signatur nicht gefunden wird oder nicht bekannt ist.

4.5.1.2 Sortieren der Dateitypen

Eine Möglichkeit der Analyse in dieser Kategorie besteht darin, Dateitypen anhand ihrer Inhaltsstruktur zu sortieren, um ähnliche Dateien zu gruppieren. So lassen sich zum Beispiel alle Bilddateien zusammenfassen und darstellen. Die meisten Forensik Programme beinhalten derartige Funktionen basierend auf Dateiendungen oder Dateisignaturen.

5. Analyse des HFS Plus Dateisystems anhand des Kategorienmodells von Carrier

Brian Carrier hat in seinem Referenzmodell der Dateisystemkategorien Apple Dateisysteme nicht behandelt. Im Nachfolgenden soll das Referenzmodell auf das Apple Macintosh OS Extended File System, welches besser unter der Bezeichnung HFS Plus oder HFS+ bekannt ist, angewandt werden. HFS steht hierbei für *Hierarchical File System*.[43]

HFS ersetzte im Jahr 1985 das Macintosh File System (MFS)[44] und wurde auf Grund von Limitierungen (z. B. Unterstützung von nur 16 Bit Allocation Blocks) mit der Einführung von Mac OS 8.1 im Jahr 1998 durch HFS Plus mit einigen Verbesserungen ersetzt.[45]

Die folgende Tabelle als Auszug aus der *Apple Technical Note TN1150* fasst einige wichtige Informationen von HFS Plus zusammen und stellt die Unterschiede zu HFS dar.

Feature	HFS	HFS Plus	Vorteil/Kommentar
Anzahl der Allocation Blocks	16 Bit	32 Bit	Nutzung von hoher Speicherkapazität und eine größere Anzahl von Dateien per Volume
Dateinamenslänge	31 Zeichen	255 Zeichen	Verbesserung Cross-Plattform Kompatibilität
Dateinamenkodierung	MacRoman	Unicode	Unterstützung internationaler Sprachen
Datei- und Ordner-Attribute	Feste Größe	Unterstützung für erweiterte Attribute	
Unterstützung für Betriebssystemstart	Systemordner ID	Dedizierte Bootdateien werden unterstützt	HFS Plus Volumes können von nicht-Mac OS-Systemen gebootet werden
Catalog Node Größe	512 Bytes	4 KB	Erhält die Effizienz im Zusammenhang mit den Änderungen von HFS auf HFS Plus
Maximale Dateigröße	2^{31} Bytes	2^{63} Bytes	Vorteile für den Benutzer (Speicherung großer Dateien, z. B. mit Multimediainhalten)

Tabelle 8: Vergleich HFS und HFS Plus[46]

[43] Singh 2007, S. 1471
[44] Wikipedia: *Hierarchical File System*
[45] Singh 2007, S. 1471
[46] Apple: *Technical Note TN1150*, S. 1

Ein weiterer, signifikanter Unterschied besteht in der Speicherung von Zeitstempeln, die in HFS in Lokalzeit gespeichert werden - bei HFS Plus dagegen in Sekunden ab Mitternacht des 1. Januars 1904, GMT.[47] [48]

In einem HFS Plus Dateisystem werden alle multi-Byte Integer-Werte im Big-Endian-Format[49] gespeichert.[50] Dies ist bei der Analyse von Datenträgern mit HFS Plus Volumes besonders zu beachten, da diese in der Regel mit GPT-Partitionierung eingerichtet sind und die GPT Header Daten in Little-Endian speichern.

Die kleinste allokierbare Dateneinheit in einem HFS Plus Dateisystem wird Allocation Block genannt, der in der Regel aus 8 Blocks mit jeweils 512 Bytes gebildet wird.[51] Eine Gruppe von zusammenhängenden Allocation Blocks wiederum wird Clump genannt.[52] HFS Plus kann Daten bevorzugt in Clumps allokieren, um externe Fragmentierung zu verhindern.[53] Folgende Übersicht veranschaulicht dies:

512	512	512	512	512	512	512	512
512	512	512	512	512	512	512	512
512	512	512	512	512	512	512	512
512	512	512	512	512	512	512	512

Abbildung 12: Darstellung Block, Allocation Block, Clump in Byte

Eine Übersicht über ein HFS Plus Dateisystemlayout kann der folgenden Abbildung entnommen werden.

[47] Greenwich Mean Time (Standard Time)
[48] Apple: *Technical Note TN1150*, S. 7
[49] Bei Big-Endian wird das höchstwertige Byte zuerst gespeichert. Bei zusammenhängenden Bytes wird das höchstwertige zuerst genannt/gelesen.
[50] Apple: *Technical Note TN1150*, S. 5
[51] Apple: *Technical Note TN1150*, S. 2–3
[52] Singh 2007, S. 1481
[53] Singh 2007, S. 1481

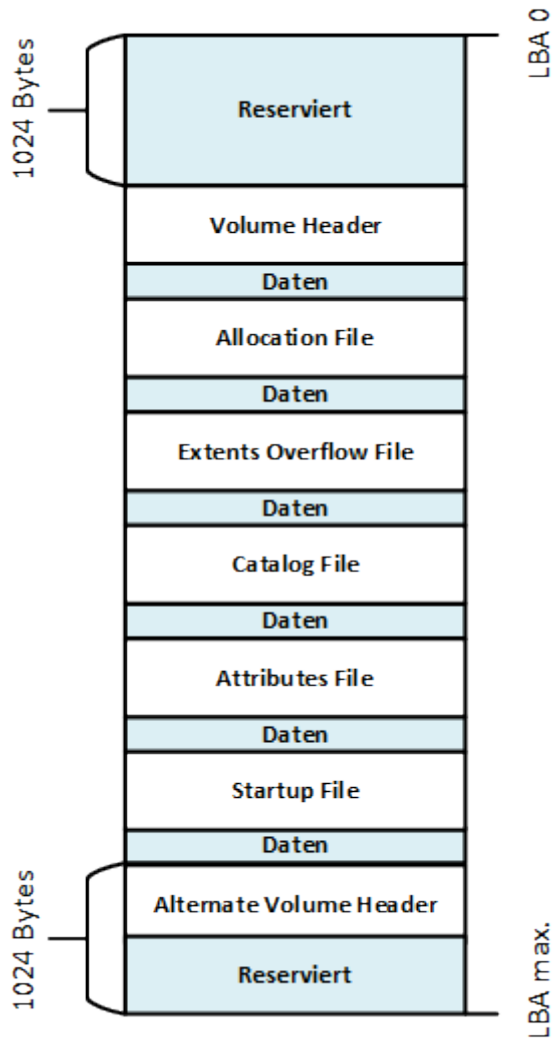


Abbildung 13: Dateisystemlayout HFS Plus

5.1 Kategorie Dateisystem

Wichtige Bestandteile einer HFS Plus Volume-Struktur sind der Volume Header, der HFS Wrapper und die Special Files. Der Volume Header beinhaltet wichtige Informationen über das Volume und die Positionen der *Special Files*. Die *Special Files* verwalten Metainformationen und sind für die Allokation von Dateneinheiten (Allocation Blocks) zuständig. Der HFS Wrapper gewährleistet eine Abwärtskompatibilität mit älteren Mac OS Versionen.

5.1.1 Volume Header

Jedes HFS Plus Volume hat einen Volume Header, der die HFS Plus Dateisystemstruktur und den Inhalt des Volumes beschreibt. Er beginnt 1024 Bytes vom Volume Offset ausgehend und hat eine Größe von 512 Bytes.[54] Eine Kopie des *Alternate Volume Headers* ist 1024 Bytes vom Ende des Volumes ausgehend gespeichert. Dateisystem-Reparier-Werkzeuge benutzen in der

[54] Varsalone et al. 2009, S. 107

Regel diese Header-Kopie.[55] Die Struktur eines HFS Plus Volume Headers ist nachfolgend dargestellt.

Offset (Byte)	Länge (Byte)	Beschreibung
0	2	Signatur (H+ oder HX)
2	2	Version
4	4	Attribute
8	4	Zuletzt eingebundene Version
12	4	Journal Info Block
16	4	Erstellungszeitpunkt (Lokalzeit)
20	4	Änderungszeitpunkt (GMT)
24	4	Backupzeitpunkt (GMT)
28	4	Überprüfungszeitpunkt (GMT)
32	4	Dateianzahl
36	4	Ordneranzahl
40	4	Größe der Allocation Blocks
44	4	Anzahl der Allocation Blocks
48	4	Freie Allocation Blocks
52	4	Nächste Allokation
56	4	Größe Ressource Clump
60	4	Größe Data Clump
64	4	Nächste Catalog ID
68	4	Anzahl, wie oft das Volume eingebunden wurde
72	8	Kodierungsbitmap
80	32	Finder Info (Attribute)
112	80	Position und Größe des Allocation Files
192	80	Position und Größe des Extents Overflow Files
272	80	Position und Größe des Catalog Files
352	80	Position und Größe des Attribute Files
432	80	Position und Größe des Startup Files

Tabelle 9: HFS Plus Volume Header Struktur[56]

Abbildung 14 zeigt anhand eines Beispiels einen HFS Plus Volume Header eines Disk Images (DMG), das mit GPT partitioniert und mit dem Dateisystem HFS Plus eingerichtet wurde. Die Analyse wurde mit *Synalyze It! Pro* durchgeführt.[57]

Im Offset 0-1 steht als Disk Signatur **H+**. Dies bedeutet, dass es sich um ein HFS Plus Dateisystem mit Journal handelt. Eine weitere Möglichkeit wäre **HX** für die case-sensitive Version von H+. In diesem Fall würde das darauf folgende Feld *Version* **5** anstelle von **4** beinhalten.[58]

Im Beispiel wird als *Last Mounted Version* **HFSJ** ausgegeben und gibt an, dass das Volume mit aktiviertem Journal in einem Mac OS X System eingebunden war. Weitere Beispiele für Werte

[55] Singh 2007, S. 1494
[56] Varsalone et al. 2009, S. 108–109
[57] Pehnack: *Synalize IT! Pro*
[58] Singh 2007, S. 1495

in diesem Feld sind *8.10* (in einem Mac OS 8.1 bis 9.2 eingebunden), *10.0* (ohne Journal eingebunden), *fsck* (eingebunden von fsck) oder Codes von Drittanbietern.

Bei der Analyse des HFS Plus Volume Headers sind die Zeitstempelangaben zu beachten. Wie zuvor bereits erwähnt, werden Zeitstempel in einem HFS Plus Dateisystem als vorzeichenloser 32 Bit Integer-Wert in Sekunden vom 1. Januar 1904 Mitternacht angegeben. Das Feld *Creation Date* zeigt dabei den Zeitstempel in Lokalzeit, während alle andern Zeitstempel in GMT angegeben werden.[59] In der Ausgabe von *Synalyze IT! Pro* steht als Erstellungszeitpunkt in Sekunden **3530368295**. Mit einem Konvertierungswerkzeug wie *BlackBag Epoch Converter*[60] kann die Zeit in GMT oder Lokalzeit konvertiert werden, hier: **2015-11-14 17:51:35 Sat UTC**.

Aus dem Screenshot der Header Analyse mit *Synalyze IT! Pro* ist ersichtlich, dass neun Dateien und neun Verzeichnisse im Volume liegen. Das Root Verzeichnis und die *Special Files* werden bei dieser Angabe im Volume Header nicht mitgezählt.[61] [62]

Ab Offset 112 werden Größe und Position der *Special Files* angegeben. Jeder Eintrag für das jeweilige *Special File* beinhaltet die logische Größe, die Größe der Clumps, die Anzahl der Allocation Blocks und ein Array mit Größe und Länge von jedem Extent des *Special Files*. Die Struktur der Extent Einträge für Extents der *Special Files* im Volume Header werden in Tabelle 10 dargestellt.

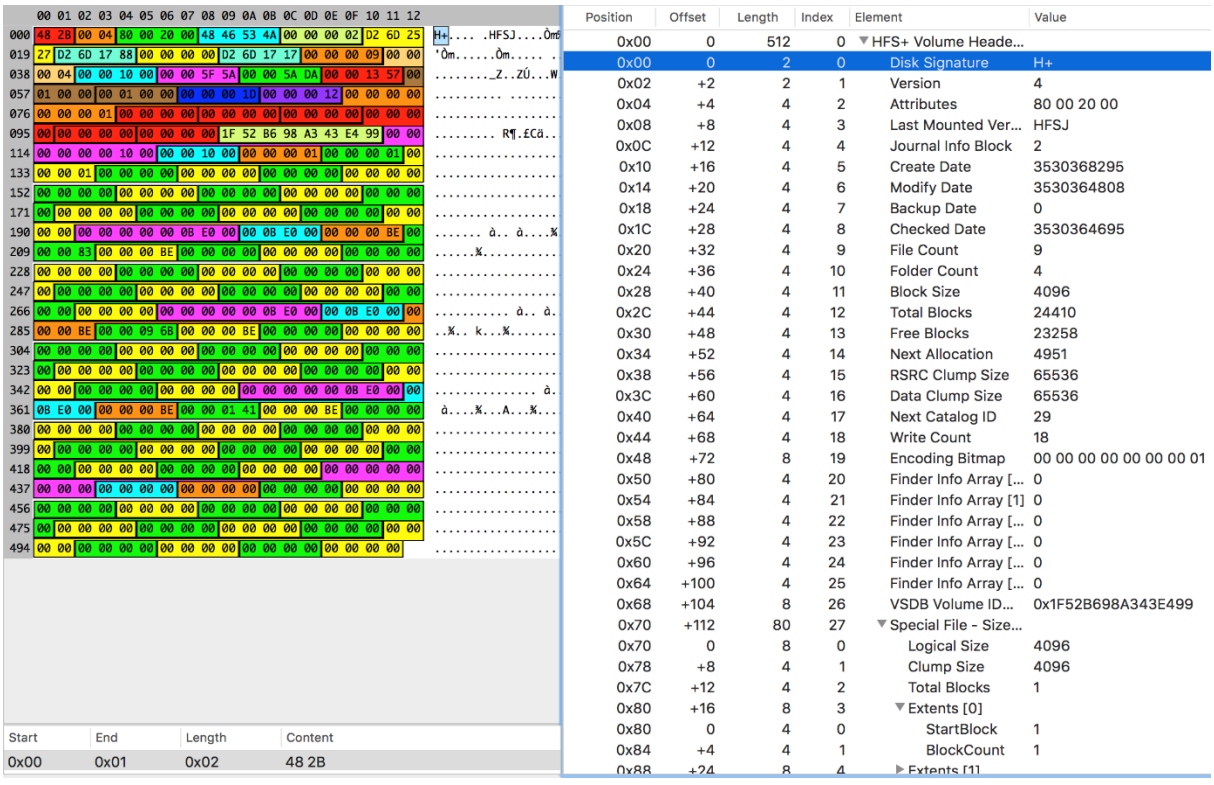


Abbildung 14: Beispiel HFS Plus Volume Header; Screenshot Synalyze IT! Pro (Auszug)

[59] Singh 2007, S. 1496
[60] www.blackbagtech.com/resources/freetools/epochconverter.html, aufgerufen am 12.11.2015
[61] Singh 2007, S. 1496
[62] Singh 2007, S. 1505

Offset	Größe in Bytes	Inhalt
0	8	Logische Größe
8	4	Clump Größe
12	4	Anzahl Allocation Blocks
16	4	Extent 1 – Start Allocation Block
20	4	Extent 1 – Anzahl Allocation Blocks
24	4	Extent 2 – Start Allocation Block
28	4	Extent 2 – Anzahl Allocation Blocks
32	4	Extent 3 – Start Allocation Block
36	4	Extent 3 – Anzahl Allocation Blocks
40	4	Extent 4 – Start Allocation Block
44	4	Extent 4 – Anzahl Allocation Blocks
48	4	Extent 5 – Start Allocation Block
52	4	Extent 5 – Anzahl Allocation Blocks
56	4	Extent 6 – Start Allocation Block
60	4	Extent 6 – Anzahl Allocation Blocks
64	4	Extent 7 – Start Allocation Block
68	4	Extent 7 – Anzahl Allocation Blocks
72	4	Extent 8 – Start Allocation Block
76	4	Extent 8 – Anzahl Allocation Blocks

Tabelle 10: Format der Fork Daten-Einträge der Special Files im Volume Header

Eine weitere Möglichkeit, mit Mac OS X-internen Werkzeugen den Volume Header zu analysieren, bietet das Mac Terminal Programm *hdiutil.* Dieses kann in einem laufendem System oder auf ein Mac Disk Image angewandt werden. Folgende Abbildung zeigt die Anwendung aus dem Beispielimage[63] mit dem Befehl: *#hdiutil fsid image_2.dmg*.

[63] Das Testimage im Expert Witness Format (hier E01) wurde mit xmount als DMG im Mac OS X System eingebunden.

```
Analyzing partition 4: disk image Apple_HFS
HFS+
volume size                             0x05F5A000 (99983360) bytes [95.4 MB]
min stretch size                        0x01368000 (20348928) bytes [19.4 MB]
max stretch size                        0x08000000 (134217728) bytes [128 MB]
current free space                      0x05ACF000 (95219712) bytes [90.8 MB]
allocation blocks                       0x00005F5A (24410)
block size                              0x00001000 (4096) bytes [4 KB]
post-al-block space                     0x00000000 (0) sectors
VH (sector 2)
   signature                    H+
   version                      0x0004
   attributes                   0x80002100
   lastMountedVersion           0x4846534A (HFSJ)
   journalInfoBlock             0x00000002
   createDate                   0xD26D2527 14.11.15, 17:51:35 MEZ
   modifyDate                   0xD270D518 17.11.15, 12:59:04 GMT
   backupDate                   0x00000000 01.01.04, 00:00:00 GMT
   checkedDate                  0xD26D1717 14.11.15, 16:51:35 GMT
   fileCount                    0x00000012
   folderCount                  0x00000005
   blockSize                    0x00001000
   totalBlocks                  0x00005F5A
   freeBlocks                   0x00005ACF
   nextAllocation               0x00000205
   rsrcClumpSize                0x00010000
   dataClumpSize                0x00010000
   nextCatalogID                0x00000029
   writeCount                   0x00000055
   encodingsBitmap              0x0000000000000001
   finderInfo
     0                          0      Blessed folder directory ID
     1                          0
     2                          0      Open folder directory ID
     3                          0      Mac OS 9 blessed folder directory ID
     4                          0
     5                          0      Mac OS X blessed folder directory ID
     VSDB Volume ID             0x1F52B698A343E499
   allocationFile
     logicalSize          0x0000000000001000
     clumpSize            0x00001000
     totalBlocks          0x00000001
     extents              startBlock blockCount
                          0x00000001 0x00000001
```

Abbildung 15: hdiutil Ausgabe der Volume Header Analyse (Auszug)

TSK bietet bei der Analyse des Volume Headers zwar einige Informationen, liefert jedoch keine Angaben über Fork Daten der Special Files im HFS Plus Dateisystem.

X-Ways analysiert ebenfalls den Volume Header und ermöglicht über eine HFS Plus Volume Header Schablone die Analyse sämtlicher Informationen im Header. Zu beachten ist hier, dass die Darstellung der Zeitstempel nicht angibt, welche Angaben in UTC bzw. in Lokalzeit im Header gespeichert sind.

Die Besonderheit bei der Analyse des Volume Headers mit BlackLight ist, dass für die Felder *File Count* und *Folder Count* nicht nur die Angaben aus dem Volume Header verwendet werden, sondern auch die *Special Files* mitgezählt werden. Diese Dateien werden im Eintrag *File Count* des HFS Plus Volume Headers nicht berücksichtigt.

In EnCase kann der HFS Plus Volume Header mit dem EnScript *HFS+ Volume Header Decoder (V2.0.0)* vom Guidance Entwickler Simon Key analysiert werden. Sämtliche Header Informationen werden automatisiert analysiert und die Ergebnisse in der Konsole und als Lesezeichen im Programm EnCase ausgegeben.

FTK stellt keine automatisierte Analyse des Volume Headers bereit. Hier muss im Programm manuell mit einem Hex Editor analysiert werden.

5.1.2 HFS Wrapper

Ein HFS Plus Volume kann in einem HFS Wrapper eingebettet sein.[64] Apple ermöglicht damit den Support bei älteren Mac OS Versionen (Mac OS 8.1 und früher) mit dem HFS Dateisystem den Start von HFS Plus Volumes. Beim Einbinden von HFS Plus Volumes in Mac OS Systeme, die HFS und nicht HFS Plus unterstützen, wird der HFS Wrapper als ein Volume eingebunden. Dieses Volume beinhaltet eine Datei „ReadMe“, die erläutert, wie auf die Daten des eingebetteten HFS Plus Volumes zugegriffen werden.[65]

Bei der Implementierung eines HFS Wrappers erscheint der HFS Plus Volume Header nicht am Byte-Offset 1024 vom Anfang des HFS Plus Volumes. An dieser Stelle liegt dann der HFS Master Directory Block (MDB), der dem HFS Plus Volume Header ähnlich ist. Aus den Informationen des MDBs lassen sich Informationen wie die Position des HFS Plus Volume Headers ermitteln.[66]

Folgende Abbildung gibt einen Überblick über die Volume Struktur mit HFS Wrapper:[67]

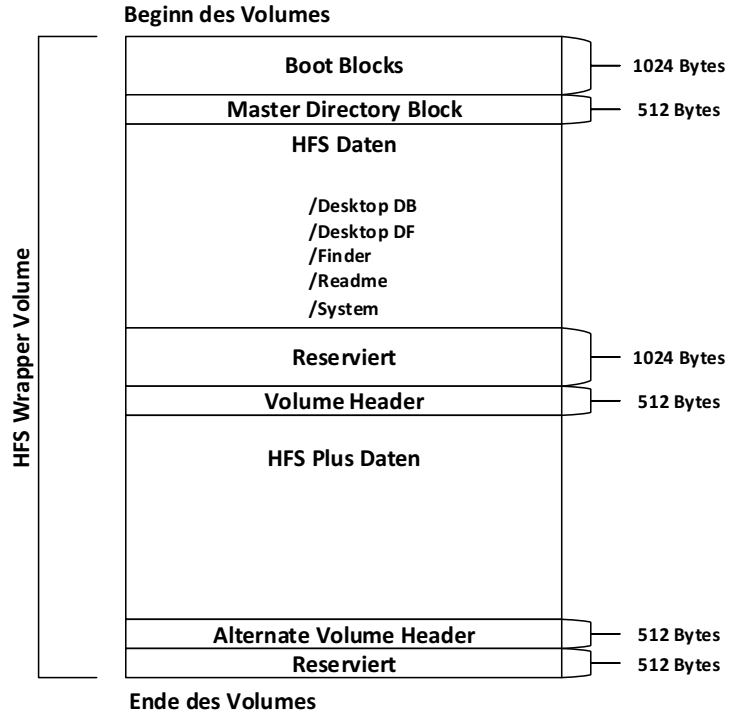


Abbildung 16: HFS Plus Volume in einem HFS Wrapper Volume eingebettet[68]

Nachfolgend wird der Aufbau des Master Directory Blocks des HFS Wrappers dargestellt.

Offset in Byte	Größe in Byte	Beschreibung

[64] Singh 2007, S. 1501
[65] Apple: *Technical Note TN1150*, S. 43
[66] Singh 2007, S. 1502
[67] Singh 2007, S. 1503
[68] Singh 2007, S. 1503

0	2	Volume Signatur
2	4	Erstellungsdatum des Volumes
6	4	Änderungszeitpunkt
10	2	Volume Attribute
12	2	Anzahl der Dateien im Volume
14	2	Start Block der Volume Bitmap
16	2	Start der nächsten Allokationssuche
18	2	Anzahl Allocation Blocks
20	4	Größe der Allocation Blocks in Byte
24	4	Default Clump Größe
28	2	Der erste Allocation Block im Volume in 512 Byte-Sektoren
30	4	Nächste unbenutzte Catalog Node ID
34	2	Anzahl der unbenutzten Allocation Blocks
36	28	Name des Volumes (Das erste Byte steht für die Länge des Namens, welche nicht größer als 27 Bytes sein kann.)
64	4	Zeitstempel vom letzten Backup
68	2	Volume Backup Sequenz Nummer
70	4	Anzahl der Schreibzugriffe auf das Volume
74	4	Clump Größe vom Extent Overflow File
78	4	Clump Größe vom Catalog File
82	2	Anzahl der Ordner im Root Verzeichnis
84	4	Anzahl der Dateien im Volume
88	4	Anzahl der Verzeichnisse im Volume
92	32	Finder Informationen; Array mit acht 32 Bit Werten
124	2	H+ oder HX Signatur des eingebetteten HFS Plus Volumes
126	4	
130	4	Extent Descriptor – beschreibt den Start des eingebetteten Volumes
134	4	Größe des Extents Overflow Files
138	12	Extent Eintrag – beschreibt Größe und Position des Extents Overflow Files
150	4	Größe des Catalog Files
154	12	Extent Eintrag – beschreibt die Größe und Position des Catalog Files

Tabelle 11: Master Directory Block eines HFS Wrappers

Wenn ein HFS Wrapper Volume von einem Mac OS System angelegt wird, werden fünf Dateien im Root Verzeichnis erstellt: die zuvor erwähnte *ReadMe*-Datei; eine *System*datei, die für den Bootvorgang benötigt wird; eine *Finder*-Datei; eine *Desktop DF*- und eine *Desktop DB*-Datei. [69]

[69] Varsalone et al. 2009, S. 102–105

Wird ein mit HFS Wrapper eingebettetes HFS Plus Volume in einem Mac OS System 8.1 oder früher eingebunden, wird der Speicherbereich des HFS Plus Volumes als „bad“ markiert. Dies verhindert, dass das HFS Plus Volume beschädigt oder anderweitig benutzt wird.[70]

5.1.3 Reservierte Bereiche

Die ersten zwei logischen Blocks (1024 Bytes) und der letzte logische Block (512 Bytes) vom HFS Plus Volume sind reserviert. Diese Bereiche werden von aktuellen Mac OS X Systemen nicht genutzt. Pre Mac OS X Systeme haben die ersten 1024 Bytes als Boot-Block genutzt. Die letzten 512 Bytes am Ende des Volumes sind von Apple für die Systemwiederherstellung vorgesehen.[71]

5.1.4 Extents

Ein *Extent* ist eine Reihe von zusammenhängenden Allocation Blocks, die in HFS Plus in einer *Extent Descriptor*-Datenstruktur dargestellt werden. Ein *Extent Descriptor* enthält ein Nummernpaar, die Allocation Block Nummer des Anfangs der zusammenhängenden Reihe und die Anzahl der Allocation Blocks der zusammenhängenden Reihe. Ein Array von acht *Extent Descriptors* bildet einen Extent Eintrag. HFS Plus nutzt dies zur Speicherung von Dateiinhalten. Die ersten acht Extents einer Datei sind als Teil der grundlegenden Metadaten im *Catalog File* gespeichert. Hat eine Datei mehr als acht Extents, werden zusätzliche Extent Einträge außerhalb dieser Metadaten im *Extents Overflow File* gespeichert.[72]

5.1.5 File Forks und Double Files

In einem HFS Plus Dateisystem besteht eine Datei aus zwei sogenannten Forks. Hierbei handelt es sich um Datenströme; einen Resource Fork und einen Data Fork. Beide Forks haben Extent Einträge, die in den Standard Metadaten der jeweiligen Datei gespeichert sind. Resource Fork und Data Fork einer Datei können jeweils oder beide eine Größe von 0 Bytes haben.[73]

Der Data Fork beinhaltet den Dateiinhalt, der Resource Fork zusätzliche Informationen zur Datei, wie zum Beispiel das für das Öffnen der Datei bevorzugte Programm, die Größe und Position des Programmfensters sowie Icons.

Auf anderen Dateisystemen als HFS Plus oder HFS werden Resource Forks nicht unterstützt. Aus diesem Grund erstellt Mac OS X zusätzliche Dateien, die die Resource Fork Daten enthalten.[74] Diese Dateien werden auch *Apple Double Files* genannt. Sie können neben den *Resource Forks* auch *Extended Attributes* enthalten, die auf dem HFS Plus Volume im *Attributes File* gespeichert sind. Jedes *Apple Double File* hat dieselbe Bezeichnung wie die Datei, deren zusätzliche Informationen es enthält, und dem Prefix „._“.

Die folgende Abbildung zeigt den Inhalt eines HFS Plus Volumes des Beispielimages „image_2“ im Finder eines Mac OS X Systems.

---

[70] Singh 2007, S. 1505
[71] Singh 2007, S. 1493
[72] Singh 2007, S. 1480
[73] Singh 2007, S. 1480–1481
[74] Varsalone et al. 2009, S. 98

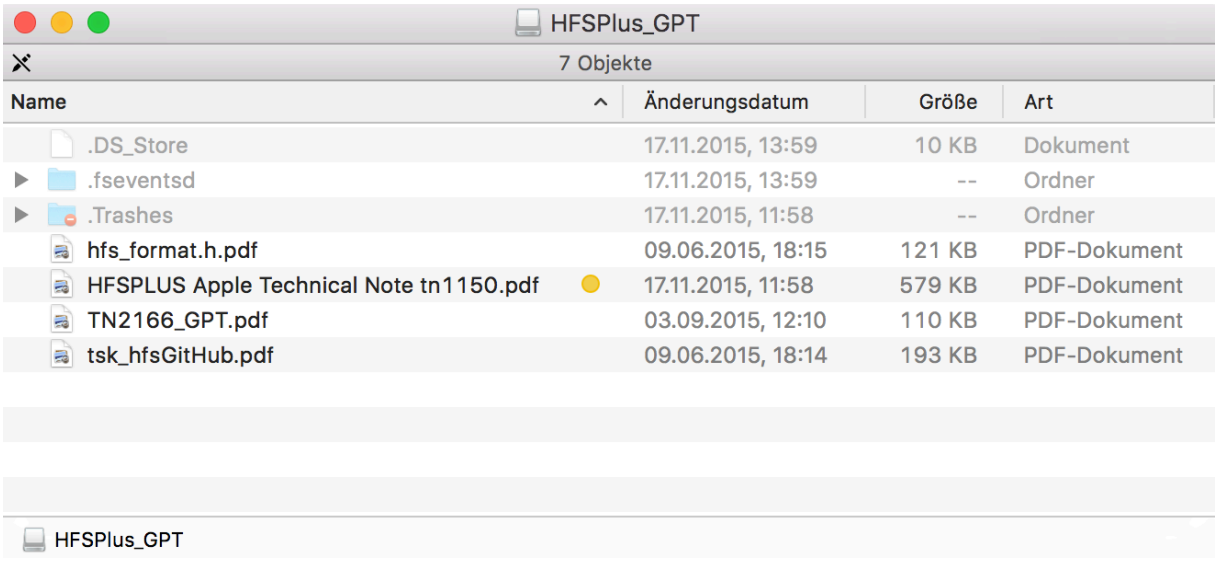


Abbildung 17: Dateiauflistung im Mac OS X Finder

Die Datei „HFSPlus Apple Technical Note tn1150.pdf“ (mit gelben Tag im Finder markiert) enthält die zusätzliche Information, dass sie als Standard mit dem Programm „ForkLift.app“ geöffnet werden soll. Diese Information ist im Resource Fork der Datei gespeichert.

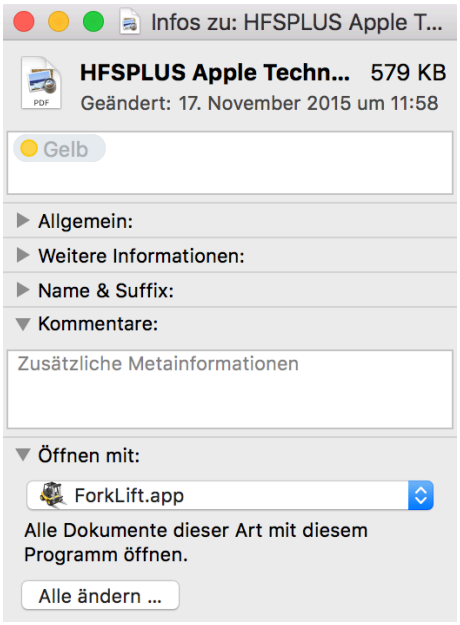


Abbildung 18: Information zur Datei "HFSPLUS Apple Technical Note tn1150.pdf"

Anschließend wurden die fünf gelisteten Dateien auf ein Volume mit FAT 32 Dateisystem kopiert und dieses in einem Windows 10-System mit NTFS Dateisystem eingebunden:

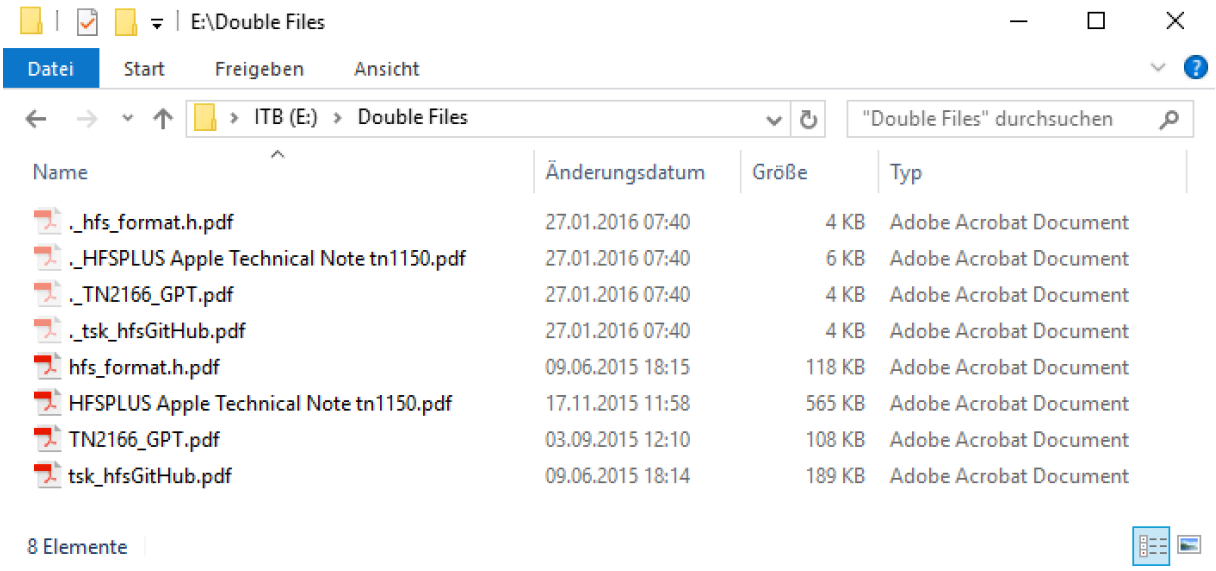


Abbildung 19: Dateiauflistung Windows 10 Explorer

Die Resource Fork Daten werden unabhängig der Größe mit „._“ als versteckte Dateien im Windows Explorer aufgelistet. Es werden hier auch Resource Fork Ströme mit einer Größe von 4 KB angezeigt, die keine Daten enthalten.

Als nächstes soll gezeigt werden, wie forensische Programme mit HFS Plus Datenströmen umgehen können und welche Möglichkeiten der Forensiker hat, die Data Fork und Resource Fork Daten zu analysieren.

Im Mac Terminal können mit Mac nativen Programmen die Resource Fork Daten erhoben werden. Im oben genannten Beispiel kann der Inhalt des aktuellen Verzeichnisses mit Attribut-Daten, die Auskunft über Metadaten geben, im Mac Terminal mit dem Befehl *#ls -@l* aufgelistet werden:

```
MacBook:HFSPlus_GPT Lars$ ls -@l
total 1984
-rwxr-xr-x@ 1 Lars  staff  577856 17 Nov 11:58 HFSPLUS Apple Technical Note tn1150.pdf
        com.apple.FinderInfo        32
        com.apple.ResourceFork    1338
        com.apple.LaunchServices.OpenWith          146
        com.apple.metadata:_kMDItemUserTags         51
        com.apple.metadata:kMDItemFinderComment    102
        com.apple.quarantine        23
-rw-r--r--@ 1 Lars  staff  109798  3 Sep 12:10 TN2166_GPT.pdf
        com.apple.metadata:kMDItemDownloadedDate            53
        com.apple.metadata:kMDItemWhereFroms       115
        com.apple.quarantine        25
-rw-r--r--@ 1 Lars  staff  120512  9 Jun  2015 hfs_format.h.pdf
        com.apple.quarantine        23
        com.dropbox.attributes      83
-rw-r--r--@ 1 Lars  staff  193300  9 Jun  2015 tsk_hfsGitHub.pdf
        com.apple.quarantine        23
        com.dropbox.attributes      83
```

Abbildung 20: Auflistung Attribut-Daten

Die Ausgabe zeigt, dass die Datei „HFSPlus Apple Technical Note tn1150.pdf“ Resource Fork Daten beinhaltet (siehe Markierung: com.apple.ResourceFork 1338). Anschließend kann mit

dem XCode-Kommandozeilen Programm *DeRez* der Inhalt der Resource Fork Daten decodiert werden:

```
MacBook:HFSPlus_GPT Lars$ DeRez HFSPLUS\ Apple\ Technical\ Note\ tn1150.pdf
data 'usro' (0) {
        $"0000 001B 2F41 7070 6C69 6361 7469 6F6E"            /* ..../Application */
        $"732F 466F 726B 4C69 6674 2E61 7070 0000"            /* s/ForkLift.app.. */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
        $"0000 0000 0000 0000 0000 0000 0000 0000"            /* ................ */
```

Abbildung 21: Decodierung mit DeRez (Auszug)

Alternativ können auch für die zu analysierende Datei im Terminal die Resource Fork Daten mit dem Suffix „/..namedfork/rsrc“ ausgegeben werden:

```
MacBook:HFSPlus_GPT Lars$ cat HFSPLUS\ Apple\ Technical\ Note\ tn1150.pdf/..namedfork/rsrc
2pplications/ForkLift.a22usro
```

Abbildung 22: Resource Fork Analyse im Terminal

Bei allen im Rahmen der Arbeit getesteten und bereits oben vorgestellten Forensik Tools wurde der Resource Fork Datenstrom der Datei „HFSPLUS Apple Technical Note tn1150.pdf“ erkannt und der Inhalt dargestellt.

5.1.6 Special Files

Im HFS Plus Volume Header ist die Fork Datenstruktur der *Special Files* enthalten. Dabei handelt es sich um drei Dateien mit B-Tree-Struktur[75] (Catalog File, Extents Overflow File, Attributes File), eine Bitmap-Datei (Allocation File) und ein optionales Startup File. Diese Dateien speichern die Dateisystemstrukturen, die für den Zugriff auf Dateien und Verzeichnisse benötigt werden.[76]

5.1.7 Zugriffsrechte

Mit der Einführung von Mac OS 10.4 wurden die Extended Attributes und mit ihnen auch Access Control Lists im Mac System implementiert.[77]

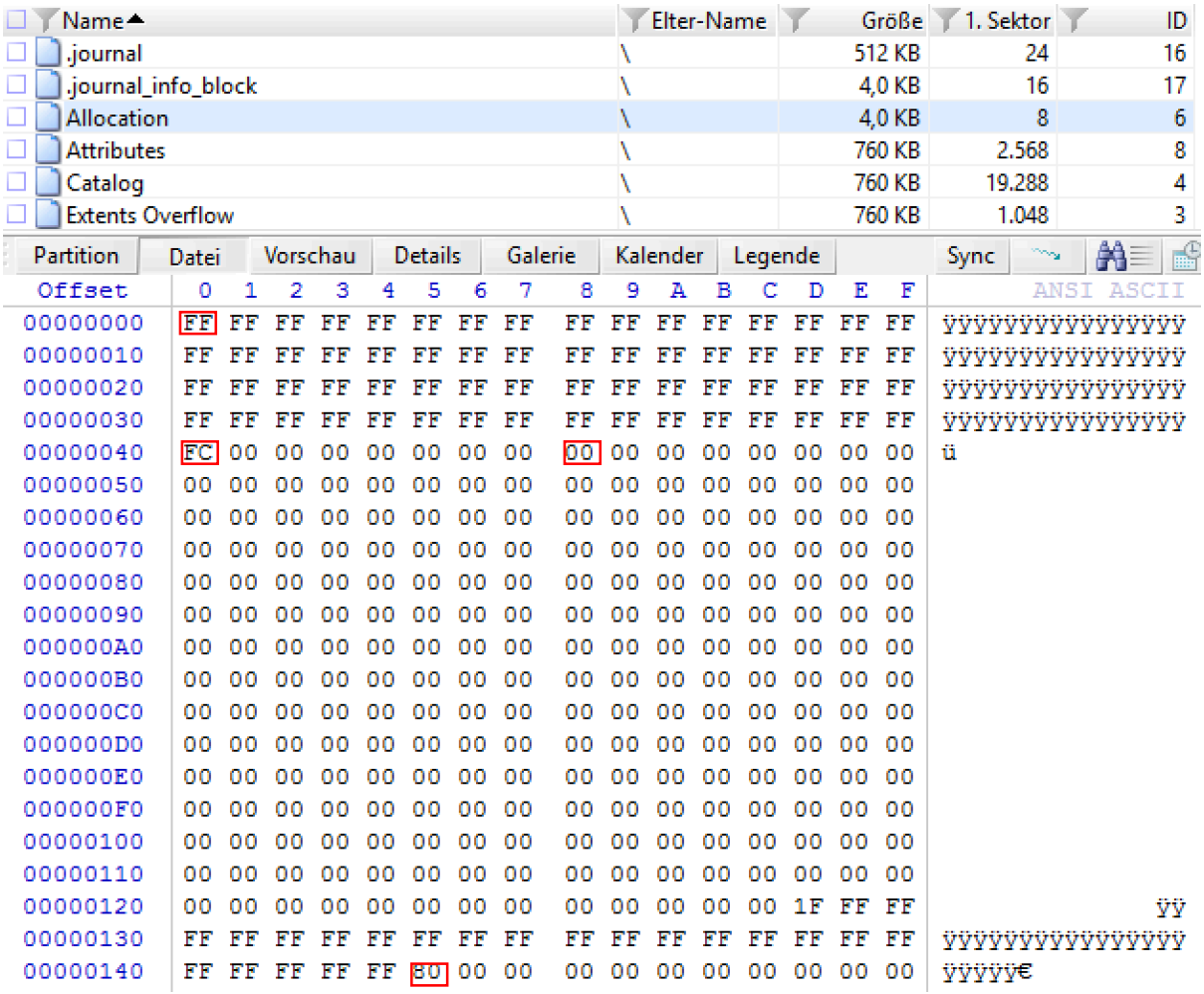
Zugriffsrechte werden in HFS Plus mit UNIX-Style Berechtigungen umgesetzt. Jede Datei und jeder Ordner beinhaltet eine BSD Informationsstruktur, die als Eintrag im Catalog File gespeichert ist.[78] Detaillierte Informationen dazu werden in der Kategorie Metadaten behandelt.

[75] Balance Tree
[76] Singh 2007, S. 1505
[77] Levin 2013, S. 608
[78] Varsalone et al. 2009, S. 99

5.2 Kategorie Inhalt

In der Kategorie Inhalt werden die Inhalte von Dateien und Verzeichnissen eines HFS Plus Dateisystems umfasst. Wie zuvor in der Arbeit dargestellt, werden Daten in Allocation Blocks zusammengefasst. Grundsätzlich besteht ein Allocation Block aus acht Blocks mit je einer Größe von 512 Bytes.

Die Verwaltung von Allocation Blocks übernimmt das *Allocation File*. Der Inhalt dieses *Allocation File*s ist eine Bitmap, die für jedes Bit einen Allocation Block im HFS Plus Volume beinhaltet und den Status verfolgt, ob ein Allocation Block allokiert oder nicht allokiert ist. Die Größe des *Allocation File*s ist von der Anzahl der Allocation Blocks des Volumes abhängig, die wiederum von der Größe des Datenträgers und der Größe der Allocation Blocks selbst abhängig sind.[79]



Name	Elter-Name	Größe	1. Sektor	ID
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17
Allocation	\	4,0 KB	8	6
Attributes	\	760 KB	2.568	8
Catalog	\	760 KB	19.288	4
Extents Overflow	\	760 KB	1.048	3

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	ANSI ASCII
00000000	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000010	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000020	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000030	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000040	FC	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	ü
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	1F	FF	FF	ÿÿ
00000130	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000140	FF	FF	FF	FF	FF	80	00	00	00	00	00	00	00	00	00	00	ÿÿÿÿÿ€

Abbildung 23: Beispiel eines Allocation Files (Auszug)

Die obere Abbildung zeigt einen Auszug eines *Allocation File*s. Jedes Byte repräsentiert acht Allocation Blocks. Ist der Bit-Wert=1, dann ist der entsprechende Allocation Block allokiert, ist der Bit-Wert=0, dann ist der entsprechende Allocation Block nicht allokiert. Folgende

[79] Apple: *Technical Note TN1150*, S. 27–28

Übersicht dokumentiert Beispielwerte aus der oben gezeigten Abbildung (rot eingerahmte Werte):

Hex	Binär	Allokation
0xFF	11111111	Alle Allocation Blocks sind allokiert.
0x00	00000000	Kein Allocation Block ist allokiert.
0xFC	11111000	Die drei höchsten Allocation Blocks sind nicht allokiert.
0x80	10000000	Der niedrigste Allocation Block ist allokiert.

Tabelle 12: Beispielwerte mit Allokations-Status eines Allocation Files

Eine Möglichkeit, das *Allocation File* auszuwerten, bietet FileXray:

```
MacBook:FileXray Lars$ sudo ./fileXray --device /dev/disk2s1 --allocation
  00000000  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
  0x000080  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
  0x000100  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
  0x000180  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
  0x000200  cf 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Abbildung 24: Auszug Auswertung eines Allocation Files mit FileXray

Die Darstellung aus FileXray vereinfacht die Bestimmung eines Allocation Blocks und dessen Allokationsstatus. Jede Zeile in der Ausgabe beginnt mit einer Allocation Block Nummer in Hexadezimal. Da jedes Byte acht Allocation Blocks (je Bit ein Allocation Block) repräsentiert, lässt sich so die Position und der Allokationsstatus jedes Allocation Blocks ermitteln (Offset X*8 bis X*8+7). FileXray zeigt im hexadezimalen Offset jeder Zeile bereits die Anzahl der Allocation Blocks, die von jedem Byte repräsentiert werden in hexadezimaler Ansicht (X*8).

Im oben gezeigten Beispiel repräsentiert das erste Byte am Offset 0 die Allocation Blocks 0 bis 7, am Offset 0x80 die Allocation Blocks 128 bis 135. Beide Byte-Positionen sind mit FF gekennzeichnet, da sie allokiert sind.

Mit der Option *--exhaustive* lassen sich detailliertere Informationen über die Allokation mit FileXray ausgeben:

```
MacBook:FileXray Lars$ sudo ./fileXray --device /dev/disk2s1 --allocation --exhaustive
  00000000  #AFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  0x000040  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
  0x000080  FFFEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
  0x0000c0  EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
  0x000100  EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
  0x000140  EXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  0x000180  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  0x0001c0  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXF
  0x000200  FF__FFFF________________________________________________________
  0x000240  ________________________________________________________________
```

Abbildung 25: Auszug der Auswertung eines Allocation Files mit Details in FileXray

Jede Zeile zeigt die Repräsentation von 64 Allocation Blocks. Allocation Block 0 ist mit # als reservierter Bereich gekennzeichnet. Wie zuvor beschrieben, nehmen reservierte Bereiche die ersten 1024 Bytes und die letzten 512 Bytes eines HFS Plus Volumes ein. Da die Größe eines Allocation Blocks im Beispiel 4096 Bytes (8 Block je 512 Bytes) beträgt und dies die kleinste allokierbare Dateneinheit darstellt, ist in diesem als reservierter Bereich markierter Allocation Block ebenfalls der Volume Header inbegriffen, der mit einer Größe von 512 Bytes am Byte-

Offset 1024 vom Anfang des Volumes beginnt. Folgende Legende aus dem *FileXray User Guide and Reference* veranschaulicht die Kennzeichnung der Allocation Blocks.[80]

#	Allocation Block ist reserviert oder vom Volume bzw. Alternate Volume Header belegt
_	Allocation Block ist unbenutzt
A	Allocation Block wird vom Allocation File belegt
C	Allocation Block wird vom Catalog File belegt
E	Allocation Block wird vom Extents Overflow File belegt
X	Allocation Block wird vom Attributes File belegt
F	Allocation Block wird auf anderer Weise belegt, z. B. von einer Datei
!	Bei der Erhebung der Besitzers des Allocation Blocks ist ein Fehler aufgetreten

Tabelle 13: Legende zur Kennzeichnung der Allocation Blocks bei Detailauswertung mit FileXray

Im Zusammenhang mit der Allokation zu beachten ist, dass die Allocation Blocks der reservierten Bereiche eines HFS Plus Volumes immer den Status allokiert haben müssen.[81]

5.2.1 Allokationsrhythmus

Um Fragmentierung zu verhindern, ist es in HFS Plus implementiert, dass bei einer Allokation von Dateien nach Clumps gesucht wird, um zusammenhängende Allocation Blocks einzelnen Allocation Blocks vorzuziehen.[82]

Um die Effizienz der Allokation weiter zu verbessern, wurde mit Mac OS X Lion die sogenannte Red-Black Tree-basierte Allokation eingeführt, bei der schneller zusammenhängende Allocation Blocks gefunden werden.[83]

Diese Art der Allokation entspricht der *Best-Fit*-Strategie.

5.2.2 Analyse-Szenario

Im folgenden Beispiel soll der Allocation Block 5121 im Beispielimage „image_2" untersucht werden. Als Annahme ergab eine String-Suche einen Treffer in diesem Bereich.

Um zu überprüfen, ob der Allocation Block mit Metadaten einer Datei allokiert ist, kann der Status für diesen Allocation Block im *Allocation File* ermittelt werden. Der Offset lässt sich wie folgt berechnen:

$n_{OF} = n_{AB} - n_{BO} * 8$

n_{AB}=Nummer des Allocation Blocks

n_{BO}=Byte-Offset im Allocation File, der aus $n_{AB}/8$ berechnet wird.

n_{OF}=Offset des Tracking-Bits des Allocation Blocks, der aus $n_{AB}-n_{BO}*8$ berechnet wird.

[80] iohead: *FileXray*, S. 39
[81] Apple: *Technical Note TN1150*, S. 28
[82] Singh 2007, S. 1481
[83] Levin 2013, S. 642

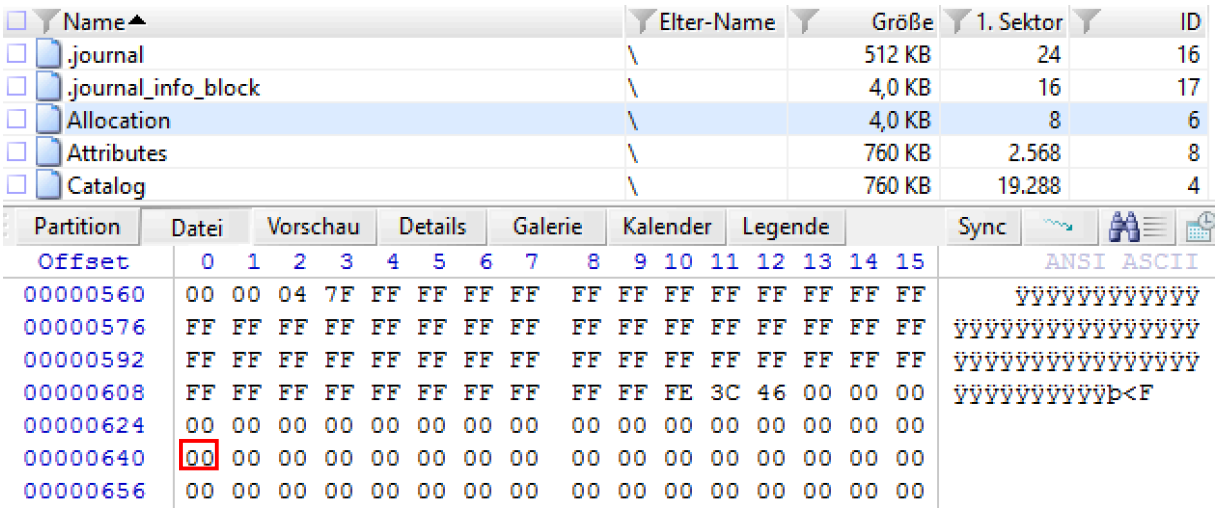
Folgende Daten liegen im Beispiel vor:

n_{AB}=5121

n_{BO}=5121/8=640

n_{OF}=5121-5120=1

Im Byte-Offset 640 des *Allocation Files* steht der Wert 0x00 = 0b00000000. Das erste Bit (n_{OF} =1) davon zeigt den Status des Allocation Blocks 5121. Da dies gleich 0 ist, ist dieser Allocation Block nicht allokiert.



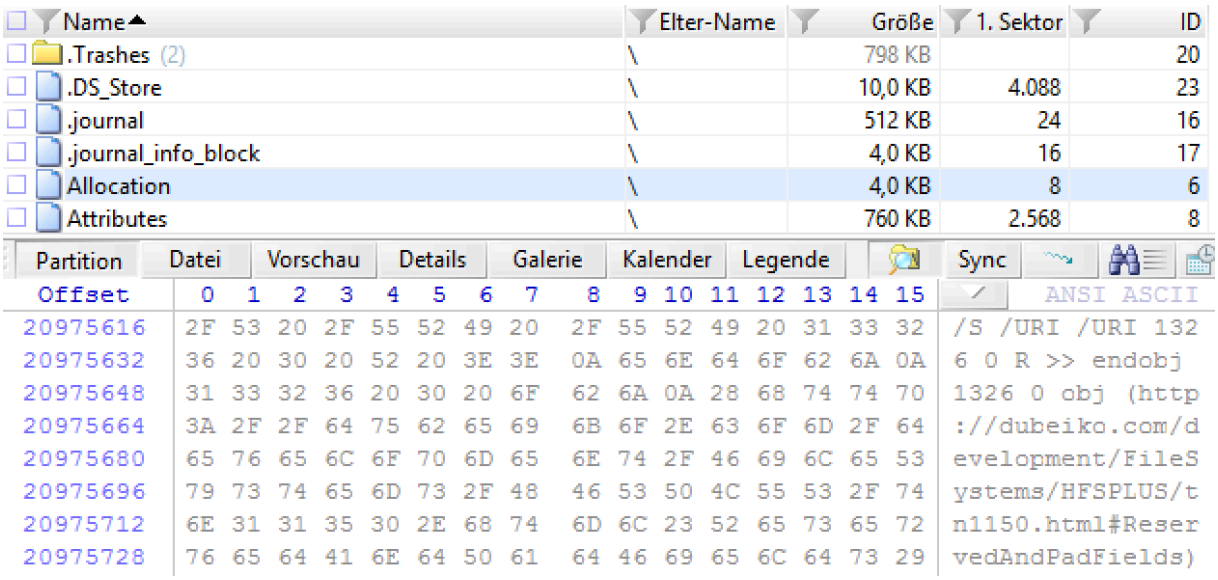
Name	Elter-Name	Größe	1. Sektor	ID
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17
Allocation	\	4,0 KB	8	6
Attributes	\	760 KB	2.568	8
Catalog	\	760 KB	19.288	4

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00000560	00	00	04	7F	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	..ÿÿÿÿÿÿÿÿÿÿ
00000576	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000592	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00000608	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FE	3C	46	00	00	00	ÿÿÿÿÿÿÿÿÿÿþ<F...
00000624	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000640	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000656	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Abbildung 26: Tracking im Allocation File für Allocation Block 5121

Der Inhalt des Allocation Blocks kann im Anschluss betrachtet werden. Da Inhaltsdaten vorhanden sind, stammen diese aus einer vorherigen Allokation und bieten Anhaltspunkte für weitere Untersuchungen:



Name	Elter-Name	Größe	1. Sektor	ID
.Trashes (2)	\	798 KB		20
.DS_Store	\	10,0 KB	4.088	23
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17
Allocation	\	4,0 KB	8	6
Attributes	\	760 KB	2.568	8

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
20975616	2F	53	20	2F	55	52	49	20	2F	55	52	49	20	31	33	32	/S /URI /URI 132
20975632	36	20	30	20	52	20	3E	3E	0A	65	6E	64	6F	62	6A	0A	6 0 R >> endobj.
20975648	31	33	32	36	20	30	20	6F	62	6A	0A	28	68	74	74	70	1326 0 obj (http
20975664	3A	2F	2F	64	75	62	65	69	6B	6F	2E	63	6F	6D	2F	64	://dubeiko.com/d
20975680	65	76	65	6C	6F	70	6D	65	6E	74	2F	46	69	6C	65	53	evelopment/FileS
20975696	79	73	74	65	6D	73	2F	48	46	53	50	4C	55	53	2F	74	ystems/HFSPLUS/t
20975712	6E	31	31	35	30	2E	68	74	6D	6C	23	52	65	73	65	72	n1150.html#Reser
20975728	76	65	64	41	6E	64	50	61	64	46	69	65	6C	64	73	29	vedAndPadFields)

Abbildung 27: Inhaltsdaten des Allocation Blocks 5121 (Auszug)

5.2.3 Analysetechniken

Die Analyse in dieser Kategorie umfasst wie oben dargestellt unter anderem die Untersuchung eines bestimmten Allocation Blocks, dessen Inhalts und Allokationsstatus.

Der Allokationsstatus einzelner Allocation Blocks kann mit *blkstat* aus dem TSK-Tool-Set abgerufen werden:

```
MacBook:Images Lars$ blkstat -o 40 image_2.E01 5121
Allocation Block: 5121
Not Allocated
```

Abbildung 28: Überprüfung des Allokationsstatus eines Allocation Blocks mit TSK

Im Ergebnis ist der Allocation Block 5121 nicht allokiert. Es folgt eine Überprüfung, ob noch Metadaten vorhanden sind:

```
MacBook:Images Lars$ ifind -d 5121 -o 40 image_2.E01
Inode not found
```

Abbildung 29: Überprüfung mit TSK, ob Metadaten zum Allocation Block 5121 vorhanden sind

Die Ausgabe zeigt, dass keine Metadaten zum Allocation Block 5121 vorhanden sind.

Als nächstes kann mit TSK der Inhalt analysiert werden:

```
MacBook:Images Lars$ blkcat -o 40 image_2.E01 5121 | xxd
0000000: 2f53 202f 5552 4920 2f55 5249 2031 3332  /S /URI /URI 132
0000010: 3620 3020 5220 3e3e 0a65 6e64 6f62 6a0a  6 0 R >>.endobj.
0000020: 3133 3236 2030 206f 626a 0a28 6874 7470  1326 0 obj.(http
0000030: 3a2f 2f64 7562 6569 6b6f 2e63 6f6d 2f64  ://dubeiko.com/d
0000040: 6576 656c 6f70 6d65 6e74 2f46 696c 6553  evelopment/FileS
0000050: 7973 7465 6d73 2f48 4653 504c 5553 2f74  ystems/HFSPLUS/t
0000060: 6e31 3135 302e 6874 6d6c 2352 6573 6572  n1150.html#Reser
0000070: 7665 6441 6e64 5061 6446 6965 6c64 7329  vedAndPadFields)
0000080: 0a65 6e64 6f62 6a0a 3238 3020 3020 6f62  .endobj.280 0 ob
0000090: 6a0a 3c3c 202f 4120 3133 3237 2030 2052  j.<< /A 1327 0 R
00000a0: 202f 426f 7264 6572 205b 2030 2030 2030   /Border [ 0 0 0
00000b0: 205d 202f 5479 7065 202f 416e 6e6f 7420   ] /Type /Annot
00000c0: 2f53 7562 7479 7065 202f 4c69 6e6b 202f  /Subtype /Link /
00000d0: 5265 6374 205b 3234 372e 3834 3637 2032  Rect [247.8467 2
00000e0: 3230 2e37 3739 3420 3237 392e 3838 3131  20.7794 279.8811
00000f0: 2032 3332 2e37 3932 335d 0a3e 3e0a 656e   232.7923].>>.en
0000100: 646f 626a 0a31 3332 3720 3020 6f62 6a0a  dobj.1327 0 obj.
```

Abbildung 30: Auszug Auflistung des Inhalts eine nicht-allokierten Allocation Blocks mit TSK

Als Ergebnis können Daten festgestellt werden, die noch aus einer vorherigen Allokation stammen.

Eine Möglichkeit der Analyse besteht darin, nach nicht-allokierten Allocation Blocks zu suchen und diese dann zu extrahieren, um anschließend Analysen in anderen Kategorien nach dem Referenzmodell von Brian Carrier durchführen zu können.

Folgende Abbildung zeigt eine Analyse mit FileXray, die die freien Allocation Blocks eines HFS Plus Volumes ermittelt.

```
MacBook:FileXray Lars$ ./fileXray --device /Users/Lars/Desktop/mnt/image_2.dmg --freespace
# Free Contiguous      Starting @        Ending @   Free Space
             1893           0x206           0x96a   7.8 MB
             1900           0xa29          0x1194   7.8 MB
                3          0x1196          0x1198   12 KB
                3          0x1357          0x1359   12 KB
                3          0x135e          0x1360   12 KB
                3          0x1362          0x1364   12 KB
            19442          0x1367          0x5f58   80 MB

# Allocation block size   = 4096 bytes
# Allocation blocks total = 24410 (0x5f5a)
# Allocation blocks free  = 23247 (0x5acf)
```

Abbildung 31: Übersicht zusammenhängender freier Allocation Blocks des Beispielimages

Im obigen Beispiel werden neben wichtigen Informationen wie Allocation Block-Größe, Gesamtzahl der Allocation Blocks und wieviel davon frei sind auch die Anzahl der jeweils zusammenhängenden freien Allocation Blocks mit Start- und Ende-Offset angegeben. Diese Ausgabe ermöglicht eine gezielte Suche (insbesondere bei bekannter Allokationsstrategie des Systems) in nicht-allokierten Speicherbereichen in der Kategorie Anwendung nach Carrier.

Wenn jetzt die Analyse mit FileXray berücksichtigt wird, kann ermittelt werden, wie viele freie Allocation Blocks an Allocation Block 5121 angrenzen und so eventuell eine vollständige Datei, die nicht mehr allokiert ist, wiederhergestellt werden (Zum Beispiel kann *blkcat* aus dem TSK-Tool-Set verwendet werden. Hier kann der relevante Allocation Block mit einer Anzahl folgender Allocation Blocks, die wiederhergestellt werden sollen, angegeben werden: *#blkcat –o 40 image_2.E01 5121 8 > recovered_file.pdf*.).

5.3 Kategorie Metadaten

Für die Analyse von Metadaten in einem HFS Plus Dateisystem spielen die sogenannten *Special Files* eine wesentliche Rolle. Dazu zählen für die Kategorie der Metadaten das *Catalog File*, das *Events Overflow File* sowie das *Attributes File*. Alle drei Dateien sind in einer B-Tree Struktur aufgebaut.

Ähnlich wie bei einem NTFS Dateisystem Metadateneinträge in der Master File Table (MFT) gespeichert werden, finden sich Metainformationen über die Dateien im HFS Plus Dateisystem als Catalog Eintrag im *Catalog File* wieder. Im Gegensatz zur MFT haben die Einträge im *Catalog File* keine fixe Länge, was es schwieriger macht, sie zu lokalisieren. Dazu kommt, dass das *Catalog File* aufgrund der B-Tree Struktur sich ständig reorganisiert.

Haben Catalog File Einträge mehr als acht Extents (je Resource Fork und Data Fork), werden sie im *Extents Overflow File* weitergeführt.

Das *Attributes File* beinhaltet zusätzliche Metadaten einer Datei oder eines Ordners.

5.3.1 B-Tree-Struktur

Bevor auf die Möglichkeiten der Analyse eingegangen werden kann, muss die B-Tree-Struktur erläutert werden. Diese generelle B-Tree-Struktur wird von HFS Plus für das *Catalog File*, das *Extents Overflow File* und für das *Attributes File* verwendet.

Jede B-Tree-Datei ist in Nodes mit fester Größe unterteilt, die Einträge enthalten. Offsets von Nodes können anhand ihrer festen Größe und der Node ID ähnlich wie Einträge in der MFT aufgesucht werden. Ein Node kann mehrere Einträge enthalten, die keine feste Größe haben müssen. Die Einträge bestehen aus einem Key und dazugehörigen Daten. Mit dieser Struktur soll ein effizientes Mappen von Keys und den dazugehörigen Daten ermöglicht werden. Dies wird durch Vergleiche der Keys (Vergleich nach Nummerngröße) erreicht.[84] Die Key-Daten-Struktur variiert bei HFS Plus je nach *Special File*.

[84] Apple: *Technical Note TN1150*, S. 13

Jeder Node in einem B-Tree beginnt mit einem *Node Descriptor*. Dieser ist 14 Bytes groß und beinhaltet Informationen über den Node, wie zum Beispiel den Node Typ und die Anzahl der Einträge im Node.

Das Layout eines B-Tree Node Descriptors ist wie folgt aufgebaut:

Offset (Byte)	Größe (Byte)	Beschreibung
0	4	ID des nächsten Node gleichen Typs (*blink*)
4	4	ID des vorherigen Node gleichen Typs (*flink*)
8	1	Node Typ
9	1	Level/Tiefe des Nodes in der B-Tree Hierarchie
10	2	Anzahl der Einträge
12	2	Reserviert

Tabelle 14: B-Tree Node Descriptor Struktur

Am Offset 8 des Node Descriptors wird mit einem Byte angegeben, ob es sich um ein Leaf-, Index-, Header- oder Map Node handelt:

Wert	Beschreibung
-1	Leaf Node (0xFF)
0	Index Node
1	Header Node
2	Map Node

Tabelle 15: Node Typ (Offset 8 in Node Descriptor-Struktur)

Mögliche Werte als Node Level am Offset 9 des Node Descriptors können der folgenden Tabelle entnommen werden:

Wert	Beschreibung
0	Header Node
1	Leaf Node
2	Index Node, der auf ein Leaf Node zeigt

Tabelle 16: Node Level (Offset 9 in Node Descriptor-Struktur)

Die Node-Einträge können über 2-Byte große Offsets gefunden werden, die am Ende des Nodes beginnen. Jeder Offset beschreibt den Start eines Eintrags vom Anfang des Nodes, wobei die Offsets in umgekehrter Reihenfolge gespeichert sind – also vom Ende des Nodes, der den ersten Eintrag im Node spezifiziert und so weiter. Ein zusätzlicher Offset zeigt nicht auf einen Node Eintrag sondern auf freien Speicher im Node, der nach dem letzten Eintrag beginnt. Es können auch weitere Offsets vorhanden sein, die nicht mehr relevant sind. Wie zuvor beschrieben, lässt sich die Anzahl der relevanten Offsets aus dem *Node Descriptor* ermitteln.[85]

Folgende Abbildung zeigt eine B-Tree Node-Struktur:

[85] Apple: *Technical Note TN1150*, S. 13–15

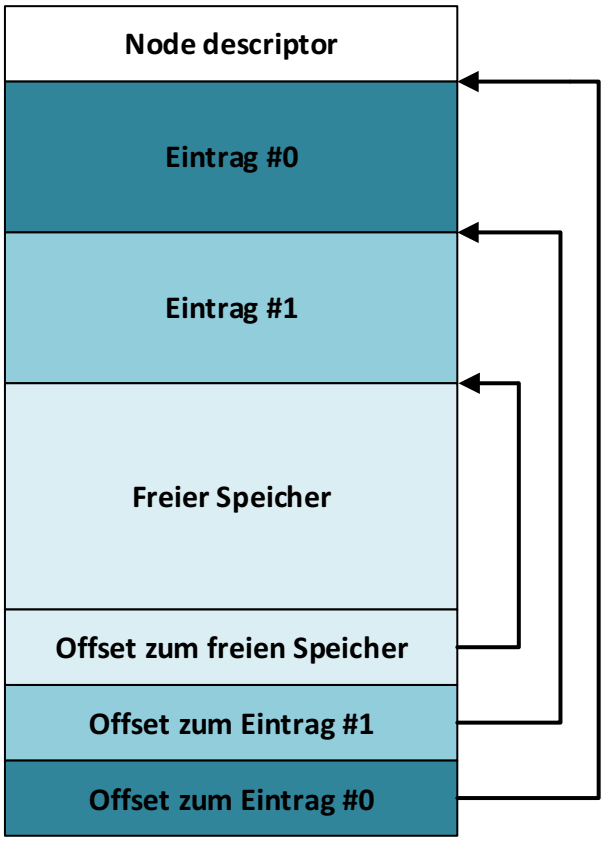


Abbildung 32: Node Struktur im B-Tree

In einer B-Tree-Struktur gibt es eine Node Hierarchie, die aus Header Node, Root Node, Index Nodes, Map Nodes und Leaf Nodes besteht. Im Node Descriptor lässt sich der Level[86] des jeweiligen Nodes in der B-Tree Hierarchie sowie die ID des nächsten und vorherigen Nodes gleichen Typs ablesen, der in der Node Hierarchie auf dem gleichen Level liegt.

Der erste Node in der Struktur ist immer der **Header Node** mit der ID 0 und hat den Node Level 0. Dieser Node beinhaltet Informationen, wie zum Beispiel die Größe der Nodes einer B-Tree-Datei sowie die ID des Root Nodes. Diese Informationen werden benötigt, um Einträge in der B-Tree-Datei zu lokalisieren. Ein Header Node besteht immer aus drei Einträgen, die nach dem *Node Descriptor* im Datei-Offset 14 folgen. Bis auf den Header Node enthalten alle anderen Nodes im B-Tree entweder Index- oder Leaf-Einträge. Ein Node kann niemals Index- und Leaf-Einträge zusammen beinhalten. Bei Nodes mit Index-Einträgen handelt es sich demnach um **Index Nodes** und bei Nodes mit Leaf-Einträgen um **Leaf Nodes**. Die Leaf Einträge der Leaf Nodes enthalten die Daten, die zur B-Tree Datei gehören. Index Nodes werden benötigt, um die Leaf Nodes zu finden. Sie enthalten Index Einträge mit der ID des nächsten Nodes. Der nächste Node in der Hierarchie nach dem Header Node ist der **Root Node**. Dieser hat immer den höchsten Level in der Hierarchie. Er kann Index Node sein, der auf andere Index Nodes oder auf Leaf Nodes verweist. Der Root Node kann aber auch Leaf Node sein, wenn die Anzahl der Node Einträge so gering ist, dass sie in einen Node gespeichert werden können.

[86] In der Fachliteratur werden unterschiedliche Bezeichnungen für die Angabe von Nodes in der Hierarchie des B-Trees verwendet. Genutzt werden die Begriffe Level, Tiefe und Höhe mit derselben Bedeutung. In dieser Arbeit wird die Bezeichnung *Node Level* benutzt.

Wenn ein Index Node direkt auf Leaf Nodes zeigt, hat er den Node Level 2. Leaf Nodes haben immer den Node Level 1.[87]

Sollten alle Leaf Einträge im gesamten B-Tree in den Root Node passen, dann werden keine weiteren Nodes benötigt, so dass es auch keine Index Nodes gibt. Sollte es mehr als einen Leaf Node geben, wird der Root Node zum Index Node und verweist auf die Leaf Nodes bzw. auf weitere Index Nodes.[88] Eine weitere Node-Art stellen die **Map Nodes** dar. Sie werden benötigt, wenn der Header Node keine Speicherkapazität mehr hat, um Informationen über die Zuordnung der Nodes in der B-Tree-Datei zu speichern. Map Einträge des Map Nodes funktionieren ähnlich der Bitmap des *Allocation Files* und kennzeichnen den Status von Nodes als allokiert oder nicht-allokiert. Die Struktur der Map Nodes ist mit der zuvor beschriebenen Struktur der anderen Node-Typen identisch.[89]

Folgende Abbildung veranschaulicht eine Node Hierarchie im B-Tree:

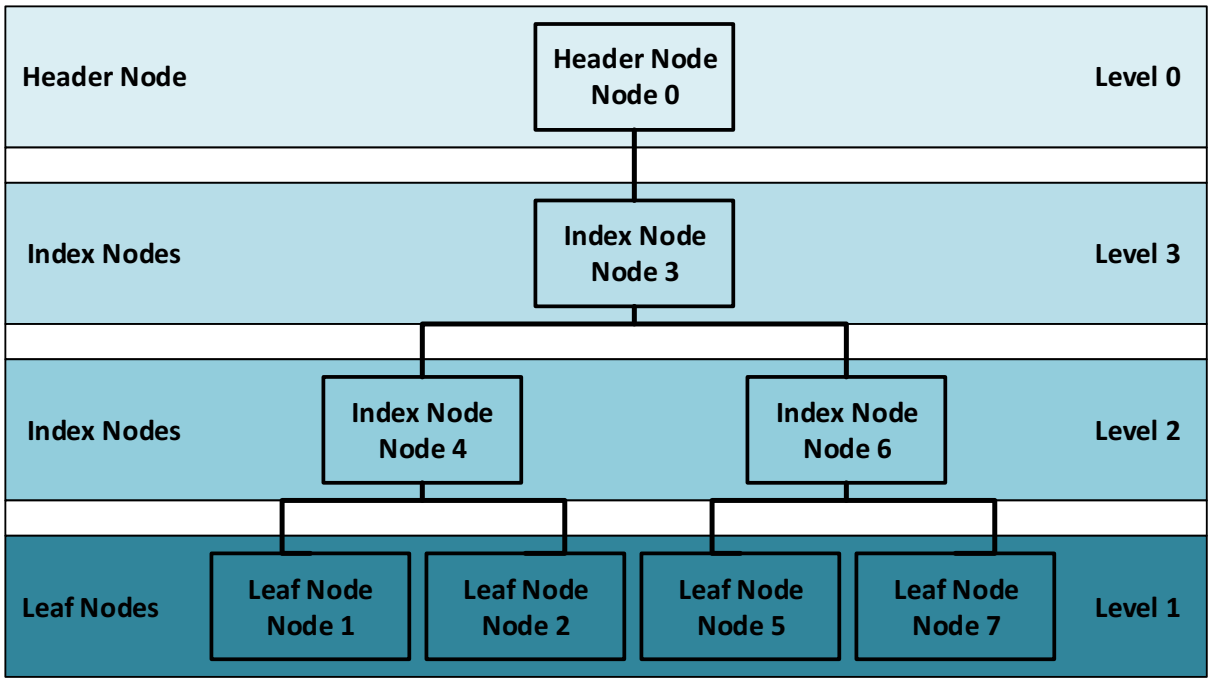


Abbildung 33: B-Tree Hierarchie

[87] Singh 2007, S. 1488–1490
[88] Apple: *Technical Note TN1150*, S. 15–17
[89] Apple: *Technical Note TN1150*, S. 16

Die Struktur des Header Nodes ist in jedem B-Tree in einem HFS Plus Dateisystem identisch. Nach dem Node Descriptor von Offset 0 bis Offset 13 beginnt der Header Eintrag, der wie folgt aufgebaut ist:

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	B-Tree-Tiefe
2	4	Root Node-ID
6	4	Anzahl der Leaf Einträge in der B-Tree-Datei
10	4	ID des ersten Leaf Nodes
14	4	ID des letzten Leaf Nodes
18	2	Größe der Nodes in der B-Tree-Datei
20	2	Maximale Key Länge für die Einträge in den Index- und Leaf Nodes
22	4	Anzahl der Nodes in der B-Tree-Datei
26	4	Anzahl der freien Nodes in der B-Tree-Datei
30	2	Reserviert
32	4	Clump Größe (Dieser Wert wird nicht von HFS Plus B-Trees genutzt.)
36	1	B-Tree Typ (0 für Catalog, Extents Overflow und Attributes B-Tree-Dateien)
37	1	Key-Vergleichstyp (Wird nur für HFSX Volumes genutzt.)
38	4	Attribute. Bei dieser Struktur handelt es sich um eine Bit-Flag-Maske.
42	64	Reserviertes Array mit 16x 4-Byte Integer-Werten

Tabelle 17: Struktur eines B-Tree Header Eintrags[90]

Diese Informationen geben Auskunft über die gesamte B-Tree Struktur und sind notwendig für die Suche in diesem B-Tree.

5.3.1.1 Suche im B-Tree

Wie zuvor erwähnt, beginnt die Analyse eines B-Trees immer im Header Node, da dieser die erste bekannte Position im B-Tree darstellt und immer am Offset 0 der B-Tree-Datei beginnt. Wie bereits dargestellt, beginnt auch dieser Node mit dem 14 Bytes großen Node Descriptor (Header Node Descriptor), der Auskunft über den Node Typ und den Node Level gibt (siehe Tabellen 14-16). Der Wert für die ID des nächsten und vorherigen Nodes gleichen Typs ist bei einem Header Node immer jeweils 0. Um einen Eintrag im B-Tree lokalisieren zu können, sind die Informationen aus dem B-Tree Header Eintrag, der direkt nach dem Node Descriptor am Offset 14 folgt, insbesondere über die Größe der Nodes im B-Tree und die ID des Root Nodes notwendig. Damit lässt sich die Position des Root Nodes feststellen. Ist der Root Node ein Index Node, können entweder weitere Index Nodes oder Leaf Nodes von hier aus ermittelt werden.

Die Suche nach Node Einträgen in einem B-Tree wird durch Key-Vergleiche umgesetzt.

[90] Apple: *Technical Note TN1150*, S. 15

Alle Keys in einem Node sind in aufsteigender Reihenfolge gespeichert. Sämtliche Nodes innerhalb eines Levels sind so angeordnet bzw. miteinander verbunden, dass der Node mit den kleinsten Keys in einem Level der erste Node und der Node mit den größten Keys der letzte Node in selben Level ist. Unter dieser Voraussetzung sind alle Keys in einem Node kleiner als alle Keys im nächsten Node im selben Level. Anders herum sind alle Keys in einem Node größer als alle Keys im vorherigen Node im selben Level.[91] Anhand der Node Descriptors können die IDs der nächsten und vorherigen Nodes, die auf dem gleichen Level miteinander verkettet sind, entnommen werden.

Die Suche nach einem bestimmten Key beginnt im Root Node. Hier wird nach einem Eintrag mit dem größten Key, der aber kleiner oder gleich dem gesuchten Key ist, gesucht. Von diesem Eintrag aus wird der entsprechend nachfolgende Node aufgesucht. Ist dies ebenfalls ein Index Node wird die Suche mit der gleichen Prozedur fortgesetzt, bis ein Leaf Node gefunden wird, der einen Eintrag mit dem passenden Key enthält.[92]

Nachfolgend werden die Datenstrukturen der speziellen B-Tree-Dateien im HFS Plus Dateisystem behandelt und an Beispielszenarien Analysemöglichkeiten vorgestellt.

5.3.2 Catalog File

Das *Catalog File* beschreibt die Verzeichnis- und Dateihierarchie eines HFS Plus Volumes in einer B-Tree Struktur, speichert Metainformationen über sämtliche Dateien und Verzeichnisse eines HFS Plus Volumes und fungiert als deren Katalog. HFS Plus speichert Datei- und Verzeichnisnamen in Unicode Strings, die in der Apple Dokumentation *hfs_format.h* von einer *HFSUniStr255* Struktur mit 2-Byte Zeichen definiert sind.[93] [94]

Jede Datei und jeder Ordner eines HFS Plus Volumes besitzen eine einmalige Catalog Node ID (CNID), die bei der Erstellung der Datei oder des Ordners vergeben wird. Hierbei handelt es sich um einen 32 Bit-Integer Wert, der sequentiell allokiert ist. Definiert ist die CNID vom *CatalogNodeID* Datentyp. Für jede Datei oder jeden Ordner ist die Parent ID die CNID des Ordners (Parent Ordner), der die jeweilige Datei oder den jeweiligen Ordner beinhaltet. Die ersten 16 CNIDs sind von Apple für die *Special Files* reserviert (CNID 0 ist unbenutzt.). CNIDs können in allen Mac OS 9, Mac OS X 10.2 und später wieder benutzt werden.[95] [96]

CNID	Beschreibung
1	Parent ID des Root Verzeichnisses
2	ID des Root Verzeichnisses
3	ID des Extents Overflow Files
4	ID des Catalog Files
5	ID des Bad Block Files
6	ID des Allocation Files
7	ID des Startup Files

[91] Apple: *Technical Note TN1150*, S. 17
[92] Apple: *Technical Note TN1150*, S. 17
[93] Apple: *hfs_format.h* 2012, S. 2–4
[94] Singh 2007, S. 1510
[95] Apple: *Technical Note TN1150*, S. 18-19
[96] Singh 2007, S. 1510

8	ID des Attributes Files
14	ID des Repair Catalog Files (Wird temporär von fsck_hfs zur Wiederherstellung des Catalog Files genutzt.)
15	ID des Bogus Extent Files (Wird temporär während *ExchangeFiles* Operationen genutzt.)
16	Erste CNID für Dateien und Ordner des Benutzers

Tabelle 18: Übersicht der reservierten CNIDs

5.3.2.1 Catalog File Key

In einer B-Tree Struktur des *Catalog Files* werden Nodes durch Vergleich der IDs und Namen gefunden. Im *Catalog File* B-Tree existiert für jede Datei, jeden Ordner oder jeden Thread Eintrag ein Catalog Key Eintrag mit der ID des Parent Ordners und den Namen der Datei oder des Ordners. Catalog File Keys werden als erstes mit der Parent ID und dann mit dem Namen verglichen. Die Key-Struktur ist in der *Apple Technical Note TN1150* für den *HFSPlusCatalogKey* Typ, der in *hfs_format.h*[97] definiert ist, wie folgt beschrieben:[98]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Länge des Keys (UInt16)
2	4	Parent CNID
6	2	Länge des folgenden Dateinamens in Unicode Zeichen
8	Variabel	Name der Datei oder des Ordners im Parent Ordner (Unicode – HFSUniStr255)
8 + Variabel	4	Nächste Node ID (Dieses Feld existiert nur bei Index Nodes)

Tabelle 19: Key-Struktur eines Catalog File Eintrags

In einem Catalog Leaf Node können nach der Key-Struktur folgende Einträge folgen: [99]

- Catalog Datei-Eintrag
- Catalog Verzeichniseintrag
- Catalog Datei-Thread-Eintrag
- Catalog Verzeichnis-Thread-Eintrag

Der Typ jedes Eintrags wird nach der Key-Struktur in den ersten zwei Bytes beschrieben: [100]

Wert	Beschreibung
0x0001	HFS Plus Ordner-Eintrag
0x0002	HFS Plus Datei-Eintrag
0x0003	HFS Plus Ordner-Thread-Eintrag
0x0004	HFS Plus Datei-Thread-Eintrag

Tabelle 20: Typ eines Catalog File Eintrags (Offset 0 in der Struktur eines HFS Plus Catalog File-Eintrags)

[97] Apple: *hfs_format.h* 2012, S. 4
[98] Apple: *Technical Note TN1150*, S. 19–20
[99] Varsalone et al. 2009, S. 116
[100] Apple: *Technical Note TN1150*, S. 19–20

5.3.2.2 Catalog Dateieintrag

Catalog Dateieinträge haben eine feste Größe von 248 Bytes und beinhalten grundlegende Metainformationen zur zugehörigen Datei des Eintrags, wie zum Beispiel die CNID, Erstellungs- und Zugriffszeitstempel, Zugriffsrechte und Angaben zu den allokierten Dateneinheiten der Datei. Nachfolgende Tabelle zeigt die Struktur eines Catalog Datei-Eintrags:[101] [102]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Typ des Eintrags (0x02 für Dateieinträge)
2	2	Flags
4	4	Reserviert
8	4	Catalog Node ID
12	4	Erstellungszeitpunkt in HFS Plus Zeitstempelformat (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
16	4	Zeitpunkt des letzten Schreibzugriffs (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
20	4	Zeitpunkt des letzten Schreibzugriffs des Catalog Eintrags (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
24	4	Zeitpunkt des letztes Zugriffs (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
28	4	Zeitpunkt des letzten Backups (Wird nicht von Mac OS X genutzt.)
32	16	Zugriffsrechte
48	4	Dateityp (optional)
52	4	Ersteller (optional)
56	2	Finder Flags
58	4	Finder Fenster Position
62	2	Reserviert
64	8	Reserviert
72	2	Extended Finder Flags
74	2	Reserviert
76	4	Catalog Node ID des Original-Ordners, falls die Datei zum Desktop verschoben wurde.
80	4	Text Encoding
84	4	Reserviert
88	8	Data Fork - logische Größe
96	4	Data Fork - Clump Größe
100	4	Data Fork - Gesamtzahl der Allocation Blocks
104	4	Data Fork – Extent Nr. 1 - Start Allocation Block
108	4	Data Fork – Extent Nr. 1 – Anzahl der Allocation Blocks
112	4	Data Fork – Extent Nr. 2 - Start Allocation Block
116	4	Data Fork – Extent Nr. 2 – Anzahl der Allocation Blocks
120	4	Data Fork – Extent Nr. 3 - Start Allocation Block
124	4	Data Fork – Extent Nr. 3 – Anzahl der Allocation Blocks

[101] Apple: *hfs_format.h* 2012, S. 5
[102] Apple: *Technical Note TN1150*, S. 22

128	4	Data Fork – Extent Nr. 4 - Start Allocation Block
132	4	Data Fork – Extent Nr. 4 – Anzahl der Allocation Blocks
136	4	Data Fork – Extent Nr. 5 - Start Allocation Block
140	4	Data Fork – Extent Nr. 5 – Anzahl der Allocation Blocks
144	4	Data Fork – Extent Nr. 6 - Start Allocation Block
148	4	Data Fork – Extent Nr. 6 – Anzahl der Allocation Blocks
152	4	Data Fork – Extent Nr. 7 - Start Allocation Block
156	4	Data Fork – Extent Nr. 7 – Anzahl der Allocation Blocks
160	4	Data Fork – Extent Nr. 8 - Start Allocation Block
164	4	Data Fork – Extent Nr. 8 – Anzahl der Allocation Blocks
168	8	Resource Fork – logische Größe
176	4	Resource Fork – Clump Größe
180	4	Resource Fork – Gesamtzahl der Allocation Blocks
184	4	Resource Fork – Extent Nr. 1 – Start Allocation Block
188	4	Resource Fork – Extent Nr. 1 – Anzahl der Allocation Blocks
192	4	Resource Fork – Extent Nr. 2 – Start Allocation Block
196	4	Resource Fork – Extent Nr. 2 – Anzahl der Allocation Blocks
200	4	Resource Fork – Extent Nr. 3 – Start Allocation Block
204	4	Resource Fork – Extent Nr. 3 – Anzahl der Allocation Blocks
208	4	Resource Fork – Extent Nr. 4 – Start Allocation Block
212	4	Resource Fork – Extent Nr. 4 – Anzahl der Allocation Blocks
216	4	Resource Fork – Extent Nr. 5 – Start Allocation Block
220	4	Resource Fork – Extent Nr. 5 – Anzahl der Allocation Blocks
224	4	Resource Fork – Extent Nr. 6 – Start Allocation Block
228	4	Resource Fork – Extent Nr. 6 – Anzahl der Allocation Blocks
232	4	Resource Fork – Extent Nr. 7 – Start Allocation Block
236	4	Resource Fork – Extent Nr. 7 – Anzahl der Allocation Blocks
240	4	Resource Fork – Extent Nr. 8 – Start Allocation Block
244	4	Resource Fork – Extent Nr. 8 – Anzahl der Allocation Blocks

Tabelle 21: Struktur eines Catalog Dateieintrags

Am Offset 2 des Catalog Dateieintrags folgen Flags mit einer Länge von 2 Bytes und folgender Bedeutung:

Wert	Beschreibung
0x0001	Datei ist geschützt.
0x0002	Datei hat einen Thread-Eintrag.

Tabelle 22: Flags (Offset 2 in der Struktur eines HFS Plus Catalog File Eintrags)

Zeitstempel

Von Offset 12 bis 31 des Eintrags stehen Zeitstempel zur Datei des Catalog Eintrags, die im Apple Entwickler Dokument *TN1150* wie folgt beschrieben werden:[103]

- Der Erstellungszeitpunkt gibt Datum und Uhrzeit an, wann die Datei erstellt wurde.
- Der Zeitpunkt des letzten Schreibzugriffs gibt Datum und Uhrzeit an, wann die Datei vergrößert, verkleinert oder der Inhalt verändert wurde.
- Der Zeitpunkt des letzten Schreibzugriffs im Catalog Eintrag gibt Datum und Uhrzeit an, wann der Eintrag der Datei im Catalog File verändert wurde.
- Der Zeitpunkt des letzten Zugriffs gibt Datum und Uhrzeit an, wann der Dateiinhalt zuletzt gelesen wurde.
- Der Zeitpunkt des letzten Backups wird nicht vom Mac OS X System genutzt und stellt es Applikationen zur Verfügung. Es wird Datum und Uhrzeit des letzten Backups der Datei angegeben.

Alle Zeitstempel im Catalog Dateieintrag sind im HFS Plus Zeitformat in GMT abgelegt.

Zugriffsrechte

Am Offset 32 sind die HFS Plus Berechtigungen der jeweiligen Datei implementiert. Sie sind ähnlich nach POSIX Standard wie folgt definiert:

Offset (Byte)	Größe (Byte)	Beschreibung
0	4	Besitzer ID
4	4	Gruppen ID
8	1	Admin Flags (Können nur vom Super-User gesetzt werden.)
9	1	Besitzer Flags (Können nur vom Super-User oder Besitzer gesetzt werden.)
10	2	Datei-Mode
12	4	Union {UInt32 iNodeNum; UInt32 linkCount; UInt32 rawDevice;} special;

Tabelle 23: Berechtigungen (Offset 32 im HFS Plus Catalog File Eintrag)

Die Bereiche Benutzer- und Finder-Informationen beinhalten spezifische Daten zum Mac OS Finder, wie zum Beispiel die Position des Finder Fensters, den Dateityp oder die Ersteller-Codes.

Wie zuvor erwähnt, kann ein Catalog File Eintrag bis zu acht Data Fork Extents und bis zu acht Resource Fork Extents speichern. Informationen über die Position und Größe dieser Extents werden ab Offset 88 des Catalog Dateieintrags beschrieben. Werden mehr als acht Data Fork Extents oder mehr als acht Resource Fork Extents benötigt, werden diese im *Extents Overflow File* gespeichert.

[103] Apple: *Technical Note TN1150*, S. 23

5.3.2.3 Catalog Verzeichniseinträge

Catalog Verzeichniseinträge enthalten grundlegende Metainformationen über Verzeichnisse im HFS Plus Volume. Die Struktur ist der von Catalog Dateieinträgen sehr ähnlich und wurde gemäß *Apple Technical Note 1150* und *hfs_format.h* als *HFSPlusCatalogFolder* wie folgt definiert:[104] [105]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Typ des Eintrags (siehe Abbildung 46)
2	2	Flags (siehe Abbildung 47)
4	4	Anzahl der Dateien und Ordner, die diesen Ordner als übergeordneten Ordner haben (Patent).
8	4	Catalog Node ID
12	4	Erstellungszeitpunkt in HFS Plus Zeitstempelformat (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
16	4	Zeitpunkt des letzten Schreibzugriffs HFS Plus Zeitstempelformat (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
20	4	Zeitpunkt des letzten Schreibzugriffs des Catalog Eintrags HFS Plus Zeitstempelformat (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
24	4	Zeitpunkt des letztes Zugriffs HFS Plus Zeitstempelformat (Zeit in Sekunden vom 1. Januar 1904, 00:00:00 Uhr, GMT)
28	4	Zeitpunkt des letzten Backups (Wird nicht von Mac OS X genutzt.)
32	16	Zugriffsrechte (siehe Tabelle 23)
48	8	Benutzer- und Ordnerinformationen
56	2	Finder Flags (siehe Tabelle 22)
58	4	Ordner Position
62	2	Reserviert
64	4	Scroll Position im Finder Fenster
68	4	Reserviert
72	2	Extended Finder Flags
74	2	Reserviert
76	4	Catalog Node ID des Original-Ordners, falls der Ordner zum Desktop verschoben wurde.
80	4	Text Encoding
84	4	Reserviert

Tabelle 24: Struktur eines HFS Plus Catalog Ordner Eintrags

[104] Apple: *Technical Note TN1150*, S. 21
[105] Apple: *hfs_format.h* 2012, S. 5

Von Offset 12 bis 31 des Eintrags stehen Zeitstempel zum Ordner des Catalog File Eintrags, die im Apple Entwickler Dokument *TN1150* wie folgt beschrieben werden:[106]

- Der Erstellungszeitpunkt gibt Datum und Uhrzeit an, wann der Ordner erstellt wurde.
- Der Zeitpunkt des letzten Schreibzugriffs gibt Datum und Uhrzeit an, wann der Inhalt des Ordners verändert wurde; wann eine Datei oder ein Ordner im Ordner erstellt, gelöscht oder verschoben wurde.
- Der Zeitpunkt des letzten Schreibzugriffs im Catalog File Eintrag gibt Datum und Uhrzeit an, wann der Eintrag des Ordners im Catalog File verändert wurde.
- Der Zeitpunkt des letzten Zugriffs gibt Datum und Uhrzeit an, wann der Inhalt des Ordners zuletzt gelesen wurde.
- Der Zeitpunkt des letzten Backups wird nicht vom Mac OS X System genutzt und stellt es Applikationen zur Verfügung. Es wird Datum und Uhrzeit des letzten Backups des Ordners angegeben.

Alle Zeitstempel im Catalog Verzeichniseintrag sind im HFS Plus Zeitformat in GMT abgelegt.

5.3.2.4 Catalog Thread-Einträge

Catalog Thread-Einträge werden genutzt, um die CNID eines Verzeichnisses oder einer Datei mit dem Catalog File Eintrag der Datei oder des Verzeichnisses zu verknüpfen, die diese CNID nutzen.[107] Daher gibt es zwei Arten von Thread-Einträgen: Thread-Dateieinträge und Thread-Verzeichniseinträge. Die Struktur von Thread-Einträgen ist in *hfs_format.h* als *HFSPlusCatalogThread* definiert:[108]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Typ des Eintrags 0x03 – Verzeichnis Thread Eintrag 0x04 – Datei Thread Eintrag
2	2	Reserviert
4	4	Parent CNID
8	2	Länge des Datei- oder Verzeichnisnamens
10	Variabel	Datei- oder Verzeichnisname (HFSUniStr255)

Tabelle 25: Struktur Catalog Thread-Einträge

Zu beachten ist, dass sich die Key-Struktur bei Thread-Einträgen von Index Node Einträgen und Leaf Node Einträgen unterscheiden. Am Offset 0 des Thread-Keys stehen zwei Bytes für die Key-Länge. Als nächstes folgen vier Bytes für die CNID. Hierbei handelt es ich nicht um CNID des Parent Ordners, sondern um die CNID der entsprechenden Datei oder Ordners. Die folgenden zwei Bytes sind reserviert. Anschließend beginnt der Thread-Eintrag wie in Tabelle 25 dargestellt.

5.3.2.5 Suche im Catalog File

Eine Suche nach einer Datei oder einem Ordner im HFS Plus Dateisystem beginnt immer mit der Analyse des Volume Headers, der Informationen über die Position der Extents des *Catalog*

[106] Apple: *Technical Note TN1150*, S. 21–22
[107] Apple: *Technical Note TN1150*, S. 23–24
[108] Apple: *hfs_format.h* 2012, S. 5

Files enthält. Im *Catalog File* wird der Eintrag zur gesuchten Datei oder zum gesuchten Ordner ermittelt. Dieser enthält die Metadaten über die Datei oder über den Ordner und die Positionen der ersten acht Extents der Datei oder des Ordners.

Um Einträge im *Catalog File* finden zu können, werden zuerst die in den Keys gespeicherten Parent IDs und dann die gesuchten Datei- oder Ordnernamen verglichen. Für den Fall, dass alle Keys dieselbe Parent ID haben, kann durch den Vergleich der Namen der gesuchte Eintrag gefunden werden. Ist die Parent ID nicht bekannt, kann eine Suche nach der CNID der gesuchten Datei oder des gesuchten Ordners durchgeführt werden, um einen Thread-Eintrag zu finden (falls dieser existiert), der die Parent ID zur gesuchten Datei enthält. Anschließend kann anhand der im Thread-Eintrag gespeicherten Parent ID im *Catalog File* nach dem Key mit der Parent ID gesucht werden, der zum Eintrag der gesuchten Datei oder des gesuchten Ordners gehört. [109]

5.3.2.6 Analyse-Szenario

Im Beispielszenario wird die Datei „datai_a.txt“ aus dem Übungsimage „image_3_2“ analysiert. Ziel ist es, den Catalog File Eintrag der Datei zu finden und die Metadaten zu untersuchen.

```
MacBook:/ Lars$ ls -ilR /Volumes/fragmentation_3/
total 17776
24 -rw-r--r--@ 1 Lars  staff  3124576 30 Dez 11:51 datei_a.txt
25 -rw-r--r--@ 1 Lars  staff  5975424 30 Dez 11:36 datei_b.txt
```

Abbildung 34: Dateiauflistung von "image_3_2"

Aus der Dateiauflistung der Abbildung 34 ist ersichtlich, dass „datei_a.txt“ im Root Verzeichnis „fragmentation_3“ liegt und die CNID 24 besitzt. Da Catalog Node Einträge eine Key-Struktur im B-Tree mit variabler Größe haben, ist es im Gegensatz zu Einträgen in der MFT bei einem NTFS Dateisystem nicht möglich, den Offset anhand der CNID und der Eintragsgröße zu berechnen. Im HFS Plus Dateisystem werden die Keys und anschließend die Namen verglichen, um den relevanten Eintrag im *Catalog File* aufzufinden. Für diesen Vergleich ist es notwendig, die ID des Parent Ordners, in dem die entsprechende Datei gespeichert ist, zu erheben. Im Beispielfall ist es der Root Ordner mit der ID 2:

```
MacBook:/ Lars$ ls -il /Volumes/
total 8
13350405 lrwxr-xr-x  1 root  admin    1 18 Jan 09:52 Macintosh HD -> /
       2 drwxr-xr-x  9 Lars  staff  374 30 Dez 11:51 fragmentation_3
```

Abbildung 35: Root Ordner mit der ID 2

Folgende Daten, nach denen im *Catalog File* gesucht wird, sind nun bekannt:

- Dateiname: datei_a.txt
- CNID: 24
- Parent ID: 2

Wie zuvor dargestellt, sind in einem Node die Einträge aufsteigend nach ihrer Key-Größe gespeichert. In einem Node Level sind alle Keys kleiner als alle nachfolgenden Keys in den nachfolgenden Nodes. Um den richtigen Key zu finden, wird der Key verglichen, der kleiner als

[109] Apple: *Technical Note TN1150*, S. 20

der gesuchte Key ist und am dichtesten am Key-Wert liegt bzw. gleich, aber nicht größer ist.[110] Ziel ist es, den Eintrag zu finden, dessen Key-Wert am dichtesten am gesuchten Key liegt, aber nicht größer ist.

Es wird immer die Parent ID verglichen. Wenn diese kleiner oder gleich ist, dann wird der Name verglichen, bis der relevante Node mit dem relevanten Eintrag gefunden wird.

Im ersten Analyseschritt muss die Struktur des *Catalog Files* anhand des B-Tree Header Nodes untersucht werden.

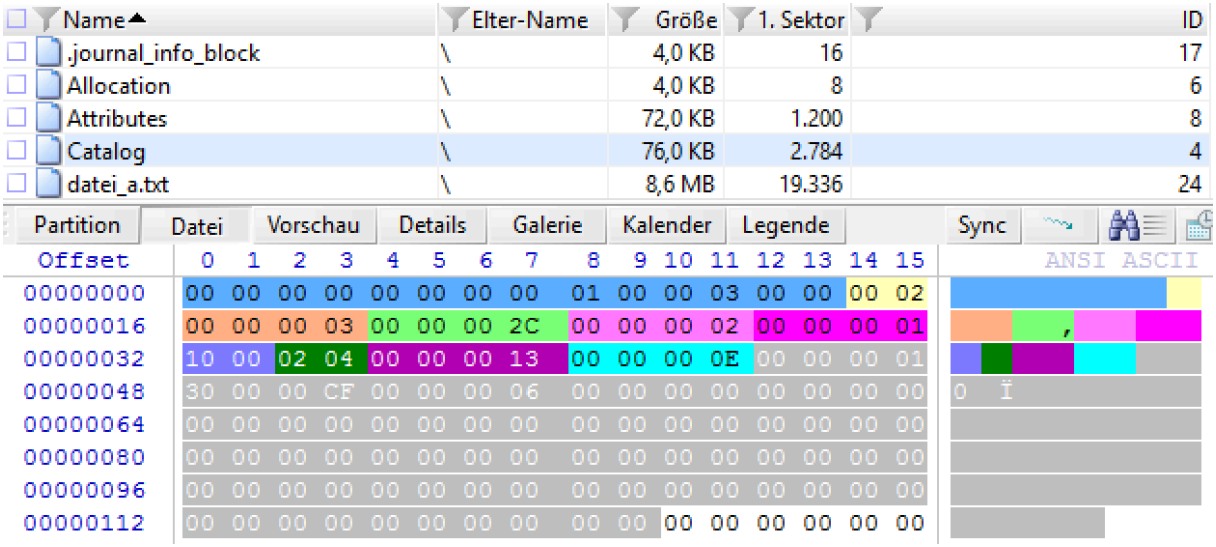


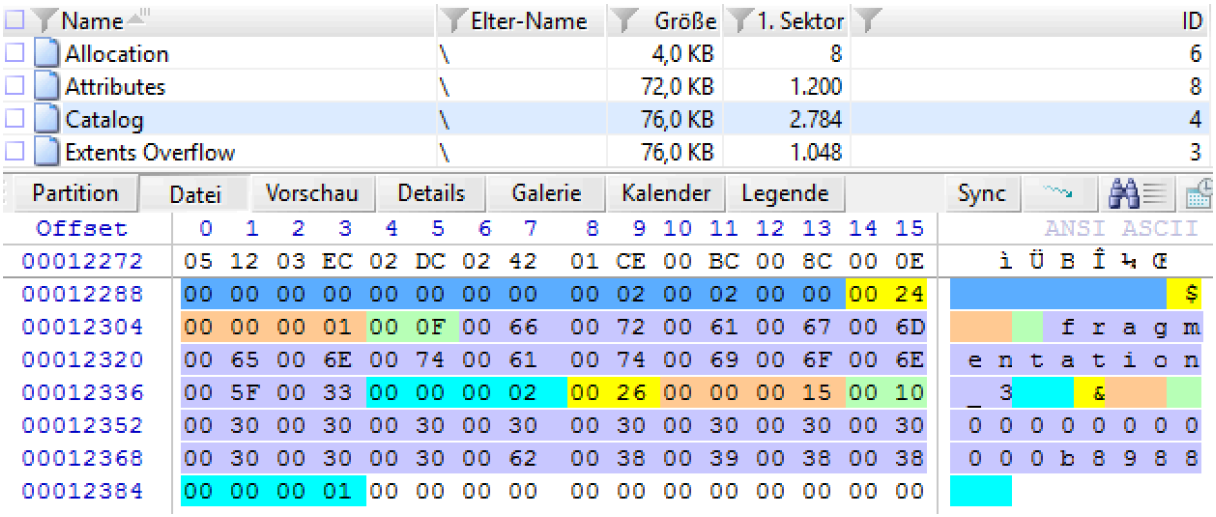
Abbildung 36: Header Node des Catalog Files

In der Abbildung 36 wird der Header Node mit dem Header Node Eintrag des *Catalog Files* dargestellt. Die blaue Markierung zeigt die 14 Bytes des Header Node Descriptors. Bei allen Werten handelt es sich um Big-Endian Integer-Werte. Das Feld für den Node Typ enthält den Wert 1, der bestätigt, dass es sich um den Header Node des *Catalog Files* handelt. Als Level des Nodes ist der Wert 0 für Header Node gespeichert. Die Anzahl der Einträge enthält den Wert 3.

Wichtig für die weitere Analyse ist vor allem der am Node-Offset 14 folgende Header Node Eintrag, der Auskunft über die Node Struktur im *Catalog File* gibt. Die ersten zwei Bytes im Eintrag (gelbe Markierung) geben die B-Tree-Tiefe an, die hier 2 beträgt. Als nächstes wird die ID des Root Nodes mit dem Wert 3 gespeichert, der hier orange markiert ist. Die Anzahl der Leaf Einträge im gesamten *Catalog File* werden durch die folgenden vier Bytes, die grün markiert sind, dargestellt. Demnach beinhaltet der B-Tree 42 Leaf Node Einträge. Der erste Leaf Node hat die ID 2 (rosa markierte vier Bytes) und der letzte Leaf Node besitzt die ID 1 (pink markierte vier Bytes). Die folgenden zwei violett markierten Bytes geben die Größe der Nodes im gesamten B-Tree an, die hier 4096 Bytes beträgt. Die zwei grün markierten Bytes stellen die maximale Key-Länge von 516 Bytes in den Index Node Einträgen und Leaf Node Einträgen dar. Die Anzahl der Nodes (vier Bytes in Bordeaux) beträgt 19 und die Anzahl der nicht belegten Nodes (vier Bytes in Türkis) beträgt 15.

[110] Singh 2007, S. 1490

Aus diesen Angaben lassen sich wichtige und für die Suche nach dem Datei-Eintrag notwendige Informationen erheben. Sie geben an, dass sich die Suche auf zwei Leaf Nodes beschränkt, die zusammen 42 Einträge beinhalten. Jede Suche beginnt im Root Node, dessen Position sich aus der ID und der Node Größe berechnen lässt. Hier ist sie ein Produkt von 3 und 4096. Der Root Node lässt sich am Byte-Offset 12288 des *Catalog Files* aufsuchen.



Name	Elter-Name	Größe	1. Sektor	ID
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4
Extents Overflow	\	76,0 KB	1.048	3

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

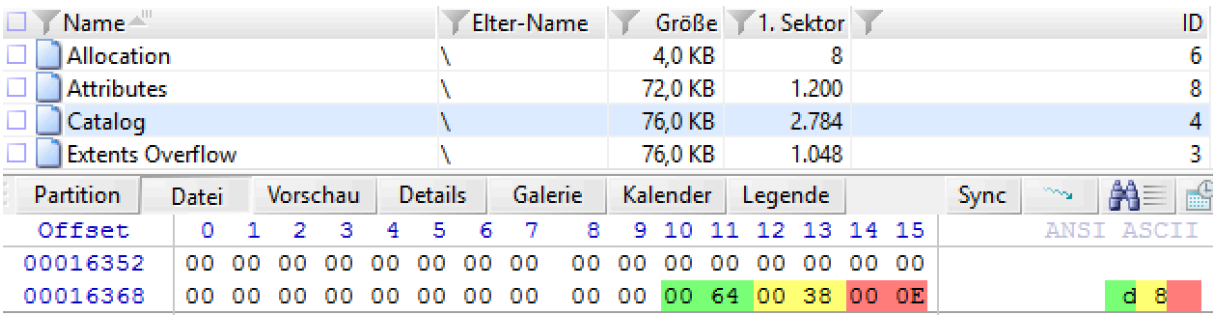
Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00012272	05	12	03	EC	02	DC	02	42	01	CE	00	BC	00	8C	00	0E	ì Ü B Î ↳ Œ
00012288	00	00	00	00	00	00	00	00	00	02	00	02	00	00	00	24	$
00012304	00	00	00	01	00	0F	00	66	00	72	00	61	00	67	00	6D	f r a g m
00012320	00	65	00	6E	00	74	00	61	00	74	00	69	00	6F	00	6E	e n t a t i o n
00012336	00	5F	00	33	00	00	00	02	00	26	00	00	00	15	00	10	_ 3 &
00012352	00	30	00	30	00	30	00	30	00	30	00	30	00	30	00	30	0 0 0 0 0 0 0 0
00012368	00	30	00	30	00	30	00	62	00	38	00	39	00	38	00	38	0 0 0 b 8 9 8 8
00012384	00	00	00	01	00	00	00	00	00	00	00	00	00	00	00	00	

Abbildung 37: Root Node mit zwei Einträgen

Abbildung 37 zeigt den Root Node des Catalog Files. Der Node Descriptor des Root Nodes ist blau markiert und nimmt die ersten 14 Bytes ein. Nach der zuvor vorgestellten Descriptor Struktur wird am Node-Offset 8 mit dem Wert 0 angegeben, dass es sich um ein Index Node handelt. Node-Offset 9 gibt mit dem Wert 2 an, dass es sich um ein Index Node handelt, der auf Leaf Nodes verweist, d. h. dass er der einzige Index Node im B-Tree ist und die B-Tree-Tiefe somit 2 betragen muss. Die Anzahl der Einträge im Root Node steht im Node-Offset 10 mit dem Wert 2. Diese zwei Einträge müssen näher untersucht werden. Aus den Informationen des Root Node Descriptors und des Header Eintrags ist bekannt, dass die zwei Index Node Einträge auf jeweils einen Leaf Node verweisen.

Die Offsets zu den zwei Einträgen im Root Node lassen sich aus den 2-Byte Werten am Ende des Nodes ermitteln (siehe folgender Abbildung 38):

- Eintrag Nr. 1: Node-Offset 16382-16383 (rot) = 14
- Eintrag Nr. 2: Node-Offset 16380-16381 (gelb) = 56
- Freier Speicher: Node-Offset 16378-16379 (grün) = 100



Name	Elter-Name	Größe	1. Sektor	ID
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4
Extents Overflow	\	76,0 KB	1.048	3

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00016352	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00016368	00	00	00	00	00	00	00	00	00	00	00	64	00	38	00	0E	d 8

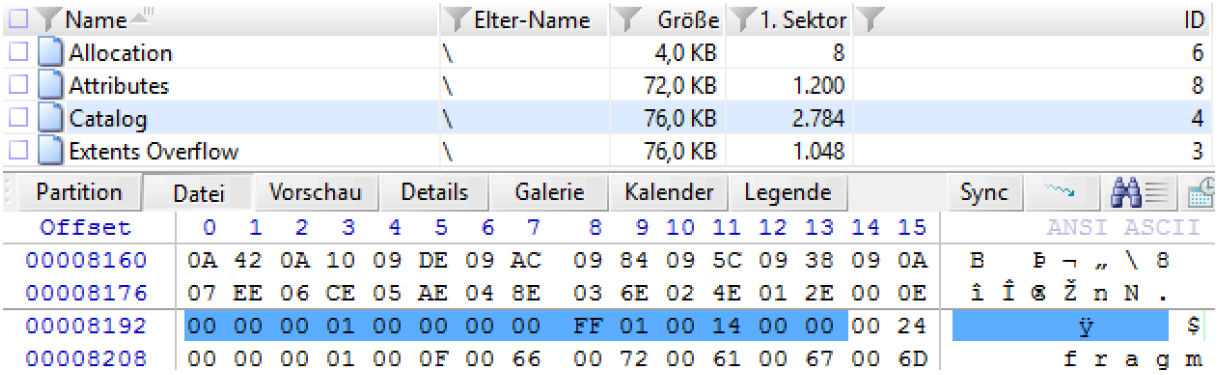
Abbildung 38: Offsets der Einträge des Root/Index Nodes

Der richtige Eintrag kann nun durch Vergleich der Parent ID zur Datei „datei_a.txt“ mit dem Parent ID-Wert der Index Node Einträge gefunden werden.

Wie zuvor festgestellt, beginnt der erste Eintrag am Node-Offset 14. Die ersten zwei Bytes (gelb markiert in Abbildung 37) geben die Key-Länge an. Diese beträgt für diesen Eintrag 36. Im Anschluss folgt die Parent ID mit vier Bytes und beträgt 1. Diese ID ist kleiner und sehr nah an der gesuchte Parent ID 2 dran. Als nächstes wird die Länge des Dateinamens mit zwei Bytes angegeben, die hier 36 beträgt. Da es sich um eine Unicode255-Struktur handelt, die 2-Byte-Zeichen verwendet, nimmt der folgende Name 72 Bytes ein. Diese sind violett gekennzeichnet. Der Name lautet demnach „fragmentation_3“. „fragmentation_3“ ist der Root Ordner, der als Parent ID immer den Wert 1 hat und als CNID die 2 besitzt, was die Parent ID zum gesuchten Catalog File Eintrag zur Datei „datei_a.txt“ mit der ID 24 darstellt. Nach dem Namen wird in den letzten vier Bytes des Eintrags die nächste Node ID gespeichert. Diese beträgt hier 2.

Zur Überprüfung, ob der zweite Index Eintrag im Root Node näher an der gesuchten CNID liegt, muss die Parent ID mit der gesuchten Parent ID verglichen werden. Der zweite Index Node Eintrag beginnt am Node-Offset 56. Nach dem Key-Längen-Feld wird die Parent ID in der Abbildung orange markiert. Diese hat den Wert 21. 21 ist nicht kleiner oder gleich 2. Somit kann der Node, auf den dieser Eintrag verweist, nicht relevant sein.

Nun kann die Position des Leaf Nodes mit der Node ID 2 durch das Produkt von 2 und 4096 bestimmt werden und beginnt somit am Offset 8192 des *Catalog Files*.



Name	Elter-Name	Größe	1. Sektor	ID
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4
Extents Overflow	\	76,0 KB	1.048	3

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

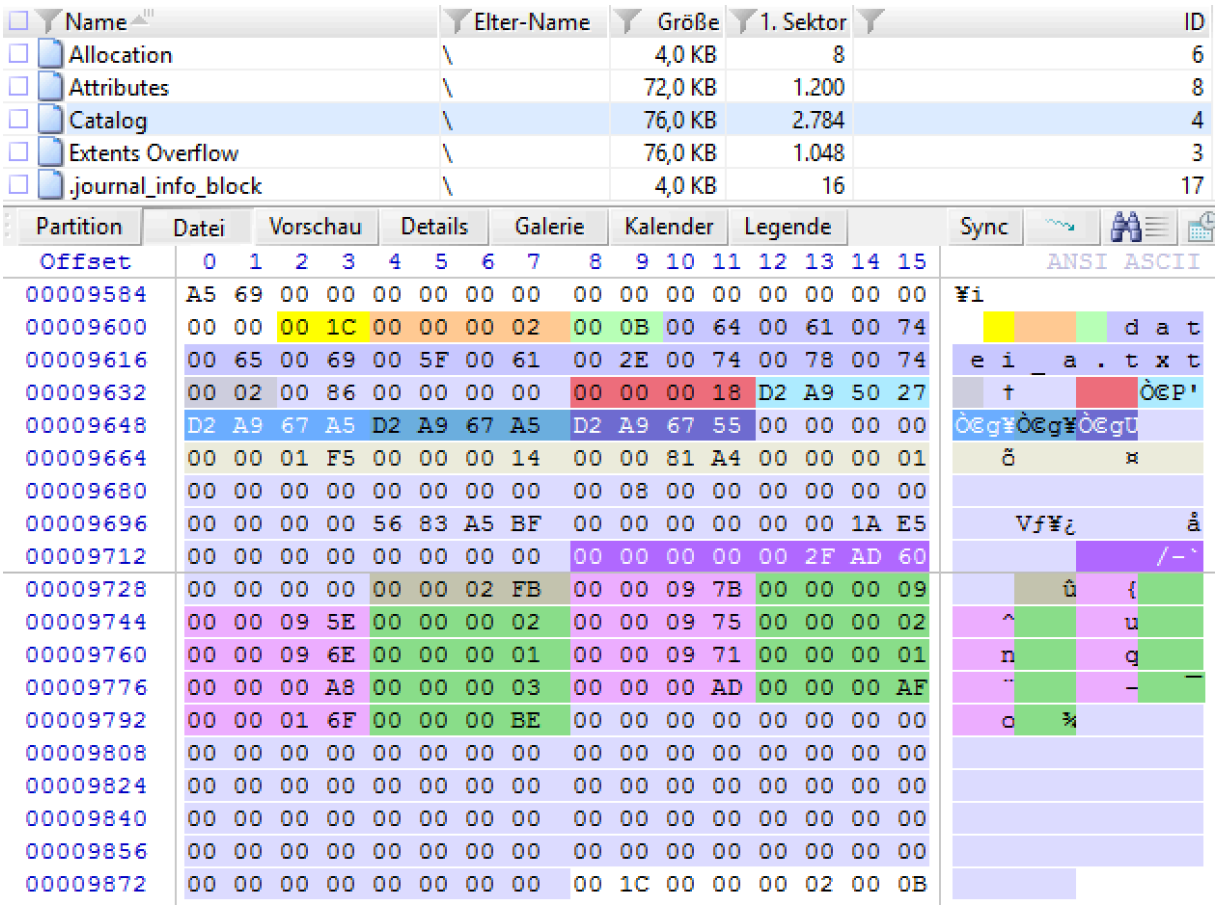
Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00008160	0A	42	0A	10	09	DE	09	AC	09	84	09	5C	09	38	09	0A	B Þ ¬ „ \ 8
00008176	07	EE	06	CE	05	AE	04	8E	03	6E	02	4E	01	2E	00	0E	î Î ® Ž n N .
00008192	00	00	00	01	00	00	00	00	FF	01	00	14	00	00	00	24	ÿ $
00008208	00	00	00	01	00	0F	00	66	00	72	00	61	00	67	00	6D	f r a g m

Abbildung 39: Node Descriptor des Leaf Nodes ID 2

Die ersten vier Bytes des Node Descriptors geben die ID des nächsten Nodes gleichen Typs an, die hier 1 beträgt. Die folgenden vier Bytes für die ID des vorherigen Nodes gleichen Typs geben 0 an, was bestätigt, dass nur noch ein zweiter Leaf Node im B-Tree existiert. Als Node Typ steht der 1-Byte Wert 0xFF, der vorzeichenbehaftet ist und mit -1 für Leaf Node steht. Auch das Level 1 im folgenden Byte gibt an, dass es sich um einen Leaf Node handelt. Die anschließenden zwei Bytes geben Auskunft über die Anzahl der Einträge im Node, die hier 20 beträgt.

Es können nun die Offsets der 20 Leaf Node Einträge am Ende des Nodes ermittelt werden. Um den relevanten Eintrag zu finden, müssen die Key-Werte der Einträge mit dem Parent Key und dem Key des gesuchten Eintrags verglichen werden. Dieser Vergleich führt zum Leaf Node Eintrag Nr. 9 am Offset 9602 des *Catalog Files*.

In der folgenden Abbildung wurden die für die Analyse interessanten Felder des Leaf Node Eintrags Nr. 9 mit Key-Struktur farblich markiert:



Name	Elter-Name	Größe	1. Sektor	ID
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4
Extents Overflow	\	76,0 KB	1.048	3
.journal_info_block	\	4,0 KB	16	17

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00009584	A5	69	00	00	00	00	00	00	00	00	00	00	00	00	00	00	¥i
00009600	00	00	00	1C	00	00	00	02	00	0B	00	64	00	61	00	74	d a t
00009616	00	65	00	69	00	5F	00	61	00	2E	00	74	00	78	00	74	e i _ a . t x t
00009632	00	02	00	86	00	00	00	00	00	00	00	18	D2	A9	50	27	† ÒP'
00009648	D2	A9	67	A5	D2	A9	67	A5	D2	A9	67	55	00	00	00	00	Ò©g¥Ò©g¥Ò©gU
00009664	00	00	01	F5	00	00	00	14	00	00	81	A4	00	00	00	01	õ ¤
00009680	00	00	00	00	00	00	00	00	00	08	00	00	00	00	00	00	
00009696	00	00	00	00	56	83	A5	BF	00	00	00	00	00	00	1A	E5	Vƒ¥¿ å
00009712	00	00	00	00	00	00	00	00	00	00	00	00	00	2F	AD	60	/-`
00009728	00	00	00	00	00	00	02	FB	00	00	09	7B	00	00	00	09	û {
00009744	00	00	09	5E	00	00	00	02	00	00	09	75	00	00	00	02	^ u
00009760	00	00	09	6E	00	00	00	01	00	00	09	71	00	00	00	01	n q
00009776	00	00	00	A8	00	00	00	03	00	00	00	AD	00	00	00	AF	¨ - ¯
00009792	00	00	01	6F	00	00	00	BE	00	00	00	00	00	00	00	00	o ¾
00009808	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00009824	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00009840	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00009856	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00009872	00	00	00	00	00	00	00	00	00	1C	00	00	00	02	00	0B	

Abbildung 40: Catalog Key und Dateieintrag von "datei_a.txt"

Am Offset 9602 des *Catalog Files* beginnt die Key-Struktur des Catalog File Eintrags, deren Länge variabel ist. Anschließend folgt der Catalog Dateieintrag mit einer festen Länge von 248 Bytes.

Nach den ersten zwei Bytes in der Key-Struktur (gelbe Markierung), die die Länge mit dem Wert 28 Bytes der Key-Struktur definieren, folgen vier Bytes mit der Parent ID 2 (orangene Markierung). Diese ist identisch mit der gesuchten Parent ID. Die nächsten zwei Bytes geben die Länge des folgenden Dateinamens an. Diese beträgt 11. Da es sich um 2-Byte Unicode255-Zeichen handelt, benötigt der Dateiname 22 Bytes. Der Name lautet „datei_a.txt“. Dieser entspricht dem Namen der gesuchten Datei.

Im Anschluss beginnt nun am Offset 9632 des *Catalog Files* der Catalog Dateieintrag zur Key-Struktur. Der Typ des Eintrags wird in den ersten zwei Bytes definiert (graue Markierung). Der Wert beträgt 2, wonach es sich um einen Catalog Dateieintrag handelt. Die nächsten vier Bytes sind reserviert. Im Anschluss wird die CNID der Datei, zu der der Catalog File Eintrag gehört, mit einem 4-Byte-Wert gespeichert. Diese ist in der Abbildung rot markiert und beträgt 24. Dies entspricht der bekannten CNID, nach der gesucht wurde. Der Catalog Dateieintrag verweist auf „datei_a.txt“.

Es folgen vier 4-Byte-Werte, bei denen es sich um Zeitstempel handelt, die im HFS Plus Zeitformat abgelegt sind (siehe Tabelle 21). Im Beispiel sind es der Reihe nach folgende Zeitstempel:

- Erstellungszeitpunkt (türkis): 30.12.2015, 09:11:03 Uhr
- Letzter Schreibzugriff (hellblau): 30.12.2015, 10:51:17 Uhr
- Letzter Schreibzugriff des Catalog-Eintrags (blau): 30.12.2015, 10:51:17 Uhr
- Letzter Zugriff (dunkelblau): 30.12.2015, 10:49:57 Uhr

Nach vier Bytes, die für Backup-Zeitstempel reserviert sind und hier den Wert 0 beinhalten, folgen in der Abbildung 40 hellgrau markierte 16 Bytes mit den Zugriffsrechten der Datei. Die ersten vier Bytes davon beinhalten die Besitzer ID, die hier 501 beträgt. Die darauffolgenden vier Bytes zeigen die Gruppen ID, die hier 20 lautet.

Ein weiteres interessantes Feld stellt der Eintrag mit der logischen Größe der Datei dar, der im Offset 88 ausgehend vom Anfang des Catalog Dateieintrags gespeichert ist und acht Bytes belegt (violette Markierung). Demnach beträgt die logische Größe für die Datei „datei_a.txt“ 3.124.576 Bytes. Im Offset 100 des Eintrags steht in vier Bytes die Gesamtzahl der Allocation Blocks, die von der Datei „datei_a.txt“ allokiert sind (dunkelgraue Markierung). Der Wert beträgt 763. Es folgen ab Offset 104 acht Extent Einträge für Data Fork, die die Position und die Anzahl der von der Datei allokierten Allocation Blocks angeben. Diese sind jeweils Paare mit je vier Bytes für den Start Allocation Block (rosa Markierung) und vier Bytes für die Anzahl der Allocation Blocks (grüne Markierung) des jeweiligen Extents. Daraus ergeben sich folgende Angaben zu den Extents:

Extent Nr.	Start Allocation Block	Anzahl der Allocation Blocks
1	2427	9
2	2398	2
3	2421	2
4	2414	1
5	2417	1
6	168	3
7	173	175
8	367	190

Tabelle 26: Extents Einträge im Catalog File

Wird die Anzahl der Allocation Blocks aller acht Extents addiert, ergibt sich eine Summe von 383 Allocation Blocks. Da das Feld mit der Anzahl aller Allocation Blocks, die die Datei allokiert, 763 beträgt, müssen weitere Extent Einträge zur Datei „datei_a.txt“ im *Extents Overflow File* vorhanden sein. Es ist erkennbar, dass die Datei „datei_a.txt“ stark fragmentiert ist. Auf Extent Einträge im *Extents Overflow File* wird im nächsten Abschnitt detailliert eingegangen.

Folgende Abbildung gibt einen Überblick über die Node Struktur aus dem Beispiel und zeigt den gesuchten Catalog Node Dateieintrag:

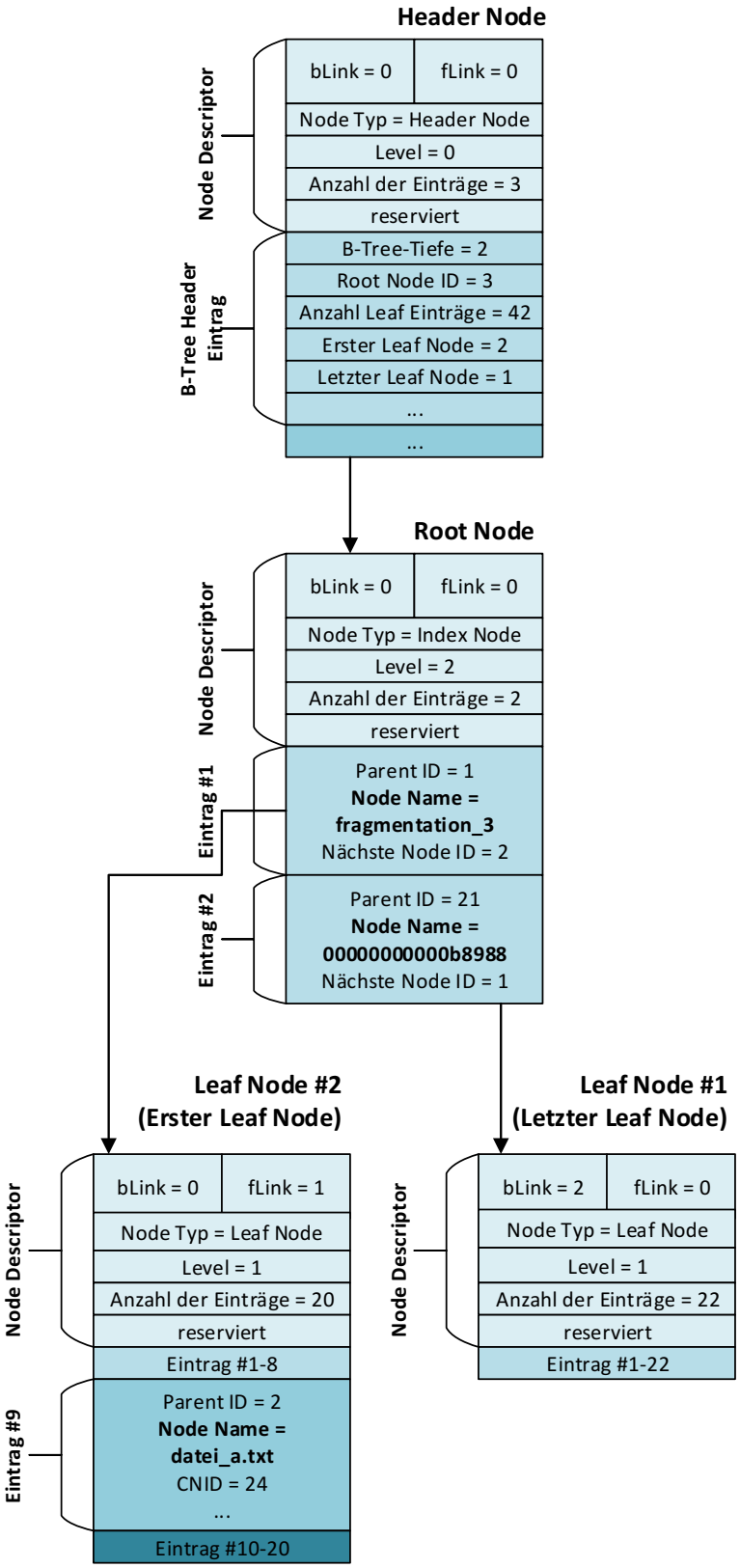


Abbildung 41: B-Tree Node Struktur des Catalog Files

5.3.3 Extents Overflow File

Wenn Catalog File Einträge mehr als acht Extent Einträge jeweils für Data Fork und Resource Fork benötigen, dann werden zusätzliche Extent Einträge im Extents Overflow File gespeichert. Diese Einträge enthalten wie die Extent Einträge im Catalog File die Position und Größe der entsprechenden Extents.[111]

Das Extents Overflow File enthält Informationen über die Allocation Blocks, die zu einem Data Fork gehören, indem es Extent Einträge für die jeweiligen Extents speichert. Extents sind zusammenhängende Allocation Blocks, die zu einer Datei gehören. Wenn eine zusammenhängende Allokation nicht möglich ist, wird der logische Inhalt einer Datei in weitere zusammenhängende Extents geteilt. Je mehr Extents benötigt werden, umso höher ist die Fragmentierung.[112]

Für dieses Beispielimage wurde durch verschiedene Dateimanipulationen „künstlich“ eine Fragmentierung erzeugt. Die einzelnen Schritte und Informationen können dem Anhang A.3 entnommen werden.

Das Extents Overflow File hat wie das Catalog File eine B-Tree-Struktur, die aber einfacher aufgebaut ist. So hat das Extents Overflow File nur einen Eintragstyp und eine feste Key-Größe.

5.3.3.1 Struktur der Extents Overflow File Einträge

Die Extents Overflow File Einträge, die im Extents Overflow File in Leaf Nodes gespeichert werden, beinhalten acht Extent Einträge (je acht Data Fork und je acht Resource Fork). Werden mehr als acht Extents je Fork Typ für eine Datei benötigt, werden auch mehr Extents Overflow File Einträge benötigt.

Die Key-Struktur von Extents Overflow File Einträgen ist in der *Apple Technical Note TN1150* und in *hfs_format.h* als *HFSPlusExtentKey* wie folgt definiert:[113] [114]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Key-Größe
2	1	Fork Type 0x00 – Data Fork 0xFF – Resource Fork
3	1	leer
4	4	CNID
8	4	Nummer des ersten Start Blocks

Tabelle 27: Key-Struktur eines Extents Overflow File Eintrags

[111] Apple: *Technical Note TN1150*, S. 25–26
[112] Singh 2007, S. 1519
[113] Apple: *Technical Note TN1150*, S. 25–26
[114] Apple: *hfs_format.h* 2012, S. 2

Anschließend folgen die 8-Byte Extent Einträge in folgender Struktur, die als *HFSPlusExtentDescriptor* definiert sind:[115]

Größe (Byte)	Beschreibung
4	Start Block
4	Block Count

Tabelle 28: Extent-Descriptor-Struktur im Extents Overflow File

Im Gegensatz zu den Catalog Keys im Catalog File hat das Extents Overflow File Key-Strukturen mit einer festen Key-Größe von zehn Bytes, die im Offset 0 des Key-Eintrags als 16 Bit Big-Endian Integer-Wert gespeichert werden.[116] Es folgt ein 8 Bit Big-Endian Integer-Wert mit dem Fork Typ, bei dem die Werte 0x00 für Data Fork und 0xFF für Resource Fork möglich sind. Das darauf folgende Byte ist reserviert und grundsätzlich leer. Im Offset 4 wird die CNID der Datei, zu der die Extent Einträge gehören, als 32 Bit Big-Endian Integer-Wert gespeichert. Anschließend folgt die Nummer des ersten Start Blocks, auf den der erste Extent Eintrag des Extents Overflow Files verweist, mit vier Bytes. Die Nummer setzt sich aus der Anzahl der Allocation Blocks zusammen, die durch die Extent Einträge im Catalog File beschrieben sind. Anschließend folgen die Extent Descriptor Paare des Extents Overflow File Eintrags jeweils mit einer Größe von acht Bytes. Davon sind die ersten vier Bytes ein 32 Bit Big-Endian Integer-Wert mit dem jeweiligen Start Allocation Block und ein folgender 32 Bit Big-Endian Integer-Wert mit der Allocation Block Anzahl.[117]

5.3.3.2 Suche im Extents Overflow File

Einträge im Extents Overflow File werden durch Vergleiche der Keys gesucht. Dazu werden die Key-Felder in folgender Reihenfolge verglichen: CNID, Fork Typ und Start Block.[118]

Da es sich um einen B-Tree handelt, beginnt die Suche mit der Analyse des Header Nodes, um einen Überblick über die Node Struktur zu gewinnen und dann die Position des Root Nodes zu ermitteln, um anschließend durch Vergleiche der Keys die gesuchten Einträge zu finden.

5.3.3.3 Analyse-Szenario

Als Beispiel dient wieder die Datei „datei_a.txt“ aus dem Übungsimage „image_3_2“. Um die Position aller Extents von „datei_a.txt“ bestimmen zu können, müssen die Metadaten dieser Datei analysiert werden. Im ersten Schritt wird der Catalog File Eintrag der Datei „datei_a.txt“ anhand ihrer CNID 24 und Parent ID 2 im Catalog File durch Catalog Key und Dateinamen-Vergleich aufgesucht. Dies wurde bereits im vorherigen Kapitel durchgeführt und im Ergebnis am Offset 9602 des Catalog Files der zugehörige Catalog Dateieintrag zu „datei_a.txt“ gefunden. Er endet an Offset 9879 (siehe Abbildung 40 unter 5.3.2.6).

Nach der Catalog Key-Struktur beginnt ab Datei-Offset 9632 der Catalog Dateieintrag. Um Informationen über die Extents zu ermitteln, ist der Catalog File Eintrag ab Offset 9732 (Offset 100 ausgehend vom Catalog Dateieintrag, siehe Tabelle 21 unter 5.3.2.2) von Bedeutung und

[115] Apple: *hfs_format.h* 2012, S. 2
[116] Apple: *Technical Note TN1150*, S. 25–26
[117] Apple: *Technical Note TN1150*, S. 26–27
[118] Apple: *Technical Note TN1150*, S. 26

gibt die Gesamtzahl der Allocation Blocks für Data Fork an, die die Datei belegt. Wie zuvor festgestellt, belegt „datei_a.txt“ 763 Allocation Blocks.

Im oben gezeigten Beispiel belegen die Extent Descriptor Paare (vier Bytes für den jeweiligen Start Allocation Block des Extents und vier Bytes für die dazugehörige Anzahl der zusammenhängenden Allocation Blocks) Offsets 9736 bis 9799. Der Bereich bis zum Ende des Catalog File Eintrags ist für Resource Fork Extents reserviert, welcher hier leer ist, da „datei_a.txt“ keine Resource Fork Daten besitzt.

Die Summe aus der Anzahl der Allocation Blocks ergibt 383 Allocation Blocks. Dies ist ein Anhaltspunkt, dass weitere Extents vorhanden sind, die nicht im Catalog File beschrieben werden. Die Anzahl der Allocation Blocks, die nicht im Catalog File Eintrag stehen, lässt sich aus der Differenz von 763 als Gesamtzahl der Allocation Blocks und 383 als Anzahl der Allocation Blocks, die im Catalog File Eintrag durch die Extent Deskriptoren beschrieben werden, ermitteln. Somit werden 380 Allocation Blocks durch zusätzliche Extent Einträge im Extents Overflow File dargestellt und können dort analysiert werden.

Um die Position der zusätzlichen Extent Einträge ermitteln zu können, muss die B-Tree-Struktur des Extents Overflow Files analysiert werden. Der Header Node ab Offset 0 des Extents Overflow Files gibt Aufschluss über die Struktur der Nodes in diesem B-Tree. Der Node Descriptor belegt die ersten 14 Bytes und bestätigt am Offset 8 mit einem 8 Bit Big-Endian Integer-Wert gleich 1 und mit dem Wert 0 am Offset 9, dass es sich um einen Header Node handelt. In folgender Abbildung wurde der Node Descriptor des Header Nodes im Extents Overflow File blau markiert.

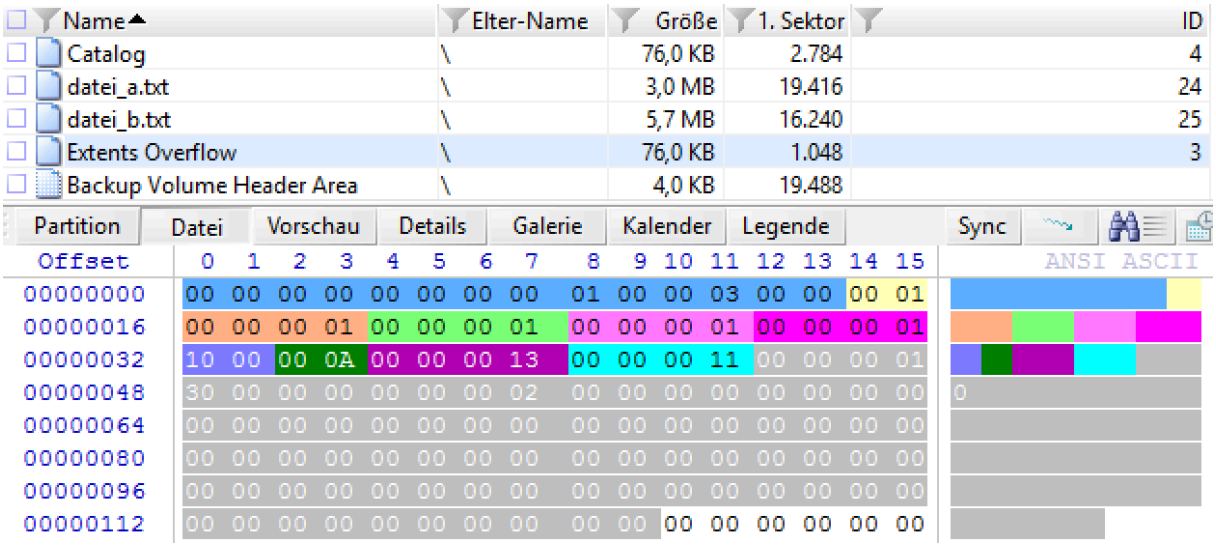


Abbildung 42: Header Node Eintrag des Extents Overflow Files

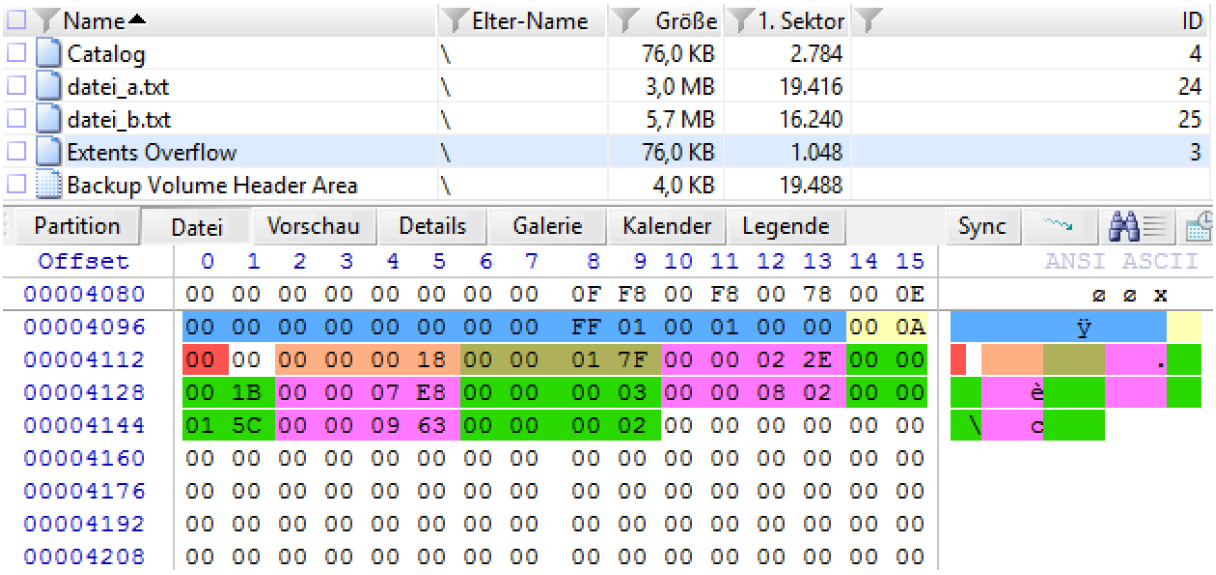
Um den relevanten Leaf Node zu ermitteln, muss der Header Eintrag, der im Header Node nach dem Node Descriptor am Offset 14 des Extents Overflow Files folgt, untersucht werden.

Die ersten zwei Bytes des Header Eintrags am Offset 14-15 des Header Nodes geben die Tiefe des B-Trees an, welche in diesem Fall 1 ist (gelbe Markierung). Die darauffolgenden vier Bytes beinhalten die Root Node ID 1 (orangene Markierung). Anschließend folgen vier Bytes mit dem Wert für die Anzahl der Leaf Nodes im gesamten Extents Overflow File, der hier 1 beträgt. Die

ID des ersten und letzten Leaf Nodes im Extents Overflow File beträgt ebenfalls 1 (vier Byte rosa und vier Byte pinkfarbene Markierung). Die nächsten zwei Bytes geben die Größe der Nodes im B-Tree an (violette Markierung); hier: 4096 Bytes. Die maximale Key-Länge wird durch die anschließenden zwei Bytes beschrieben. Hier steht der Wert 10 Bytes, was die Standardgröße für Keys im Extents Overflow File darstellt.

Die Suche nach dem relevanten Leaf Node mit dem entsprechenden Leaf Node-Eintrag ist in diesem Fall einfach, da es nur einen Leaf Node mit der ID 1 gibt, der hier gleichzeitig auch den Root Node darstellt. Aus dem Header Node Eintrag geht hervor, dass die Größe jedes Nodes in diesem B-Tree 4096 Bytes und dass die Root Node-ID 1 beträgt. Die Position des Root Nodes und gleichzeitig einzigen Leaf Nodes im B-Tree lässt sich so einfach durch das Produkt von 1 und 4096 ermitteln.

Die folgende Abbildung zeigt den Leaf Node beginnend am Offset 4096 im Extents Overflow File.



Name	Elter-Name	Größe	1. Sektor	ID
Catalog	\	76,0 KB	2.784	4
datei_a.txt	\	3,0 MB	19.416	24
datei_b.txt	\	5,7 MB	16.240	25
Extents Overflow	\	76,0 KB	1.048	3
Backup Volume Header Area	\	4,0 KB	19.488	

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00004080	00	00	00	00	00	00	00	00	0F	F8	00	F8	00	78	00	0E	ø ø x
00004096	00	00	00	00	00	00	00	00	FF	01	00	01	00	00	00	0A	ÿ
00004112	00	00	00	00	00	18	00	00	01	7F	00	00	02	2E	00	00	.
00004128	00	1B	00	00	07	E8	00	00	00	03	00	00	08	02	00	00	è
00004144	01	5C	00	00	09	63	00	00	00	02	00	00	00	00	00	00	\ c
00004160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00004176	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00004192	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00004208	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Abbildung 43: Leaf Node mit Node Descriptor und Leaf Node Eintrag

Die ersten 14 Bytes des Nodes werden durch den Node Descriptor belegt (blaue Markierung). Aus den Informationen des Descriptors geht hervor, dass es der einzige Node des gleichen Typs im B-Tree ist und es sich um ein Leaf Node handelt (siehe Offset 8 des Nodes; Hier beträgt der Wert -1 (0xFF), was den Node als Leaf Node kennzeichnet.). Im Node-Offset 11 lässt sich die Anzahl der Node Einträge erheben, die hier 1 beträgt.

Aus dem Header Node Eintrag ist ebenfalls bekannt, dass nur ein Leaf Node Eintrag in diesem B-Tree existiert. Dieser beginnt am Offset 14 des Leaf Nodes. Die Struktur kann aus der zuvor dargestellten Tabelle 17 unter 5.3.1 entnommen werden. Die Key-Struktur des Extents Overflow File Eintrags beginnt direkt nach dem Node Descriptor am Offset 14 des Nodes. Die ersten zwei gelb markierten Bytes geben die Key-Größe an, die 10 Bytes beträgt und die der zuvor besprochenen Standardgröße bei Extents Overflow Key-Strukturen entspricht. Das rot markierte Feld gibt den Fork Typ an, der mit dem Wert 0 für Data Fork steht. Anschließend wird mit vier Bytes die CNID der Datei gespeichert (orangene Markierung), die die zusätzlichen Extents belegt. Der Wert beträgt im Beispiel 24 und bestätigt damit die Zugehörigkeit zur Datei

„datei_a.txt“. Im Feld für den Start Allocation Block steht der 32 Bit Integer-Wert 383. Dies stellt die bereits angesprochene Summe der Allocation Blocks dar, die durch die Extent Descriptors im Catalog File Eintrag von „datei_a.txt“ belegt werden. Der erste zur Datei „datei_a.txt“ zugehörige Extent-Eintrag im Extents Overflow File verweist auf diese Blockzahl im Catalog File.[119] Die darauffolgenden vier Extent Descriptor Paare geben den jeweiligen Start Allocation Block (rosa Markierung) und die Anzahl der Allocation Blocks (grüne Markierung) des Extents wieder.

Für eine bessere Übersicht werden alle Extents von „datei_a.txt“ in der folgenden Tabelle aufgeführt:

Extent Nr.	Start Allocation Block	Anzahl Blocks	Herkunft Extent Eintrag
1	2.427	9	Catalog File
2	2.398	2	Catalog File
3	2.4.21	2	Catalog File
4	2.414	1	Catalog File
5	2.417	1	Catalog File
6	168	3	Catalog File
7	173	175	Catalog File
8	367	190	Catalog File
9	110.592	558	Extents Overflow File
10	2.024	3	Extents Overflow File
11	2.050	348	Extents Overflow File
12	2.403	2	Extents Overflow File

Tabelle 29: Sämtliche Extents zur "datei_a.txt"

Die Summe aller Allocation Blocks ergibt 763 und entspricht der im Catalog Dateieintrag angegebenen Anzahl für die Datei „datei_a.txt“.

[119] Apple: *Technical Note TN1150*, S. 26

Die nachfolgende Abbildung gibt einen Überblick der Node Struktur des Extents Overflow Files aus dem Beispielimage „image_3_2“ mit den gesuchten Extent Einträgen zur Datei „datei_a.txt“:

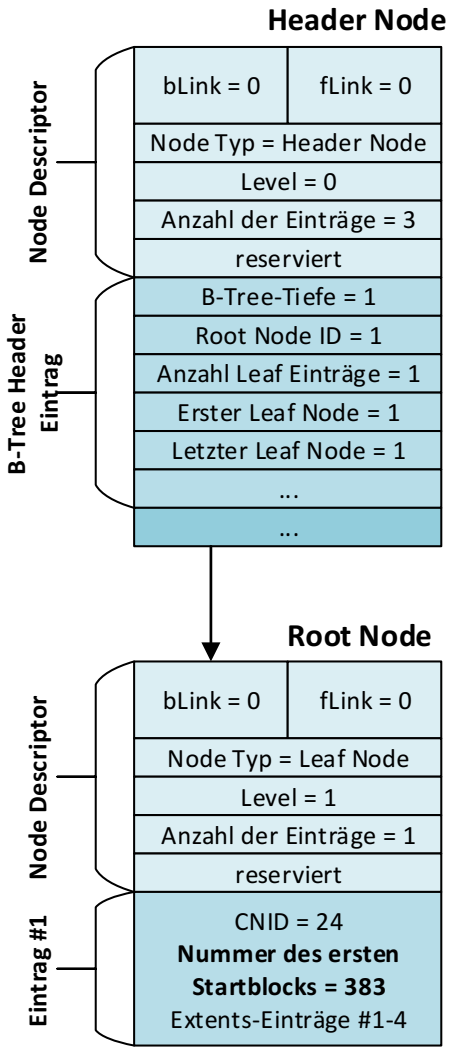


Abbildung 44: B-Tree Node Struktur des Extents Overflow Files

5.3.4 Attributes File

Mit Mac OS X 10.4 wurden von Apple die *Extended Attributs* eingeführt, um dem Dateisystem HFS Plus mehr Metadatenfunktionalität hinzuzufügen. Dateien und Verzeichnisse können seit dem zusätzliche Metadaten (*Extended Attributes*) besitzen. Die *Extended Attributes* werden im *Attributes File* gespeichert, welches ebenfalls als B-Tree aufgebaut ist.[120]

Als *Extended Attributes* kommen Metadaten, wie zum Beispiel Finder Flags, Finder Kommentare, Download-Informationen, Sicherheitsinformationen und Metadaten von Drittprogrammen usw. in Betracht.

Es ist möglich, dass auf einem HFS Plus Volume kein *Attributes File* vorhanden ist. Dies lässt sich aus dem HFS Plus Volume Header ermitteln. Hat der erste Extent für das *Attributes File* Allocation Blocks mit dem Wert 0, dann existiert das *Attributes File* nicht und somit auch keine *Extended Attributes* in diesem Volume.[121]

Die Key-Struktur dieses B-Trees hat eine variable Größe und beginnt in den ersten zwei Bytes, die die Key-Größe angeben. Die zwei darauffolgenden Bytes sind ohne Inhalt. Am Byte-Offset 4 der Key-Struktur befindet sich ein 4-Byte-Wert, der die CNID der zugehörigen Datei repräsentiert. Am Offset 8 ist ein vier Bytes großer Wert, der den Start Block angibt. Dieser Wert wird für Inline Attribute nicht genutzt, da diese vollständig im *Attributes File* gespeichert werden. Die folgenden zwei Bytes geben die Größe des Attribut-Namens an. Dieser Wert ist variabel und beginnt ab Offset 14 mit 2-Byte-Unicode255-Zeichen. Die nachfolgenden Tabellen geben eine Übersicht über die Key-Struktur und die jeweiligen Attribut-Einträge.[122] [123]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Key Größe
2	2	leer
4	4	CNID
8	4	Start Block
12	2	Größe des Attribut-Namens
14	variabel	Name des Attributes

Tabelle 30: Key-Struktur im Attributes File

Die Struktur der auf die Keys folgenden Einträge ist abhängig vom Eintrags-Typ der Attribute *Inline Attribute*, *Fork Data Attribute* und *Extension Attribute.* Die ersten vier Bytes bestimmen den Typ:

Wert	Beschreibung
0x10	Inline Attribute
0x20	Fork Data Attribute
0x30	Extension Attribute

Tabelle 31: Eintrags-Typen im Attributes File

[120] Singh 2007, S. 1524
[121] Apple: *Technical Note TN1150*, S. 28
[122] Apple: *Technical Note TN1150*, S. 28–29
[123] Singh 2007, S. 1525

Inline Attributes benötigen keine Extents und sind somit vollständig im *Attributes File* in einem B-Tree Eintrag gespeichert. Sie haben folgende Struktur:

Größe (Byte)	Beschreibung
4	Eintrags-Typ 0x10 für Inline Attribute
8	reserviert
4	Größe des Attributs
variabel	Daten des Attributs

Tabelle 32: Struktur der Inline Attributes im Attributes File

Das *Fork Data Attribute* wird für große Attribute genutzt. Die Daten dieses Attributs sind in Extents gespeichert. Die Attribut Einträge beinhalten die Position und die Größe der zugehörigen Extents.

Größe (Byte)	Beschreibung
4	Eintrags-Typ 0x20 für Fork Data Attribute
4	reserviert
80	Fork Data Struktur beschreibt Größe und Position des Attributs Data Streams

Tabelle 33: Struktur der Fork Data Attributes im Attributes File

Extension Attributes erweitern *Fork Data Attributes*, wenn mehr als acht Extents benötigt werden.

Größe (Byte)	Beschreibung
4	Eintrags-Typ 0x30 für Extension Attribute
4	reserviert
64	Extent Descriptor beschreibt acht zusätzliche Overflow Extents für Fork Stream

Tabelle 34: Struktur der Extension Attributes im Attributes File

Es gibt viele verschiedene Attributs-Typen. Einige für forensische Untersuchungen bedeutsame Attribute können folgende sein:

Extended Attribut	Beschreibung
com.apple.decmpfs	HFS Plus Dateikomprimierung
com.apple.quarantine	Apple Sandbox (Kennzeichen für aus dem Internet geladenen Daten)
com.apple.system.Security	Access Control Lists
com.apple.metadata	Spotlight Metadaten
com.apple.backupd	Time Machine Daten
Applikationen von Drittanbietern	z. B. Dropbox, OneDrive (Microsoft)
com.apple.FinderInfo	Verweis zu den Datei- oder Ordner-Informationen für den Finder
com.apple.ResourceFork	Verweis zu den Resource Fork Daten der Datei

Tabelle 35: Beispielhafte Aufzählung von Extended Attributes

Für die Bezeichnung jedes Attributes wird ein umgekehrtes DNS-Format genutzt, zum Beispiel *com.apple.quarantine*. *Extended Attributes* enthalten Dateien für eigene Zwecke und haben kein statisches Format.

Die Metadaten „com.appleFinderInfo“ und „com.apple.ResourceFork“ werden nicht direkt im *Attributes File* gespeichert. Es handelt sich hierbei um Verweise auf die entsprechenden Metadaten.[124]

5.3.4.1 Analyse-Szenario

Als Beispielszenario dient das zuvor für das *Catalog File* und *Extents Overflow File* verwendete Beispielimage „image_3_2“. Dort soll wieder die Datei „datei_a.txt“ mit der CNID 24 analysiert werden. Wie bereits festgestellt, liegt diese im Root Ordner mit der Parent ID 2.

Um jetzt die *Extended Attributes* analysieren zu können, wird das *Attributes File* des HFS Plus Volumes untersucht.

Da es sich um eine B-Tree-Struktur handelt, muss im ersten Schritt wieder der Header Node am Byte-Offset 0 des *Attributes Files* aufgesucht werden. Aus den Informationen des B-Tree Node Descriptors und des darauffolgenden Header Node Eintrags lassen sich so wichtige Informationen über den Aufbau und die Größe der Nodes und der Node Typen gewinnen.

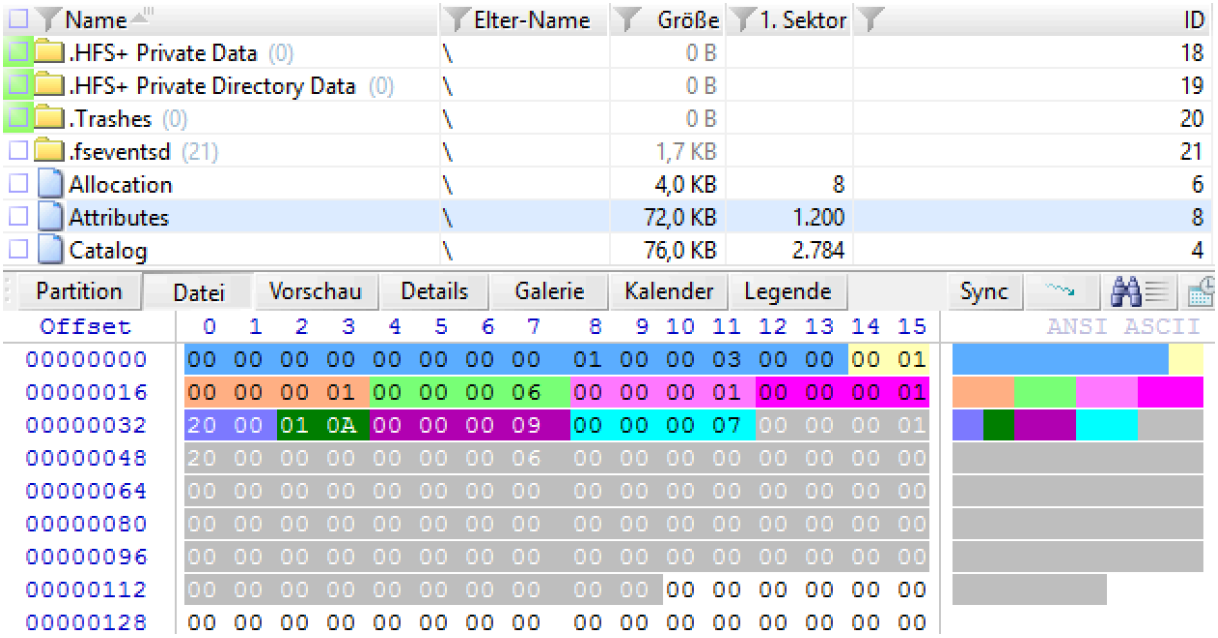


Abbildung 45: Header Node des Attributes Files

In der Abbildung blau markiert ist der Node Descriptor des Header Nodes vom *Attributes File*. Am Byte-Offset 8 bestätigt dies der Wert 1, der den Node Typ beschreibt und hier für Header Node steht. Der Header Node hat drei Einträge (Offset 10-11).

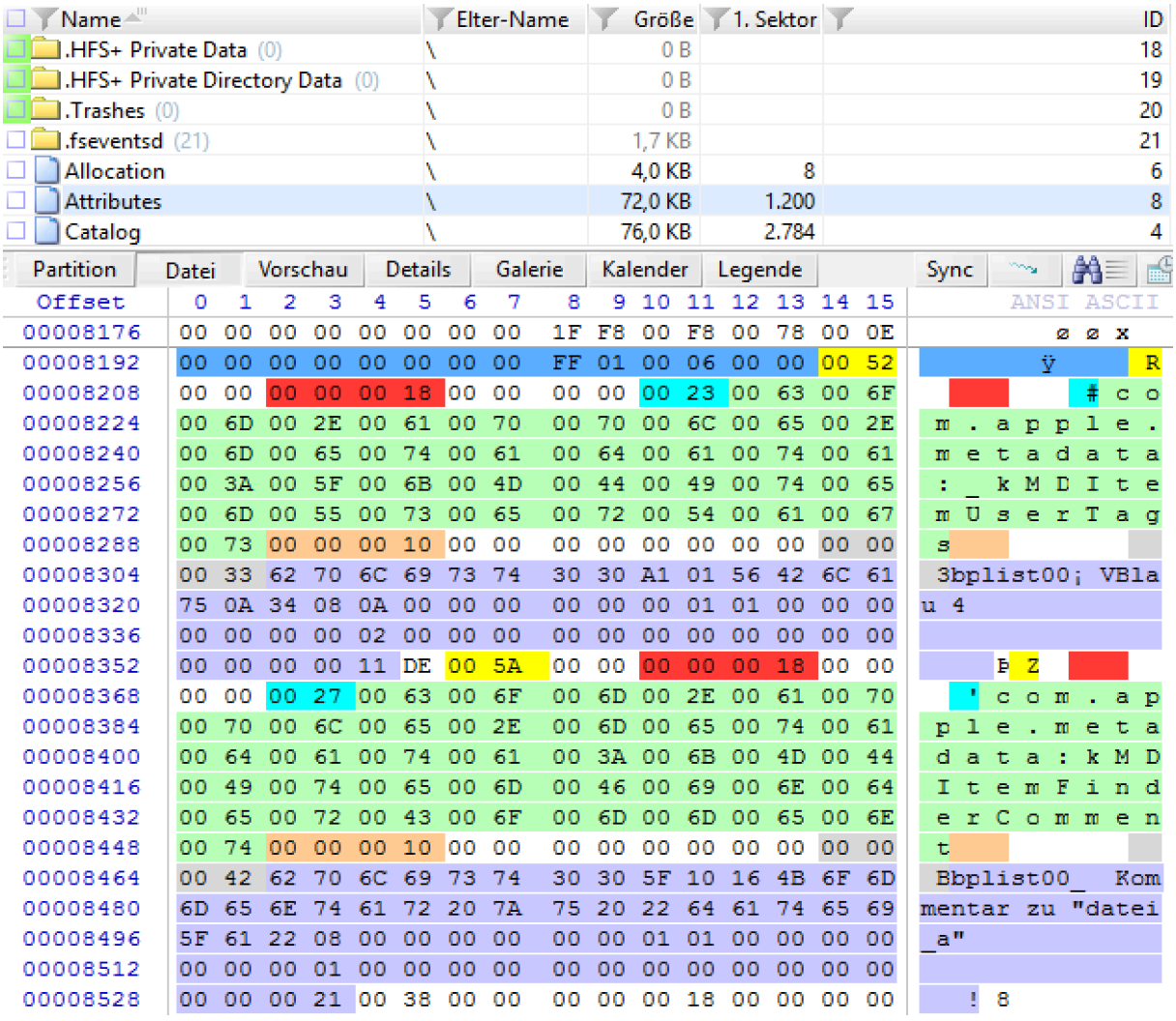
Der erste Eintrag ist der Header Node Eintrag, der direkt nach dem Node Descriptor ab Byte-Offset 14 beginnt. Die ersten zwei Bytes (gelbe Markierung) geben die Node Tiefe an, die 1 beträgt. Als nächstes folgen vier Bytes mit der ID des Root Nodes (orangene Markierung), die im Beispiel 1 ist. Offset 6 ausgehend vom Header Node Eintrag gibt an, dass sechs Leaf Node

[124] Singh 2007, S. 1526

Einträge im gesamten B-Tree vorhanden sind (grüne Markierung). Die nächsten acht Bytes geben jeweils die ID des ersten Leaf Nodes (rosa) und die ID des letzten Leaf Nodes (pink) an. Da sie bei beiden Einträgen den Wert 1 enthalten, existiert nur ein Leaf Node im B-Tree, der belegt ist. Ein weiterer wichtiger Wert im Header Node Eintrag ist die Information über die Größe der Nodes im B-Tree, die hier violett gekennzeichnet ist und demnach 8192 Bytes beträgt. Die maximale Key-Länge beträgt 266 Bytes (dunkelgrüne Markierung am Eintrags-Offset 20). Der vier Byte große Wert ab Byte-Offset 22 (bordeaux) zeigt die Anzahl der Nodes im gesamten B-Tree. Die folgenden vier Bytes (hellblau) geben die Anzahl der freien Nodes im gesamten B-Tree an.

Anhand dieser Informationen kann nun der relevante Node mit den zusätzlichen Attributen der Datei „datei_a.txt“ gefunden werden. Da aus dem Header Eintrag hervorgeht, dass von neun Nodes im B-Tree zwei belegt sind, existiert nur noch ein zweiter Node im B-Tree, der untersucht werden muss. Dieser zweite Node kann demnach nur der Root Node sein, dessen ID 1 beträgt. Da alle Nodes im B-Tree eine feste Größe von 8192 Bytes haben, kann die Position des Root Nodes einfach aus dem Produkt von 1 und 8192 bestimmt werden.

Im *Attributes File* kann nun der Offset 8192 aufgesucht werden, der den Anfang vom Root Node in dieser B-Tree Struktur darstellt.



Name	Elter-Name	Größe	1. Sektor	ID
.HFS+ Private Data (0)	\	0 B		18
.HFS+ Private Directory Data (0)	\	0 B		19
.Trashes (0)	\	0 B		20
.fseventsd (21)	\	1,7 KB		21
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00008176	00	00	00	00	00	00	00	00	1F	F8	00	F8	00	78	00	0E	ø ø x
00008192	00	00	00	00	00	00	00	00	FF	01	00	06	00	00	00	52	ÿ R
00008208	00	00	00	00	00	18	00	00	00	00	00	23	00	63	00	6F	# c o
00008224	00	6D	00	2E	00	61	00	70	00	70	00	6C	00	65	00	2E	m . a p p l e .
00008240	00	6D	00	65	00	74	00	61	00	64	00	61	00	74	00	61	m e t a d a t a
00008256	00	3A	00	5F	00	6B	00	4D	00	44	00	49	00	74	00	65	: _ k M D I t e
00008272	00	6D	00	55	00	73	00	65	00	72	00	54	00	61	00	67	m U s e r T a g
00008288	00	73	00	00	00	10	00	00	00	00	00	00	00	00	00	00	s
00008304	00	33	62	70	6C	69	73	74	30	30	A1	01	56	42	6C	61	3bplist00¡ VBla
00008320	75	0A	34	08	0A	00	00	00	00	00	00	01	01	00	00	00	u 4
00008336	00	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	
00008352	00	00	00	00	11	DE	00	5A	00	00	00	00	00	18	00	00	Þ Z
00008368	00	00	00	27	00	63	00	6F	00	6D	00	2E	00	61	00	70	' c o m . a p
00008384	00	70	00	6C	00	65	00	2E	00	6D	00	65	00	74	00	61	p l e . m e t a
00008400	00	64	00	61	00	74	00	61	00	3A	00	6B	00	4D	00	44	d a t a : k M D
00008416	00	49	00	74	00	65	00	6D	00	46	00	69	00	6E	00	64	I t e m F i n d
00008432	00	65	00	72	00	43	00	6F	00	6D	00	6D	00	65	00	6E	e r C o m m e n
00008448	00	74	00	00	00	10	00	00	00	00	00	00	00	00	00	00	t
00008464	00	42	62	70	6C	69	73	74	30	30	5F	10	16	4B	6F	6D	Bbplist00_ Kom
00008480	6D	65	6E	74	61	72	20	7A	75	20	22	64	61	74	65	69	mentar zu "datei
00008496	5F	61	22	08	00	00	00	00	00	00	01	01	00	00	00	00	_a"
00008512	00	00	00	01	00	00	00	00	00	00	00	00	00	00	00	00	
00008528	00	00	00	21	00	38	00	00	00	00	00	18	00	00	00	00	! 8

Abbildung 46: Root Node im Attributes File mit den ersten zwei Leaf Node Einträgen

Die in der Abbildung 46 blau markierten 14 Bytes beinhalten den Node Descriptor für diesen Node, der mit einer Größe von 8192 Bytes Offset 8192-16368 im B-Tree einnimmt. Die ersten acht Bytes geben an, dass keine weiteren Nodes des gleichen Typs im *Attributes File* existieren, da die Werte für die IDs der Nodes des nächsten und vorherigen Nodes gleichen Typs 0 sind. Für das Descriptor Feld für den Node Typ steht vorzeichenbehaftet 0xFF (-1), was den Node als Leaf Node kennzeichnet. Da aus der Analyse des Header Nodes bekannt ist, dass sechs Leaf Node Einträge im *Attributes File* vorhanden sind, müssen diese Einträge in diesem Node gespeichert sein, was auch durch das 2-Byte Feld für die Anzahl der Einträge im Node am Node-Offset 10 mit dem Wert 6 bestätigt wird und zudem keine weiteren Nodes im B-Tree belegt sind (siehe vorherige Ausführungen).

Die Angaben zu den Offsets der sechs Leaf Node Einträge befinden sich am Ende des Nodes als 2-Byte Big-Endian Integer-Werte für jeden Eintrag (siehe Abbildung 47).

Um die zur Datei „datei_a.txt“ mit der CNID 24 gehörenden Extended Attributes-Daten zu finden, müssen die Key-Daten der Leaf Node Einträge analysiert werden. Der Offset zum ersten Leaf Node Eintrag ausgehend vom Beginn des Nodes steht am Offset 8191-8192 (Offset im *Attributes File* 16367-16368) des Leaf Nodes (siehe rote Markierung in Abbildung 47):

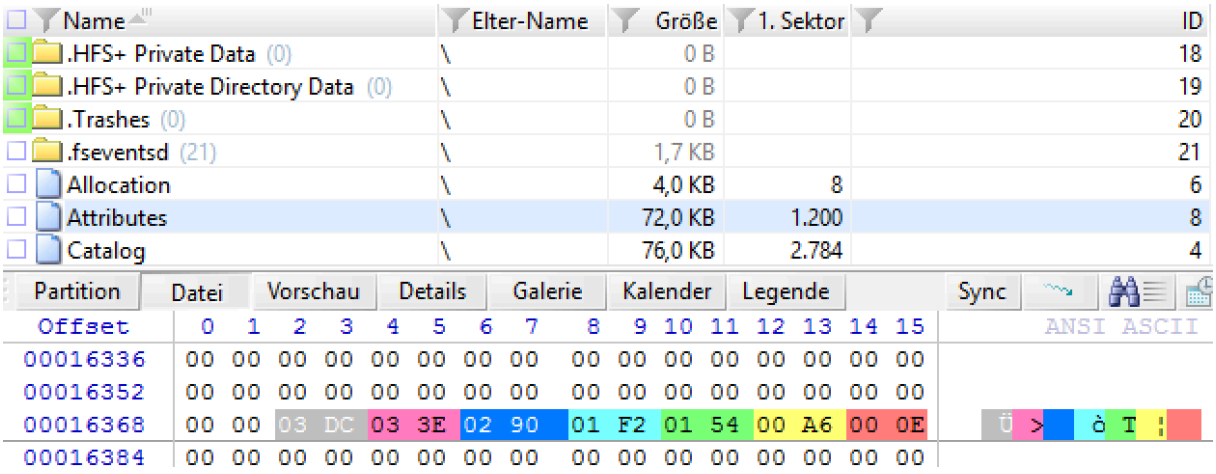


Abbildung 47: Offsets der Leaf Node Einträge von Node Nr. 1

Der erste Eintrag beginnt am Offset 14 vom Anfang des Leaf Nodes. Die ersten zwei Bytes im Eintrag beinhalten für den zu analysierenden Eintrag eine Key-Länge von 82 Bytes (siehe gelbe Markierung in Abbildung 46). Die rot markierten vier Bytes ab Offset 16 beinhalten hier den Wert 0x18 (rote Markierung), was der CNID 24 der Datei „datai_a.txt“ entspricht. In der Key-Struktur kann nun die Größe des Attribut-Namens am Offset 22 (Türkis) mit einem 2-Byte-Big-Endian Integer-Wert von 35 entnommen werden. Mit dieser Angabe kann nun der vollständige Name des Attributs bestimmt werden. Da es sich hier auch um eine 2-Byte Unicode255-Struktur handelt, werden 70 Bytes für den Attribut-Namen benötigt. In der Abbildung wird der Attribut-Name grün gekennzeichnet und lautet: *com.apple.metadata:_kMDItemUserTags*. Nach dem Attribut-Namen folgt ein 4-Byte-Big-Endian Integer-Wert (orange), der den Typ des Eintrags bestimmt. In diesem Fall beträgt dieser 0x10, was für *Inline Attribut*e steht. Dies bedeutet, dass die Attribut-Daten vollständig im *Attributes File* gespeichert sind. Da die Größe hier auch variabel ist, wird mit einem 4-Byte-Big-Endian Integer-Wert, der hier grau markiert ist, die Größe der Attribut-Daten angegeben.

Diese beträgt für den ersten Attribut Eintrag im Beispiel 51 Bytes. Die Attribut-Daten lassen sich so vollständig erheben. Sie sind violett markiert. Es handelt sich hierbei um eine binäre PLIST-Datei mit dem Inhalt „VBlau 4“. Hierbei handelt es sich um einen Finder Tag mit der Farbe Blau. Tags können im Mac Dateiexplorer „Finder“ im Mac OS X 10.11 System vom Benutzer für Dateien und Verzeichnisse gesetzt werden, um sie zum Beispiel nach besonderer Relevanz zu kennzeichnen.

Der zweite Eintrag im Leaf Node kann mit der gleichen Vorgehensweise und Struktur analysiert werden. Der Node-Offset für den Eintrag befindet sich an Position 8189-8190 und ist in der Abbildung 47 gelb markiert. Für die Key-Struktur und den Eintrag wurden die gleichen farblichen Markierungen in der Abbildung 46 verwendet. Der zweite Attribut-Eintrag gehört ebenfalls zur Datei mit der CNID 24 und ist ein *Inline Attribute*, das vollständig im *Attributes File* gespeichert ist. Der Attribut-Name lautet hier *com.apple.kMDItemFinderComment* mit den Attribut-Daten als binäre PLIST: Dies ist ein Kommentar zu „datei_a“. Hierbei handelt es sich um einen Spotlight-Kommentar, der vom Benutzer in einem Mac OS X 10.11 System für Dateien und Verzeichnisse vergeben werden kann.

Folgende Abbildung zeigt die Datei „datei_a.txt“ in einer nativen Finder-Umgebung im Mac OS X 10.11 System mit den zuvor analysierten Metainformationen aus dem *Attributes File*.

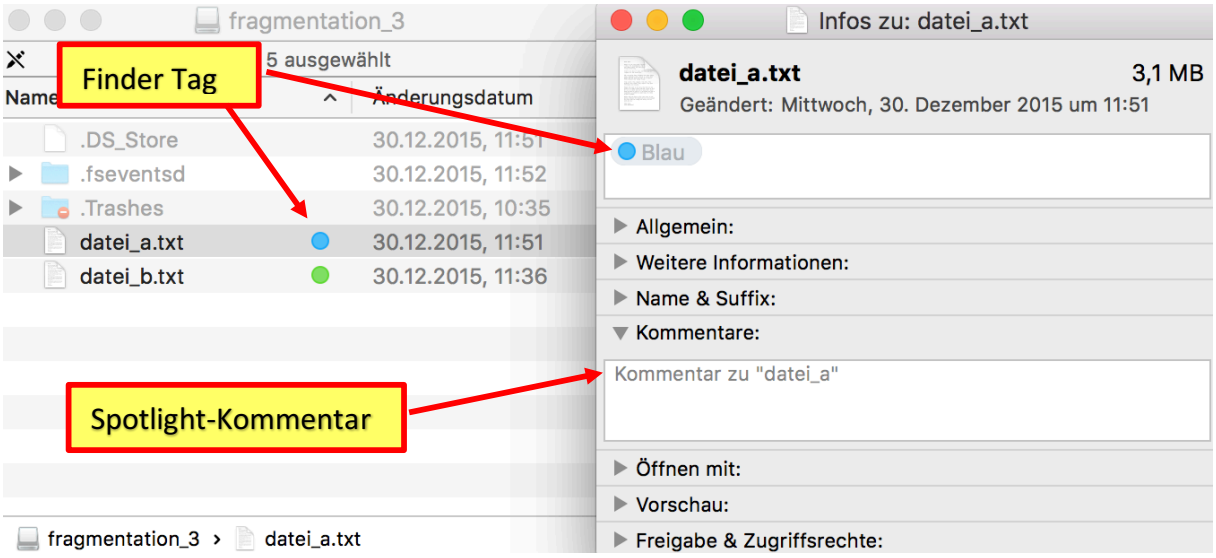


Abbildung 48: Extended Attributes in nativer Ansicht des Finders

Folgende Abbildung gibt einen Überblick über die Node Struktur des *Attributes Files* aus dem Beispielimage „image_3_2“ mit zusätzlichen Metadaten der Datei „datei_a.txt“:

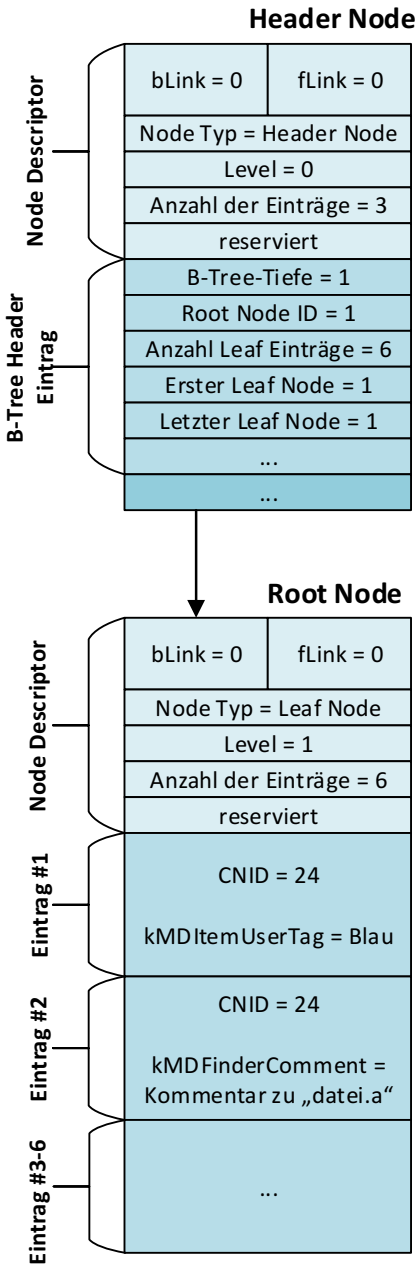


Abbildung 49: Node Struktur im Attributes File

5.3.5 Analysetechniken

Um eine relevante Datei mit den Metadaten aus dem Catalog Node Eintrag analysieren zu können, muss im ersten Schritt auf Disk- und Dateisystemebene die Volume-Struktur analysiert werden.

Mit *mmls* aus dem TSK-Tool-Set kann der Start des HFS Plus Volumes der Disk ermittelt werden. Folgende Abbildung zeigt, dass am LBA 40 das Volume mit der Bezeichnung „disk image“ beginnt.

```
MacBook:images_final Lars$ mmls image_3_2.E01
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors

     Slot    Start        End          Length       Description
00:  Meta    0000000000   0000000000   0000000001   Safety Table
01:  -----   0000000000   0000000039   0000000040   Unallocated
02:  Meta    0000000001   0000000001   0000000001   GPT Header
03:  Meta    0000000002   0000000033   0000000032   Partition Table
04:  00      0000000040   0000019535   0000019496   disk image
05:  -----   0000019536   0000019571   0000000036   Unallocated
```

Abbildung 50: Diskstruktur mit mmls

Auf Dateisystem-Ebene kann nun mit *fsstat* die Struktur des HFS Plus Volumes erhoben werden.

```
MacBook:images_final Lars$ fsstat -o 40 image_3_2.E01
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: HFS+
File System Version: HFS+

Volume Name: fragmentation_3
Volume Identifier: b87616ced527b3f9

Last Mounted By: Mac OS X, Journaled
Volume Unmounted Properly
Mount Count: 9852

Creation Date:  2015-12-30 10:35:37 (CET)
Last Written Date:      2015-12-30 11:52:02 (CET)
Last Backup Date:       0000-00-00 00:00:00 (UTC)
Last Checked Date:      2015-12-30 10:35:37 (CET)

Journal Info Block: 2

METADATA INFORMATION
--------------------------------------------
Range: 2 - 41
Bootable Folder ID: 0
Startup App ID: 0
Startup Open Folder ID: 0
Mac OS 8/9 Blessed System Folder ID: 0
Mac OS X Blessed System Folder ID: 0
Number of files: 16
Number of folders: 4

CONTENT INFORMATION
--------------------------------------------
Block Range: 0 - 2436
Allocation Block Size: 4096
Number of Free Blocks: 13
```

Abbildung 51: Informationen über HFS Plus Volume

Als Annahme wird das vorherige Beispiel „image_3_2“ genommen. Als Datei von Relevanz dient wieder die Datei „datei_a.txt“ mit der CNID 24.

Um die CNID der Datei zu ermitteln, können zum Beispiel mit dem TSK Befehl *fls* auf Kategorienebene der Dateinamen alle Dateien und Verzeichnisse mit zugehöriger CNID ausgegeben bzw. es kann nach speziellen Dateinamen gesucht werden:

```
MacBook:Images Lars$ fls -o 40 fragmentation_3.dmg
r/r 3:  $ExtentsFile
r/r 4:  $CatalogFile
r/r 5:  $BadBlockFile
r/r 6:  $AllocationFile
r/r 8:  $AttributesFile
r/r 23: .DS_Store
d/d 21: .fseventsd
d/d 19: .HFS+ Private Directory Data^
r/r 16: .journal
r/r 17: .journal_info_block
d/d 20: .Trashes
r/r 24: datei_a.txt
r/r 25: datei_b.txt
d/d 18: ^^^^HFS+ Private Data
```

Abbildung 52: Ausgabe der Dateien und Verzeichnisse im Root-Verzeichnis mit TSK

Für die Kategorie Metadaten können anhand der CNID 24 der Datei die Metadaten mit *istat* aus dem TSK-Tool-Set abgerufen werden:

```
MacBook:images_final Lars$ istat -o 40 image_3_2.E01 24
File Path: /datei_a.txt
Catalog Record: 24
Allocated
Type:    File
Mode:    rrw-r--r--
Size:    3124576
uid / gid: 501 / 20
Link count:      1

File Name: datei_a.txt
Admin flags: 0
Owner flags: 0
Has extended attributes
File type:       0000
File creator:    0000
Text encoding:   0 = MacRoman
Resource fork size:      0

Times:
Created:         2015-12-30 10:11:03 (CET)
Content Modified:        2015-12-30 11:51:17 (CET)
Attributes Modified:     2015-12-30 11:51:17 (CET)
Accessed:        2015-12-30 11:49:57 (CET)
Backed Up:       0000-00-00 00:00:00 (UTC)

Data Fork Blocks:
2427-2435  2398-2399  2421-2422  2414  2417
168-170  173-347  367-556  558-584
2024-2026  2050-2397  2403-2404

Attributes:
Type: ExATTR (4354-2)   Name: com.apple.metadata:_kMDItemUserTags   Resident   size: 51
Type: ExATTR (4354-3)   Name: com.apple.metadata:kMDItemFinderComment   Resident   size: 66
Type: ExATTR (4354-4)   Name: com.dropbox.attributes   Resident   size: 83
Type: DATA (4352-0)   Name: DATA   Non-Resident   size: 3124576  init_size: 3124576
```

Abbildung 53: istat-Ausgabe zur Datei „datei_a.txt“

Die Analyse mit TSK wertet automatisiert nicht nur das *Catalog File* aus, sondern auch die Metadaten aus dem *Extents Overflow File* und dem *Attributes File*.

Es können grundlegende Metadaten wie Zeitstempel und Zugriffsrechte, die aus dem *Catalog File* stammen, aus der Ausgabe gewonnen werden.

Im Abschnitt „Data Fork Blocks“ der *istat*-Ausgabe werden alle Extents der Data Forks aufgeführt. Diese Informationen setzen sich aus dem Catalog Node Eintrag im *Catalog File* zur Datei „datei_a.txt“ und dem Eintrag im *Extents Overflow File* zusammen. Die ersten acht Extent Descriptors, in der Abbildung blau umrahmt, stammen aus dem Catalog Dateieintrag und die folgenden vier Extent Descriptors (rot umrahmt) aus dem Extents Overflow File Eintrag. Anhand der dargestellten Extent Descriptors lässt sich der Start Allocation Block und die Anzahl der Allocation Blocks der Extents ablesen. Der erste Extent vom Data Fork der Datei „datei_a.txt“ beginnt im Allocation Block 2427 und enthält neun zusammenhängende Allocation Blocks (2427-2435), der zweite Extent beginnt im Allocation Block 2398 und enthält zwei zusammenhängende Allocation Blocks (2398-2399) usw..

Mit diesen aus Metadaten gewonnenen Informationen kann mit *icat* aus dem TSK-Tool-Set der Inhalt der gesamten Allocation Blocks, die zur Datei gehören, ausgegeben werden. Mit *blkcat* aus dem TSK-Tool-Set kann der Inhalt der einzelnen Allocation Blocks auf der Inhaltsebene betrachtet werden. Die folgende Abbildung zeigt einen Auszug vom Inhalt der Datei „datei_a.txt“ in der *icat*-Ausgabe:

```
MacBook:images_final Lars$ icat -o 40 image_3_2.E01 24 | xxd -l 100
0000000: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000010: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000020: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000030: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000040: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000050: 6161 6161 6161 6161 6161 6161 6161 6161  aaaaaaaaaaaaaaaa
0000060: 6161 6161                                aaaa
```

Abbildung 54: icat-Ausgabe von Datei "datei_a.txt" (Auszug der ersten 100 Bytes)

Mit TSK können die Metadaten aus dem *Attributes File* ebenfalls analysiert werden. Aus der *istat*-Ausgabe (siehe Abbildung 53) können drei *Extended Attributes* unter „Attributes“ mit dem Typ-Feld „ExATTR“ entnommen werden. Von TSK wird zu jedem Attribut eine Nummer vergeben, zum Beispiel für *com.apple.metadata:_kMDItemUserTags* die Nummer 4354-2 (in Klammern in der Ausgabe in der Zeile des genannten Attributs). Diese Nummer wird von TSK zur Analyse vergeben; „4354“ steht für *Extended Attribute*s[125] und kann genutzt werden, um mit *icat* aus dem TSK-Tool-Set den Inhalt des Attributs zu analysieren:

```
MacBook:images_final Lars$ icat -o 40 image_3_2.E01 24-4354-2 | xxd
0000000: 6270 6c69 7374 3030 a101 5642 6c61 750a  bplist00..VBlau.
0000010: 3408 0a00 0000 0000 0001 0100 0000 0000  4...............
0000020: 0000 0200 0000 0000 0000 0000 0000 0000  ................
0000030: 0000 11                                  ...
```

Abbildung 55: Inhalt des Attributs com.apple.metadata:_kMDItemUserTags

Auch hier wird dem *icat*-Befehl der Offset LBA 40 aus der Dateisystemauswertung als Option übergeben, um auf das HFS Plus Volume des Beispielimages „image_3_2“, das an diesem Offset beginnt, zugreifen zu können. Anschließend folgt die CNID 24 der Datei „datei_a.txt“ und dann die Attribut-Nummer. Die Ausgabe wird mit *xxd* in eine hexadezimale Ansicht umgeleitet. Im Ergebnis ist der Finder Tag blau als binäre PLIST erkennbar.

Es können alternativ auch mit nativen Mac Terminal-Befehlen *Extended Attributes* analysiert werden. Dazu wird im Beispiel der Inhalt des Root Verzeichnisses mit *#ls –ila* aufgelistet.

[125] Sleuthkit Wiki: *HFS*

Sollten *Extended Attributes* vorhanden sein, wird im Feld *File Mode* ein „@“ ausgegeben (blau umrahmt):

```
MacBook:fragmentation_3 Lars$ ls -ila
total 17800
    2 drwxr-xr-x@  9 Lars  staff      374 30 Dez 11:51 .
13727 drwxrwxrwt@  6 root  admin      204 21 Jan 16:18 ..
   23 -rw-r--r--@  1 Lars  staff    10244 30 Dez 11:51 .DS_Store
   20 d-wx-wx-wt   2 Lars  staff       68 30 Dez 10:35 .Trashes
   21 drwx------  13 Lars  staff      442 30 Dez 11:52 .fseventsd
   24 -rw-r--r--@  1 Lars  staff  3124576 30 Dez 11:51 datei_a.txt
   25 -rw-r--r--@  1 Lars  staff  5975424 30 Dez 11:36 datei_b.txt
```

Abbildung 56: Inhalt eines Verzeichnisses mit Terminal-Befehl ls auflisten

Die Metadaten der Datei „datei_a.txt“ können anschließend mit dem Terminal-Befehl *xattr* angezeigt werden; die Option *xl* zeigt den Namen und den Inhalt der Attribute:

```
MacBook:fragmentation_3 Lars$ xattr -xl datei_a.txt
com.apple.FinderInfo:
00000000  00 00 00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00  |..................|
00000010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |..................|
00000020
com.apple.metadata:_kMDItemUserTags:
00000000  62 70 6C 69 73 74 30 30 A1 01 56 42 6C 61 75 0A  |bplist00..VBlau.|
00000010  34 08 0A 00 00 00 00 00 00 01 01 00 00 00 00 00  |4...............|
00000020  00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000030  00 00 11                                         |...|
00000033
com.apple.metadata:kMDItemFinderComment:
00000000  62 70 6C 69 73 74 30 30 5F 10 16 4B 6F 6D 6D 65  |bplist00_..Komme|
00000010  6E 74 61 72 20 7A 75 20 22 64 61 74 65 69 5F 61  |ntar zu "datei_a|
00000020  22 08 00 00 00 00 00 00 01 01 00 00 00 00 00 00  |"...............|
00000030  00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000040  00 21                                            |.!|
00000042
com.dropbox.attributes:
00000000  78 9C AB 56 4A 29 CA 2F 48 CA AF 88 4F CB CC 49  |x..VJ)./H...O..I|
00000010  CD 4C 89 CF C9 4F 4E CC 51 B2 52 A8 56 CA 4D 4C  |.L...ON.Q.R.V.ML|
00000020  CE C8 CC 03 89 25 96 94 14 81 85 52 12 4B 12 81  |.....%.....R.K..|
00000030  0C A5 82 70 47 E7 E4 D2 92 64 E3 90 48 23 67 EF  |...pG....d..H#g.|
00000040  2A 17 93 CA D0 94 F4 40 5B 5B A5 DA DA 5A 00 C8  |*......@[[...Z..|
00000050  77 1C 81                                         |w..|
00000053
```

Abbildung 57: Auflistung von Attributen der Datei "TN2166_GPT.pdf" mit dem Terminal-Befehl xattr

Im Beispiel hat die Datei „datei_a.txt“ drei Attribut-Einträge, die im *Attributes File* gespeichert sind. „com.apple.FinderInfo“ ist ein „Fake-Attribut“, da es nur ein Verweis auf Finderdaten ist und selbst keinen Inhalt im *Attributes File* gespeichert hat.

5.3.6 Besonderheiten bei der Analyse von HFS Plus Dateikomprimierung

Das Attribut *com.apple.decmpfs* ist besonders interessant, da es darauf hindeutet, dass die entsprechende Datei unter Nutzung der HFS Komprimierung komprimiert ist. Beispiele für komprimierte Daten sind Systemprogramme *ls*, *echo* oder *dd* im Verzeichnis */bin* eines Mac OS X Systems.

Dateikomprimierung spielt eine besonders wichtige Rolle bei der forensischen Analyse des HFS Plus Dateisystems, da es in der Regel von vielen forensischen Untersuchungsprogrammen nicht verarbeitet werden kann und so unter Umständen vom IT-Forensiker nicht erkannt wird. Bei der Analyse mit X-Ways Forensics (18.4) und Forensic Toolkit (6.0.1) werden die HFS Plus komprimierten Dateien zum Beispiel mit einer Größe von 0 Bytes-Dateien dargestellt und nicht dekomprimiert.

Die HFS Plus Dateikomprimierung wurde mit Mac OS 10.6 (Snow Leopard) 2009 von Apple eingeführt. Das zuvor angesprochene *Extended Attribute com.apple.decmpfs* im *Attributes File* kennzeichnet die entsprechende Datei als komprimiert. Die komprimierten Daten werden im Resource Fork gespeichert, der Data Fork der Datei hat die Größe 0 Byte. Im Mac OS X System werden Dekomprimierungen in Echtzeit im Hintergrund durchgeführt.[126]

Das Attribut com.apple.decmpfs wird vom *xattr* Befehl gefiltert und im Terminal nicht ausgegeben. Mit *#ls -lO* kann aber im Terminal gelistet werden, welche Dateien komprimiert sind; gibt aber keine weiteren Details zu Komprimierung:

```
MacBook:decmpfs Lars$ ls -lO
total 160
-rw-r-----  1 Lars  staff  -           1103 17 Nov 14:04 HFSPlus_GPT_01.E01.txt
-rw-r-----@ 1 Lars  staff  -          70696 13 Dez 13:03 HFSPlus_GPT_01.E01_Kopie.txt
-rw-r-----@ 1 Lars  staff  compressed 70696 13 Dez 13:03 HFSPlus_GPT_01.E01_Kopie_decmpfs.txt
-rw-r-----@ 1 Lars  staff  compressed  1103 17 Nov 14:04 HFSPlus_GPT_01.E01_decmpfs.txt
```

Abbildung 58: HFS Plus Komprimierung mit ls erkennen

Eine detailliertere Möglichkeit, *Extended Attributes* einer Datei zu überprüfen, bietet TSK mit dem bereits vorgestellten Befehl *istat*. Die zu untersuchende Datei aus dem Beispiel „image_4“ heißt „HFSPlus_GPT_01.E01_Kopie_decmpfs.txt“ und hat die CNID 27:

```
MacBook:images_final Lars$ istat -o 409640 image_4.E01 27
File Path: /HFSPlus_GPT_01.E01_Kopie_decmpfs.txt
Catalog Record: 27
Allocated
Type:    File
Mode:    rrw-r-----
Size:    70696
uid / gid: 501 / 20
Link count:     1

File Name: HFSPlus_GPT_01.E01_Kopie_decmpfs.txt
Admin flags: 0
Owner flags: 96 - compressed
Has extended attributes
File type:      0000
File creator:   0000
Text encoding:  0 = MacRoman
Resource fork size:     2008

Times:
Created:        2015-12-13 13:03:10 (CET)
Content Modified:       2015-12-13 13:03:10 (CET)
Attributes Modified:    2015-12-13 13:22:37 (CET)
Accessed:       2015-12-13 13:29:45 (CET)
Backed Up:      0000-00-00 00:00:00 (UTC)

Resource Fork Blocks:
39903

Attributes:
Type: ExATTR (4354-2)   Name: com.apple.TextEncoding   Resident   size: 15
Type: CMPF (4355-3)   Name: com.apple.decmpfs   Resident   size: 16
Type: RSRC (4353-1)   Name: RSRC   Non-Resident   size: 2008  init_size: 2008
Type: DATA (4352-0)   Name: DATA   Non-Resident, Compressed   size: 2008  init_size: 2008

Compressed File:
    Uncompressed size: 70696
    Data is zlib compressed in the resource fork

Resources:
  Type: cmpf    ID: 1           Offset: 260     Size: 1698      Name: <none>
```

Abbildung 59: HFS Plus Komprimierung mit TSK erkennen

[126] Levin 2013, S. 612–613

Anhand des Beispiels aus obiger Abbildung ist erkennbar, dass das *Extended Attribute com.apple.decmpfs* vorhanden ist und die komprimierten Daten im Resource Fork gespeichert sind. Relevante Informationen über die Dateikomprimierung wurden in der Abbildung rot umrahmt. Die Datei „HFSPlus_GPT_01.E01_Kopie_decmpfs.txt“ mit der CNID 27 hat das *Owner Flag* „compressed“ gesetzt und enthält Resource Fork Daten im Allocation Block 39903. Anhand der Attribut Daten „Type:CMPF“ mit dem Namen „com.apple.decmpfs“ lässt sich entnehmen, dass die Datei im *Attributes File* einen Eintrag besitzt, der die Datei als HFS komprimiert kennzeichnet. Die *istat*-Ausgabe gibt detaillierte Informationen über Komprimierungsalgorithmus und Position des Allocation Blocks mit dem Inhalt der komprimierten Daten.

Um die Attribut-Daten betrachten zu können, kann mit *icat* aus dem TSK-Tool-Set der Inhalt im Terminal ausgegeben werden. Folgende Ausgabe zeigt den Inhalt des Attributs *com.apple.decmpfs* zur Datei „HFSPlus_GPT_01.E01_Kopie_decmpfs.txt“ mit dem Eintrag „fpmc“ im *Attributes File*:

```
MacBook:Images Lars$ icat -o 409640 decmpfs.dmg 27-4355-3 | xxd
0000000: 6670 6d63 0400 0000 2814 0100 0000 0000  fpmc....(.......
```

Abbildung 60: Inhalt des Extended Attributes com.apple.decmpfs

Für HFS komprimierte Daten verwendet TSK die interne Nummer 4355 anstatt der für *Extended Attributes* vergebenen Nummer 4354.

Folgende Aufzählung gibt einen Überblick über die wichtigsten Abkürzungen und Attribut-Nummern von TSK im Zusammenhang mit der Auswertung von HFS Plus Dateisystemen:[127]

- CMPF – Compressed File Data, 4355
- RSRC – Resource Fork, 4353
- DATA – Data Fork, 4352
- ExATTR – Extended Attributes, 4354

Wird der Inhalt vom Resource Fork aufgerufen, gibt TSK folgende Daten aus:

```
MacBook:images_final Lars$ icat -o 409640 image_4.E01 27-4353-1
??2?Xl6x^?Ko?8F??\NK??vy?       P?$?M????Z?+J?x~};Ic]tXx? p
??u?}?I;W*???B???Wk??u.&o??i???J??                        ???{?Eg?uU?M[??n깊 N?/?n?P????\??_
                                    ??N????O?z^e???G#?|H?{??]??e??`??N??U?>?\???Y??????}??f?5:OWj?>?N?
?R?6}?˛6ѧs??|?H??Qw▵???^??VoD????????N????ZZu+y?X??(z?G?5}??U2????3??yu????ES???z??A?d⁈|ي TV
                                                                                          )t??lEپ??
????jq?r?|W??|滅 Z??=???,?ك−??*_J??????I:)??<??r?de??q2−?xXf?d&????$9XS?????h4M??Y6:??Q6??L?I\???d?e3
{???t?[??h??M????7??A?n6*??d4???L???pA?km??q???r?−??ӽ?Zi??D0???2?f?ΰ8y~???ь=??3?%?_u??|???w??,??.?−H
??}\j?6??H?pr?w?3@??4???????9h#??r?i???@C?U0?
A#h??}??n2BY?????
 F??A?????OCg????????62?@WA#h??/t??r?B&?????SЙ恂 ?_? ?FA????jM??]"h???/??
                                                          ?bd??????9?4t?y???*?t??4?Fo?@'??f??ҹ?:?'?!
?W??+@??4????D9d!??r?)???@A??/?
?y`h?}??n12F??????:??<??}L?DA#h_~       *E?A???????O?f??5??w??4?Fo?@7?#md?O?C??
 BF?!??/??
     ??????OAg?
??~     @WA?,??zLx^?VMo?0??W?q??Q`
                                    RA?B?RU??L?*????e}]X?J????(qf⁇X'????E?O`??
*o??jo?mv??????????C????d&7?,?;?ze?E>Y₱+?i                                   +?ML?;?J{7?>??LYyz?z?
                                            0???Xg??8d??z????;????D???"<H?Ai?,ɘ????,?X?İ???1????n??,
                                                                                                  ??
???4h%!Ѣ?????I?8/%???FY??m?1??x???j−?E
??iV?J??n???R4?=?lm`??0?=?5Rr?p?e?Fm?????*T???)−k???ˀ01??o?mrę?9?o???I?????7???fb?P&X?S????p??M0?aru
?*?????yU[?È[??w(?Y???o?jQ˳&?K;?5??????g???H\??~p??N?v?t^J?n@?????/????1q$?2cmpf
```

Abbildung 61: Komprimierte Daten aus dem Resource Fork

[127] Sleuthkit Wiki: *HFS*

Es handelt sich hierbei um die komprimierten Daten der Datei „HFSPlus_GPT_01.E01_Kopie_decmpfs.txt“. Eine Überprüfung des Allocation Blocks für den Resource Fork mit der Nummer 39903 bestätigt dies mit einer identischen Ausgabe.[128]

Um die Daten mit TSK dekomprimiert betrachten zu können, kann das Data Fork Attribut mit *icat* aufgerufen werden:[129]

```
[MacBook:images_final Lars$ icat -o 409640 image_4.E01 27-4352-0                ]

Case Information:
Acquired using: ADI3
Case Number:
Evidence Number:
Unique Description:
Examiner:
Notes:

------------------------------------------------------------

Information for /Users/Lars/Desktop/share/Master/Images/HFSPlus_GPT_01:

Physical Evidentiary Item (Source) Information:
[Device Info]
 Source Type: Physical
[Drive Geometry]
 Bytes per Sector: 512
 Sector Count: 195353
[Physical Drive Information]
 Drive Model: Apple Lesen/Schreiben Media
 Source data size: 95 MB
 Sector count:    195353
[Computed Hashes]
 MD5 checksum:    7a7d34c00bf71bf1cc018f103fb919ef
 SHA1 checksum:   de66672481889b4653072b8e0fa170df0e4e8bb9

Image Information:
 Acquisition started:   Tue Nov 17 14:04:09 2015
 Acquisition finished:  Tue Nov 17 14:04:09 2015
Case Information:
Acquired using: ADI3
Case Number:
Evidence Number:
Unique Description:
Examiner:
Notes:

------------------------------------------------------------
```

Abbildung 62: Auszug aus dem dekomprimierten Inhalt der Datei „HFSPlus_GPT_01.E01_Kopie_decmpfs.txt“

Bei komprimierten Dateien mit geringer Größe werden die Daten direkt als *Inline Attribute* im *Attributes File* gespeichert, so dass kein Resource Fork benötigt wird.[130] Als Beispiel kann im Übungsimage „image_4“ die Datei „HFSPlus_GPT_01.E01_decmpfs.txt“ mit der CNID 25 analysiert werden.

[128] TSK-Befehl: #blkcat -o 409640 image_4.E01 39903

[129] Alternativ können die Daten dekomprimiert mit folgendem TSK-Befehl betrachtet werden: #icat –o 409640 image_4.E01 27

[130] Levin 2013, S. 614

```
MacBook:images_final Lars$ istat -o 409640 image_4.E01 25
File Path: /HFSPlus_GPT_01.E01_decmpfs.txt
Catalog Record: 25
Allocated
Type:    File
Mode:    rrw-r-----
Size:    1103
uid / gid: 501 / 20
Link count:      1

File Name: HFSPlus_GPT_01.E01_decmpfs.txt
Admin flags: 0
Owner flags: 32 - compressed
Has extended attributes
File type:       0000
File creator:    0000
Text encoding:   0 = MacRoman
Resource fork size:      0

Times:
Created:         2015-11-17 14:04:10 (CET)
Content Modified:        2015-11-17 14:04:10 (CET)
Attributes Modified:     2015-12-13 13:22:37 (CET)
Accessed:        2015-12-13 13:29:44 (CET)
Backed Up:       0000-00-00 00:00:00 (UTC)

Attributes:
Type: DATA (4352-0)   Name: DATA   Resident   size: 1103
Type: CMPF (4355-2)   Name: com.apple.decmpfs   Resident   size: 516
Type: ExATTR (4354-3)   Name: com.apple.metadata:_kMDItemUserTags   Resident   size: 51

Compressed File:
    Uncompressed size: 1103
    Data follows compression record in the CMPF attribute
    500 bytes of data at offset 16, zlib compressed
```

Abbildung 63: istat-Ausgabe von TSK bei HFS Plus Komprimierung mit Inline Attribute

Die rot umrahmten Informationen der TSK-Ausgabe geben Auskunft über die Dateikomprimierung und die entsprechenden Metadaten. Im Gegensatz zum ersten Beispiel ist Resource Fork hier mit 0 Allocation Blocks belegt. Die Daten sind direkt im *Attributes File* gespeichert.

Neben TSK 4.1.3 kann EnCase 7.11.01 in Verbindung mit EnScripts als kommerzielles Forensik Programm sämtliche Metadatenstrukturen im Rahmen der Analyse-Szenarien analysieren und ermöglicht eine umfassende Untersuchung von B-Tree-Strukturen der *Special Files* und von HFS komprimierten Dateien.

5.4 Kategorie Dateinamen

Wie in der Kategorie Metadaten bereits dargestellt, werden Metadaten durch B-Trees organisiert. Datei- und Verzeichnisnamen werden im jeweiligen Catalog Datei- oder Verzeichniseintrag gespeichert. Sie werden in *hfs_format.h* als HFSUniStr255 Struktur definiert, die aus 255 2-Byte Unicode Zeichen besteht.[131] Um entsprechende Namen zu finden, muss die B-Tree-Struktur des *Catalog Files* analysiert werden. Durch Vergleiche der Keys nach CNID und Namen im *Catalog File* können Einträge lokalisiert werden, die mit der gesuchten Datei zusammenhängen. Um den vollständigen Pfad ermitteln zu können, ist die Position des

[131] Apple: *hfs_format.h* 2012, S. 2

Root Verzeichnisses von Bedeutung. Das Root Verzeichnis hat immer die CNID 2 und die Parent ID 1.[132]

5.4.1 Links

Für die Analyse von HFS Plus Dateisystemen in der Kategorie Dateinamen haben Links zu Dateien und Verzeichnissen eine wichtige Rolle.

In einem HFS Plus Dateisystem gibt es drei Arten, wie Dateien und Verzeichnisse verknüpft sein können:

- Hard Links
- Symbolische Links
- Alias

5.4.1.1 Hard Links

In einem HFS Plus Dateisystem ist es möglich, dass mehrere Dateinamen auf eine Datei verweisen. Dies wird durch Hard Links realisiert. Da bei Erstellung eines Hard Links nicht mehr ersichtlich ist, welcher Verweis auf den Dateiinhalt das Original ist, wird der Dateiinhalt, auf den die verschiedenen Namen referenzieren, in spezielle indirekte Node Dateien (*iNodes*) gespeichert. Die iNodes liegen in einem Metadaten Ordner, der im Root Verzeichnis gespeichert ist, mit der Bezeichnung „ HFS+ Private Data“ (*/0x00/0x00/0x00/0x00HFS+ Private Data/)*. Die iNode Dateien setzen sich aus der Bezeichnung iNode und einer Referenznummer zusammen. Die Nummer ist einmalig im Volume vergeben und ist zufällig zwischen 100 und 1073741923 gewählt. Der Catalog Node Eintrag zum iNode gibt im Feld für die Zugriffsrechte „linkCount“ Informationen über die Anzahl der mit dem iNode verlinkten Dateinamen. Diese Zahl wird entsprechend inkrementiert oder dekrementiert, wenn Links hinzugefügt oder gelöscht werden (zur Struktur der Catalog Node Einträge siehe Abschnitt Kategorie Metadaten, Catalog File, Tabellen 21-24).[133]

Hard Links haben ebenfalls Einträge mit eigener CNID im *Catalog File*. Dort wird als Dateityp-Bezeichnung *hlnk* und als Ersteller *hfs+* verwendet.[134] Hier steht im Feld für die Zugriffsrechte „iNodeNum“ die Nummer des iNodes, mit dem der Eintrag verlinkt ist.

Hard Links spielen insbesondere im Mac OS X Systemen implementierten Backup Programm *Time Machine* eine wichtige Rolle. Hier werden inkrementelle Backups durch Hard Links ermöglicht, das heißt, dass unveränderte Dateien mit einem Hard Link zur Originaldatei referenziert und nur veränderte Dateien in der neuen Version gespeichert werden.

5.4.1.2 Symbolische Links

Symbolische Links im HFS Plus Dateisystem sind von der Funktionsweise mit denen anderer UNIX-Systeme identisch. Sie verweisen auf eine Datei oder auf ein Verzeichnis und beinhalten den Pfadnamen (absoluter Pfad) zu der verknüpften Datei oder zum verknüpften Verzeichnis. Die Pfade sind nach POSIX Standard bezeichnet und in UTF-8 gespeichert. Der Dateityp im Catalog File Eintrag im Feld für die Zugriffsrechte enthält als Wert *slnk* und im Feld für Ersteller

[132] Apple: *Technical Note TN1150*, S. 18–19
[133] Singh 2007, S. 47–50
[134] Apple: *Technical Note TN1150*, S. 30

den Wert *rhap*.[135] Symbolische Links werden also als normale Dateien im HFS Plus Dateisystem behandelt. Der Pfad zum verknüpften Ziel wird im Data Fork der Links gespeichert.

Im Mac Terminal oder im Finder zeigt der Link zum Original:

```
MacBook:links Lars$ ls -il
total 48
23 -rw-r--r--  4 Lars  staff  100 23 Jan 16:25 datei_a.txt
33 -rw-r--r--  1 Lars  staff  100 26 Jan 08:40 datei_b.txt
23 -rw-r--r--  4 Lars  staff  100 23 Jan 16:25 hardlink_1.txt
23 -rw-r--r--  4 Lars  staff  100 23 Jan 16:25 hardlink_2.txt
23 -rw-r--r--  4 Lars  staff  100 23 Jan 16:25 hardlink_3.txt
34 lrwxr-xr-x  1 Lars  staff   11 26 Jan 08:43 symlink_1.txt -> datei_b.txt
```

Abbildung 64: Symbolischer Link in der Terminal-Ausgabe

5.4.1.3 Alias

Alias sind symbolischen Links sehr ähnlich. Sie beinhalten die Pfadangaben zur Originaldatei im POSIX und HFS/HFS Plus-Format. Es können auch noch weitere Daten wie zum Beispiel Icons gespeichert sein, die dann im Resource Fork des Alias liegen. Der signifikante Unterschied zum symbolischen Link besteht darin, dass sich ein Alias durch einen relativen Pfad dynamisch der Zieldatei oder dem Zielordner anpasst, so dass, wenn eine Zieldatei oder ein Zielordner verschoben wird, der Alias weiterhin zur Zieldatei oder dem Zielordner zeigt.[136]

5.4.2 Analyse-Szenario

Als Beispiel dient das Image „image_5“ mit verschiedenen Link-Dateien, die analysiert werden sollen.

5.4.3 Analysetechniken

Im ersten Schritt können alle Datei- und Verzeichnisnamen rekursive aufgelistet werden, um einen Überblick über die Verzeichnisstruktur und das Root Verzeichnis zu bekommen. Dies kann im Beispiel mit *fls* durchgeführt werden:

[135] Apple: *Technical Note TN1150*, S. 31
[136] Singh 2007, S. 1554–1555

```
MacBook:Images Lars$ fls -a -r -o 40 image_5.E01
r/r 3:  $ExtentsFile
r/r 4:  $CatalogFile
r/r 5:  $BadBlockFile
r/r 6:  $AllocationFile
r/r 8:  $AttributesFile
d/- 1:  ..
r/r 32: .DS_Store
d/d 21: .fseventsd
+ d/d 2:        ..
+ r/r 28:       fc007595b19fe122
+ r/r 29:       fc007595b19fe123
+ r/r 30:       fc007595b19fe124
+ r/r 31:       fc007595b19fe125
+ r/r 35:       fc007595b1a035cb
+ r/r 36:       fc007595b1a035cc
+ r/r 39:       fc007595b1a0e264
+ r/r 40:       fc007595b1a0e265
+ r/r 22:       fseventsd-uuid
d/d 19: .HFS+ Private Directory Data^
+ d/d 2:        ..
r/r 16: .journal
r/r 17: .journal_info_block
d/d 20: .Trashes
+ d/d 2:        ..
r/r 38: alias_datei_c.txt
r/r 23: datei_a.txt
r/r 33: datei_b.txt
r/r 37: datei_c.txt
r/r 23: hardlink_1.txt
r/r 23: hardlink_2.txt
r/r 23: hardlink_3.txt
l/l 34: symlink_1.txt
d/d 18: ^^^^HFS+ Private Data
+ d/d 2:        ..
+ r/r 23:       iNode23
```

Abbildung 65: Auflistung aller Dateinamen

Da bekannt ist, dass das Root Verzeichnis in einem HFS Plus Dateisystem immer die CNID 2 und Parent ID 1 hat, ist hier einfach erkennbar, dass sich im Root Verzeichnis mehrere Dateien und Verzeichnisse befinden.

Auffallend sind drei Dateien im Root Verzeichnis mit der Bezeichnung „hardlink_“ im Dateinamen und die Datei „datei_a.txt“, da *fls* hier dieselbe CNID 23 anzeigt. Im Unterverzeichnis „ HFS+ Private Data“ mit der CNID 18 befindet sich eine weitere Datei „iNode23“, die ebenfalls die CNID 23 besitzt. Dies ist ein Hinweis darauf, dass es sich um eine hart-verlinkte Datei handeln könnte.

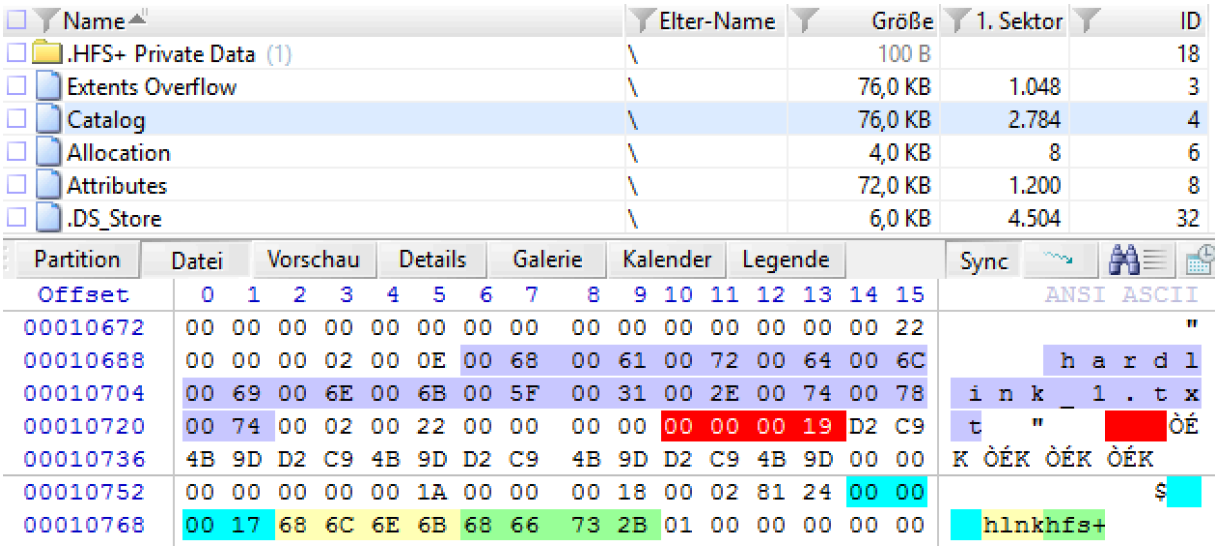
Im nächsten Schritt kann die CNID 23 untersucht werden, um anhand der Metadaten aus dem Catalog Node Eintrag mehrere Informationen erheben zu können:

```
MacBook:Images Lars$ istat -o 40 image_5.E01 23
File Path: /^^^^HFS+ Private Data/iNode23
Catalog Record: 23
Allocated
Type:    File
Mode:    rrw-r--r--
Size:    100
uid / gid: 501 / 20
Link count:      4

File Name: iNode23
This is a hard link to a file
Admin flags: 0
Owner flags: 0
File type:       0000
File creator:    0000
Text encoding:   0 = MacRoman
Resource fork size:      0

Times:
Created:         2016-01-23 16:25:23 (CET)
Content Modified:        2016-01-23 16:25:23 (CET)
Attributes Modified:     2016-01-23 16:26:13 (CET)
Accessed:        2016-01-23 16:25:23 (CET)
Backed Up:       0000-00-00 00:00:00 (UTC)

Data Fork Blocks:
558

Attributes:
Type: DATA (4352-0)   _Name: DATA    Non-Resident    size: 100   init_size: 100
```

Abbildung 66: istat-Ausgabe zur CNID 23

Es ist erkennbar, dass die mit dem Catalog Node Dateieintrag assoziierte Datei mit Namen „iNode23“ referenziert ist und dass es sich um eine mit vier Hard Links verknüpfte Datei handelt. Der Dateiinhalt liegt im mit der Datei allokierten Allocation Block 558. Bei den vier Dateien im Root Verzeichnis mit derselben CNID handelt es sich demnach um weitere Dateinamen, die auf diesen Inhalt und die dazugehörigen Metadaten verweisen. Es ist nicht ermittelbar, welcher Dateiname ursprünglich als Original mit dem Dateiinhalt und den Metadaten verknüpft war.

Da *fls* für die Dateinamen der Hard Links mit der referenzierten CNID 23 darstellt, können die Metadaten dieser Einträge mit TSK nicht aufgerufen werden. Im *Catalog File* existiert für jeden Dateinamen, der verlinkt wird, ein Eintrag. Als Beispiel wurde in der folgenden Abbildung für „hardlink_1.txt“ der zugehörige Catalog Dateieintrag aufgesucht. Dieser beginnt am Offset 10686 des *Catalog Files*.



Name	Elter-Name	Größe	1. Sektor	ID
.HFS+ Private Data (1)	\	100 B		18
Extents Overflow	\	76,0 KB	1.048	3
Catalog	\	76,0 KB	2.784	4
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
.DS_Store	\	6,0 KB	4.504	32

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00010672	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	22	"
00010688	00	00	00	02	00	0E	00	68	00	61	00	72	00	64	00	6C	h a r d l
00010704	00	69	00	6E	00	6B	00	5F	00	31	00	2E	00	74	00	78	i n k _ 1 . t x
00010720	00	74	00	02	00	22	00	00	00	00	00	00	00	19	D2	C9	t " ÒÉ
00010736	4B	9D	D2	C9	4B	9D	D2	C9	4B	9D	D2	C9	4B	9D	00	00	K ÒÉK ÒÉK ÒÉK
00010752	00	00	00	00	00	1A	00	00	00	18	00	02	81	24	00	00	$
00010768	00	17	68	6C	6E	6B	68	66	73	2B	01	00	00	00	00	00	hlnkhfs+

Abbildung 67: Catalog Node Eintrag zu "hardlink_1.txt"

Interessante Felder des Eintrags wurden in der Abbildung 67 markiert. Der Dateiname steht am Offset 8 der Key-Daten und lautet „hardlink_1.txt“ (violette Markierung). Die CNID des Eintrags wurde rot markiert und lautet 23 (0x00000019). Die türkisfarbene Markierung zeigt den Wert für das Feld *iNodeNum* innerhalb der Zugriffsrechte im Catalog Dateieintrag. Hierbei handelt es sich um die Referenznummer zum iNode, mit dem dieser Eintrag verlinkt ist und lautet 23 (0x00000017). Es folgt die Angabe zum Dateityp in Gelb mit „hlnk“ und anschließend im Ersteller-Feld des Eintrags in Grün hfs+. Die Strukturen des Eintrags können den Tabellen 21-23 entnommen werden.

Die Informationen bestätigen, dass es sich um einen hart-verlinkten Dateinamen handelt, der selbst die CNID 24 hat und auf einen iNode mit der ID 23 verweist. Der Inhalt kann anhand der CNID 23 oder durch Analyse der mit iNode23 allokierten Allocation Blocks betrachtet werden; innerhalb der Kategorien Inhalt und Metadaten.

Die Datei „symlink_1.txt“ soll als nächstes analysiert werden. Aus Abbildung 65 kann entnommen werden, dass sie die CNID 34 hat und mit „l“ als Link gekennzeichnet ist. Laut *istat*-Analyse handelt es sich um einen symbolischen Link (siehe Abbildung 68). TSK zeigt auch die Zielreferenz an, die hier „datei_b.txt“ darstellt. In einer weiteren Analyse kann nun „datei_b.txt“ untersucht werden.

```
MacBook:Images Lars$ istat -o 40 image_5.E01 34
File Path: /symlink_1.txt
Catalog Record: 34
Allocated
Type:
Mode:   lrwxr-xr-x
Size:   11
Symbolic link to:       datei_b.txt
uid / gid: 501 / 20
Link count:     1

File Name: symlink_1.txt
Admin flags: 0
Owner flags: 0
File type:      736c6e6b  slnk
File creator:   72686170  rhap
Text encoding:  0 = MacRoman
Resource fork size:     0

Times:
Created:        2016-01-26 08:43:48 (CET)
Content Modified:       2016-01-26 08:43:48 (CET)
Attributes Modified:    2016-01-26 08:43:48 (CET)
Accessed:       2016-01-26 08:43:48 (CET)
Backed Up:      0000-00-00 00:00:00 (UTC)

Data Fork Blocks:
566

Attributes:
Type: DATA (4352-0)   Name: DATA   Non-Resident   size: 11  init_size: 11
```

Abbildung 68: istat-Ausgabe zu "symlink_1.txt"

Als letztes Beispiel soll „alias_datei_c.txt“ (siehe Abbildung 65) analysiert werden:

```
MacBook:Images Lars$ istat -o 40 image_5.E01 38
File Path: /alias_datei_c.txt
Catalog Record: 38
Allocated
Type:   File
Mode:   rrw-r--r--
Size:   444684
uid / gid: 501 / 20
Link count:     1

File Name: alias_datei_c.txt
Admin flags: 0
Owner flags: 0
File type:      616c6973  alis
File creator:   4d414353  MACS
Is alias
Text encoding:  0 = MacRoman
Resource fork size:     956251

Times:
Created:        2016-01-26 09:45:11 (CET)
Content Modified:       2016-01-26 09:45:39 (CET)
Attributes Modified:    2016-01-26 09:45:39 (CET)
Accessed:       2016-01-26 09:45:39 (CET)
Backed Up:      0000-00-00 00:00:00 (UTC)

Data Fork Blocks:
570-678

Resource Fork Blocks:
679-912

Attributes:
Type: DATA (4352-0)   Name: DATA   Non-Resident   size: 444684  init_size: 444684
Type: RSRC (4353-1)   Name: RSRC   Non-Resident   size: 956251  init_size: 956251

Resources:
  Type: icns    ID: 49081       Offset: 260     Size: 955941    Name: <none>
```

Abbildung 69: istat-Ausgabe zu "alias_datei_c.txt"

Aus Abbildung 69 ist ersichtlich, dass es sich um einen Alias handelt. Es wird jedoch kein Verweis auf das referenzierte Ziel dokumentiert. Da der Pfad zur Zieldatei im Data Fork des Links gespeichert ist, kann durch Aufruf der allokierten Allocation Blocks (570-678) oder mit *icat* und den TSK-Attribut-Nummern für Data Fork der Pfad entnommen werden:

```
MacBook:Images Lars$ icat -o 40 image_5.E01 38-4352-0 | xxd -l 150
0000000: 626f 6f6b 0000 0000 6d61 726b 0000 0000  book....mark....
0000010: 3800 0000 3800 0000 d4c8 0600 0000 0410  8...8...........
0000020: 0000 0000 0000 0010 309e 4197 6957 bc41  ........0.A.iW.A
0000030: 0000 0000 756d 6573 b8c7 0600 0400 0000  ....umes........
0000040: 0303 0000 0004 0000 0700 0000 0101 0000  ................
0000050: 566f 6c75 6d65 7300 0500 0000 0101 0000  Volumes.........
0000060: 6c69 6e6b 7300 0000 0b00 0000 0101 0000  links...........
0000070: 6461 7465 695f 632e 7478 7400 0c00 0000  datei_c.txt.....
0000080: 0106 0000 1000 0000 2000 0000 3000 0000  ........ ...0...
0000090: 0800 0000 0403                           ......
```

Abbildung 70: Pfad zur Zieldatei von "alias_datei_c.txt"

Nun kann die Zieldatei „datei_c.txt“ untersucht werden.

Die forensischen Werkzeuge EnCase 7.11.01, X-Ways 18.4 und BlackLight R3.1 können automatisiert die Links und die dazugehörigen Catalog Node Einträge analysieren. FTK 6.0.1 hat diese Funktionalität nur teilweise, da die Einträge nicht vollständig ausgewertet werden.

5.5 Kategorie Anwendung

In der Kategorie Anwendung verfügt HFS Plus über zwei wesentliche Funktionen, die für die Protokollierung von Dateisystemänderungen zuständig sind: Das *Journal* und die *File System Event Database*. Beide Funktionalitäten werden vorgestellt und es wird auf die Analysetechniken eingegangen.

5.5.1 Journal

Das Journal im HFS Plus Dateisystem kann optional verwendet werden. Die meisten Mac OS X Installationen haben es per Default aktiviert. Es kann genutzt werden, um das Dateisystem nach einem Absturz, zum Beispiel durch fehlerhaftes Einbinden in das System, wiederherzustellen, ohne dass die gesamte Dateisystemstruktur überprüft werden muss. Das Journal wird nur für die Protokollierung von Volume-Strukturen und Metadaten, jedoch nicht für die Protokollierung von Dateiinhalten verwendet.[137] Wenn zum Beispiel Schreiboperationen in der Kategorie Dateiinhalt unabhängig vom Journal stattfinden, kann nicht mit den Informationen aus dem Journal die Konsistenz zwischen Metadaten und Inhalt einer Datei garantiert werden.[138]

Ein HFS Plus Journal kann auch auf einem anderen Volume „ausgelagert“ werden.

Werden Dateisystemtransaktionen erfolgreich auf das Volume geschrieben, wird das Journal bereinigt. Sollte das System abstürzen, bevor die Dateisystemtransaktionen auf das Volume geschrieben wurden, kann das Dateisystem mit den Informationen aus dem Journal in einen konsistenten Zustand wiederhergestellt werden. [139]

[137] Apple: *Technical Note TN1150*, S. 31
[138] Singh 2007, S. 1538–1539
[139] Levin 2013, S. 619

Die Funktionalität des Journals führt folgende Schritte für die Transaktionen durch:[140]

1. Die Transaktion wird durch einen Schreibvorgang von allen System-Metadaten-Veränderungen in die Journal Datei gestartet.
2. Das Journal wird im Volume aktualisiert.
3. Der Transaktionsvorgang wird im Journal Header eingetragen.
4. Die Änderungen werden an den aktuellen System-Metadaten der entsprechenden Dateien umgesetzt.
5. Der Journal Header wird aktualisiert, um die Transaktion im Journal als abgeschlossen zu kennzeichnen.

Um die Journalinformationen zu erheben, wird bei einem eingebundenen HFS Plus Volume mit aktiviertem Journal im Volume Header der Eintrag im Feld *lastMountedVersion* (zuletzt eingebundene Version) überprüft (siehe Tabelle 9). Wurde das Volume zuvor von einer Applikation mit Journal-Eigenschaft eingebunden, beinhaltet der Eintrag die Signatur *HFSJ* und noch ausstehende Transaktionen, die im Journal stehen, werden ausgeführt. Anschließend ist das Volume im System verfügbar.[141]

Das HFS Plus Dateisystem Journal beinhaltet zwei Dateien, die im Root Verzeichnis des Volumes liegen:

- .journal_info_block
- .journal

Der *.journal_info_block* beinhaltet die Größe und die Position des *Journal Headers* und *Buffers*. Folgende Tabelle zeigt den Aufbau des Journal Info Blocks:

Offset (Byte)	Größe (Byte)	Beschreibung
0	4	Flags; 1-Bit Flag-Feld
4	4	Geräte Signatur, wenn das Journal nicht im Volume selbst gespeichert ist.
8	8	Offset (Beginn des Volumes bis zum Journal Header)
16	8	Größe des Journals
24	4	Reserviert

Tabelle 36: Struktur des Journal Info Blocks

Die möglichen Werte des Bit Flag-Feldes ab Offset 0 des Journal Info Blocks können der Tabelle 37 entnommen werden:

Bit-Wert	Bezeichnung	Beschreibung
0x01	kJIJournalInFSMask	Der Journal Header und die Transaktionen sind im vom Journal überwachten Volume gespeichert. Der Offset vom Journal Info Block ist relativ zum Beginn des Volumes und die Allocation Block Nummer ist Null.

[140] Burghardt und Feldman 2008, S. 77
[141] Singh 2007, S. 1540–1541

0x02	kJIJournalOnOtherDeviceMask	Der Journal Header und die Transaktionen sind nicht im vom Journal überwachten Volume gespeichert. Der Eintrag zur Geräte Signatur im Journal Info Block enthält die Informationen zum Gerät, auf dem Journal Header und die Transaktionen gespeichert.
0x04	kJIJournalNeedInitMask	Der Journal Header ist nicht gültig. Dies ist grundsätzlich der Fall, wenn das Journal zum ersten Mal erstellt und der Speicher allokiert wird. Es existieren keine gültigen Transaktionen im Journal. Das erste Einbinden des Volumes initialisiert den Journal Header und löscht das Bit wieder.

Tabelle 37: Bit-Flag-Maske vom Journal Info Block Offset 0

Das Journal beinhaltet den Journal Header und Buffer. Der Header beinhaltet die Informationen über die Dateisystem-Transaktionen. Der Buffer speichert die Dateisystem-Transaktionen.

Folgende Abbildung zeigt, wie ein HFS Plus Volume mit Journal strukturiert ist:

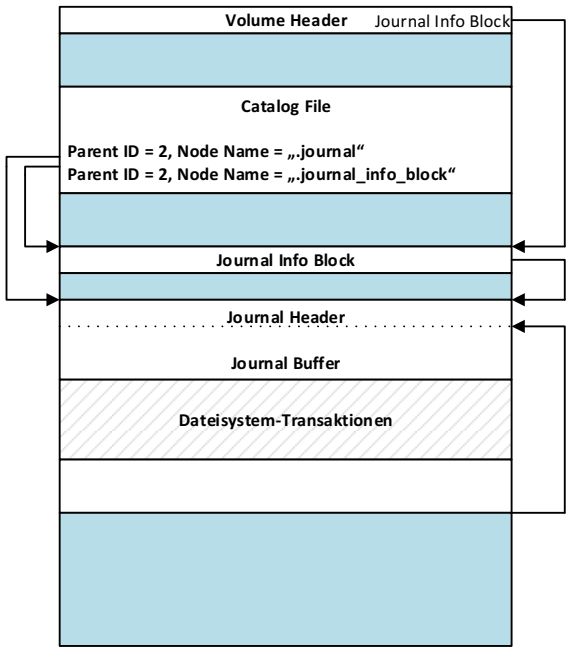


Abbildung 71: Übersicht HFS Plus Journal[142]

[142] Apple: *Technical Note TN1150*, S. 32

Der Journal Header ist wie folgt aufgebaut:

Offset (Byte)	Größe (Byte)	Beschreibung
0	4	Signatur (magic) für die Verifizierung der Integrität des Journal Headers
4	4	Endian für die Verifizierung der Integrität des Journal Headers
8	8	Start der ersten Transaktion (Offset in Bytes vom Anfang des Journal Headers zum Beginn der ersten Transaktion)
16	8	Ende der letzten Transaktion (Offset in Bytes vom Anfang des Journal Headers zum Ende der letzten Transaktion)
24	8	Größe des Journals (Beinhaltet auch Journal Header und Journal Buffer und muss die gleiche Größe betragen wie der Größen-Eintrag im Journal Info Block)
32	4	Größe des Block List Headers
36	4	Prüfsumme des Journal Headers
40	4	Größe des Journal Headers
44	Variabel	Block Info Array (Array mit Blocks variabler Größe)

Tabelle 38: Struktur des Journal Headers

Eine einzelne Dateisystem-Transaktion besteht aus Block List Header, Block List Info Einträge und dazugehörigen Daten Blocks, die Informationen darüber enthalten, welche Daten auf das Volume und auf welche Position des Volumes geschrieben werden sollen. Dies wird durch den Block List Header umgesetzt, der die Anzahl und die Größe der Blocks mit den entsprechenden Daten beschreibt. Der Block List Header ist wie folgt aufgebaut:[143]

Offset (Byte)	Größe (Byte)	Beschreibung
0	2	Maximale Anzahl der Blocks (reserviert auf dem Volume)
2	2	Anzahl Block Info
4	4	Anzahl der Bytes im Block dieser Block List des Journals (inkl. Block List Header, Block Info und Block Daten)
8	4	Prüfsumme des Block List Headers
12	4	Reserviert

Tabelle 39: Struktur des Block List Headers

[143] Apple: *Technical Note TN1150*, S. 34–35

Struktur des Block Info Arrays (Offset 2 im Block List Header):

Offset (Byte)	Größe (Byte)	Beschreibung
0	8	Nummer des Sektors, in den die Daten geschrieben werden sollen. Dieses Feld ist reserviert für das erste Element im Block Info Array.
8	4	Anzahl der Bytes, die geschrieben werden sollen
12	4	Dieses Feld hat auf dem Volume keine Bedeutung.

Tabelle 40: Struktur des Block Info Arrays

5.5.1.1 Analyse-Szenario

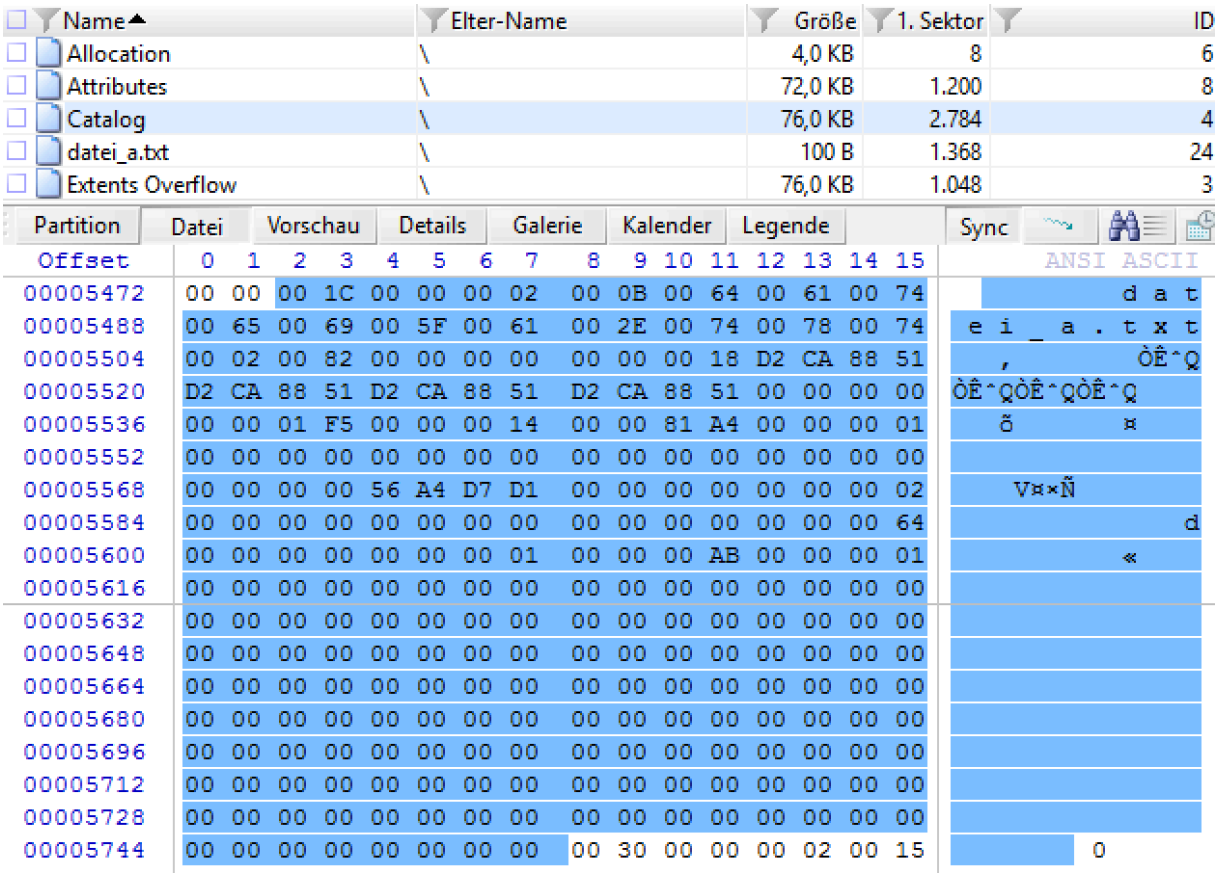
Als Szenario für eine forensische Analyse kommt eine gelöschte Datei in Betracht, bei der der Catalog Dateieintrag im *Catalog File* durch die Reorganisation des B-Trees bereits überschrieben wurde, aber die Dateisystem-Transaktion noch im Journal gespeichert ist. Es kann in diesem Fall versucht werden, den Catalog Dateieintrag im Journal zu ermitteln und so die Metadaten der gelöschten Datei und Informationen über die früher zur Datei allokierten Dateneinheiten zu gewinnen. Sofern die Dateneinheiten nicht bereits durch erneute Allokation überschrieben wurden, kann auch der Dateiinhalt wiederhergestellt werden.

Als Beispiel wurde ein Apple Diskimage mit der Bezeichnung „image_6_1“ mit HFS Plus Dateisystem erstellt. Es wurde mit dem Python Skript „data.py“ die Datei „datei_a.txt“ im Root Verzeichnis erstellt. Diese Datei hat die CNID 24 mit einem Catalog Dateieintrag im *Catalog File* ab Offset 5474 des HFS Plus Volumes.

Als Aktion wurde eine Kopie des Diskimages „image_6_1“ mit der Bezeichnung „image_6_2“ im Mac OS X 10.11 System eingebunden. „datei_a.txt“ wurde im Mac OS X Finder gelöscht, indem sie in den Papierkorb verschoben wurde, der anschließend entleert wurde. Das Diskimage wurde nicht ordnungsgemäß ausgehängt. Es wurde ein System-Crash durch Ausschalten des Mac Gerätes simuliert, so dass die Dateisystem-Transaktionen noch nicht ins HFS Plus Volume umgesetzt werden konnten.

Analyse des Basis-Images „image_6_1“

Der Catalog Dateieintrag der Datei „datei_a.txt“ befindet sich am Offset 5474 des *Catalog Files*. Die Datei hat die CNID 24 und die Parent ID 2. Sie belegt einen Allocation Block mit der Nr. 171.

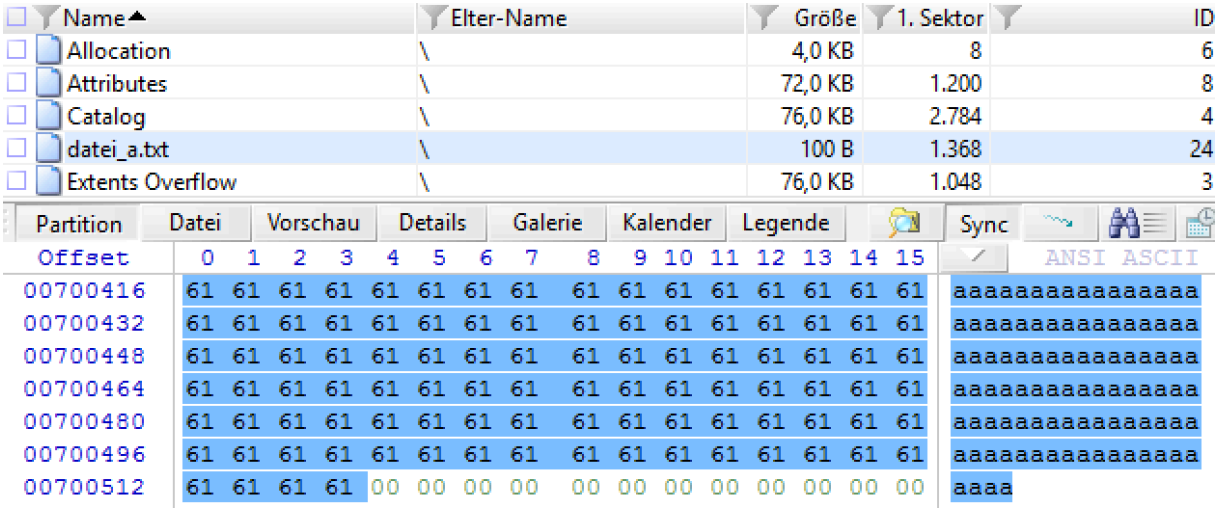


Name	Elter-Name	Größe	1. Sektor	ID
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4
datei_a.txt	\	100 B	1.368	24
Extents Overflow	\	76,0 KB	1.048	3

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00005472	00	00	00	1C	00	00	00	02	00	0B	00	64	00	61	00	74	d a t
00005488	00	65	00	69	00	5F	00	61	00	2E	00	74	00	78	00	74	e i _ a . t x t
00005504	00	02	00	82	00	00	00	00	00	00	00	18	D2	CA	88	51	, ÒÊˆQ
00005520	D2	CA	88	51	D2	CA	88	51	D2	CA	88	51	00	00	00	00	ÒÊˆQÒÊˆQÒÊˆQ
00005536	00	00	01	F5	00	00	00	14	00	00	81	A4	00	00	00	01	õ ¤
00005552	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005568	00	00	00	00	56	A4	D7	D1	00	00	00	00	00	00	00	02	V¤×Ñ
00005584	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	64	d
00005600	00	00	00	00	00	00	00	01	00	00	00	AB	00	00	00	01	«
00005616	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005632	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005648	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005664	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005680	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005696	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005712	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005728	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00005744	00	00	00	00	00	00	00	00	00	30	00	00	00	02	00	15	0

Abbildung 72: Catalog Dateieintrag zu "datei_a.txt" im Basisimage "image_6_1"



Name	Elter-Name	Größe	1. Sektor	ID
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8
Catalog	\	76,0 KB	2.784	4
datei_a.txt	\	100 B	1.368	24
Extents Overflow	\	76,0 KB	1.048	3

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00700416	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700432	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700448	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700464	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700480	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700496	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700512	61	61	61	61	00	00	00	00	00	00	00	00	00	00	00	00	aaaa

Abbildung 73: Inhalt der Datei "datei_a.txt"

Analyse des Images „image_6_2“

Nach der Löschaktion kann kein Catalog Dateieintrag zur Datei „datei_a.txt“ mit der CNID 24 im gesamten *Catalog File* festgestellt werden.

Eine Analyse des Journals ergab, dass am Offset 115554 des Journals ein Catalog Dateieintrag zur CNID 24 mit dem Dateinamen „datei_a.txt“ vorhanden ist. Dieser Eintrag konnte nicht mehr im *Catalog File* gefunden werden. Eine Überprüfung des Extent Eintrags im Catalog Dateieintrag, der im *Journal File* gefunden wurde, ergab einen belegten Allocation Block am

HFS Plus Volume Offset 700416 (Allocation Block #171). Dieser Allocation Block ist laut Allocation File nicht mehr allokiert. Er enthält aber die Inhaltsdaten, die mit denen von der Originaldatei „datei_a.txt“ aus dem Basisimage „image_6_1“ identisch sind.

Die relevanten Strukturen lassen sich wie folgt untersuchen:

Im ersten Schritt muss der Volume Header ausgewertet werden, der am Volume Offset 1024 beginnt (siehe Abschnitt Kategorie Dateisystem, Volume Header).

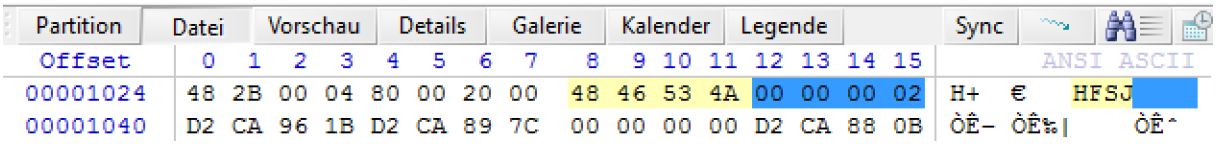


Abbildung 74: Informationen über das Journal aus dem Volume Header

Am Offset 8 des Volume Headers steht für das Feld *lastMountedVersion* (zuletzt eingebundene Version) mit vier Bytes (gelbe Markierung) HFSJ, was bedeutet, dass das Dateisystem in einem System eingebunden war, bei dem die Journal-Funktion aktiviert war.

Am Offset 12 kann die Position der Datei „.journal_info_block“ entnommen werden. Der Wert wird mit vier Bytes angegeben. Im Beispielfall befindet sich der Journal Info Block im LBA 2, was am Offset 8192 des Volumes liegt. Hier ist in den ersten vier Bytes das Flag „0x01“ für *kJIJournalInFSMask* gesetzt. Das Journal ist demnach im Volume gespeichert und folgt dem Journal Info Block ab LBA 3 am Volume Offset 12288.

Besonders bei der Interpretation des Journals zu beachten ist, dass die Werte in der Datei „.journal“ in Little Endian gespeichert werden. Im Offset 0 des Journals beginnt der Journal Header (siehe Tabelle 38).

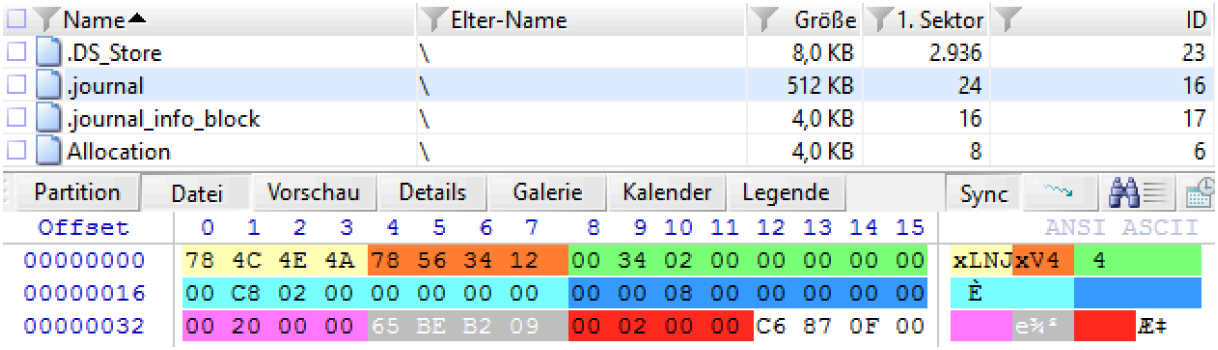


Abbildung 75: Journal Header des Journals

Die ersten vier Bytes des Journal Headers beinhalten die Signatur 0x4A4E4C78 (gelbe Markierung), gefolgt von vier Bytes für das Feld *Endian* mit dem Wert 0x12345678 (orangene Markierung). Beide Werte dienen zur Verifizierung der Integrität des Journal Headers.

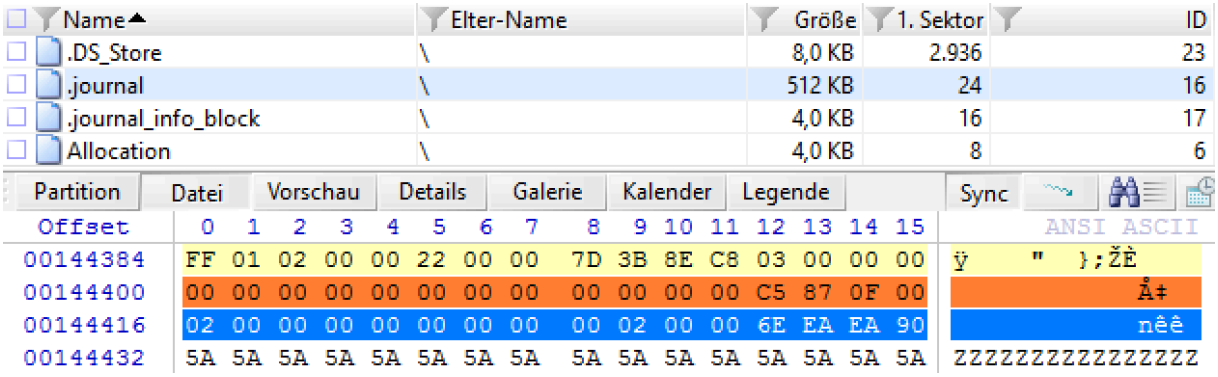
Von Offset 8-15 wird der Beginn der ersten Transaktion mit dem 64 Bit Little-Endian Integer-Wert 144384 (0x023400) angegeben (grüne Markierung). Das Ende der letzten Transaktion folgt mit dem Wert 182272 (0x02C800) (türkisfarbene Markierung). Die nächsten acht Bytes geben die Größe des Journals, die 524288 (0x080000) beträgt (blaue Markierung), an. Die Größe des *Block List Headers* beträgt 8192 Bytes (0x2000) (violette Markierung). In Grau

wurde die Prüfsumme des *Journal Headers* markiert, die den Wert 0x09B2BE65 hat. Die Größe des *Journal Headers* beträgt 512 Bytes, siehe rote Markierung (0x0200).

Im nächsten Schritt werden die Werte der ersten mit denen der letzten Transaktion verglichen, um festzustellen, ob Dateisystem-Transaktionen vorhanden sind, die noch nicht im Volume umgesetzt wurden. Dies ist der Fall, wenn die Werte nicht identisch sind. Da sie im Beispiel unterschiedlich sind, wurden die Transaktionen noch nicht umgesetzt, so dass Daten aus offenen Transaktionen im Journal erhoben werden können.

Um möglicherweise relevante Daten im Journal zu finden, müssen die einzelnen Transaktionen analysiert werden. Die Position der ersten Transaktion ist aus dem *Journal Header* bekannt und beginnt am Offset 144384 des Journals.

Wie zuvor dargestellt, besteht eine Transaktion aus Block List Header, Block Info Blocks und Block Daten.



Name	Elter-Name	Größe	1. Sektor	ID
.DS_Store	\	8,0 KB	2.936	23
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17
Allocation	\	4,0 KB	8	6

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00144384	FF	01	02	00	00	22	00	00	7D	3B	8E	C8	03	00	00	00	ÿ " };ŽÈ
00144400	00	00	00	00	00	00	00	00	00	00	00	00	C5	87	0F	00	Å‡
00144416	02	00	00	00	00	00	00	00	00	02	00	00	6E	EA	EA	90	nêê
00144432	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	5A	ZZZZZZZZZZZZZZZZ

Abbildung 76: Block Info Blocks der ersten Transaktion im Journal

Der Block List Header der ersten Transaktion ist in der Abbildung gelb markiert. Er belegt die ersten 16 Bytes der Transaktion. Folgende Daten können entnommen werden: Die ersten zwei Bytes (Offset 0-1) sind reserviert. Es folgen zwei Bytes (Offset 2-3) mit dem Wert 2 (0x02) als Anzahl der Block Info Einträge in der Transaktion. Im Offsetbereich 4-7 steht die Größe in Bytes für die gesamte Transaktion, die hier 8704 Bytes beträgt. Es folgen die Prüfsumme für den Block List Header (0xC88E3B7D) und vier reservierte Bytes.

Anschließend stehen zwei Block Info Einträge mit je 16 Bytes. Der erste mit orangener Markierung ist für die Analyse uninteressant, da diese Werte keine Bedeutung für dieses Volume haben.

Der blau markierte Block Info Eintrag gibt Auskunft über die Position, wo die Transaktionsdaten in diesem Block gespeichert sind und wie viel Speicherplatz davon belegt werden. Offsetbereich 0-7 haben den Wert 0x02 für die Position der Daten und Offsetbereich 8-11 die Anzahl der Bytes, die die Transaktion belegt, diese beträgt 512 (0x0200).

Durch Addition des Offsets (144384) der ersten Transaktion und der Größe des Block List Headers (8192) kann die Position der Block Daten der ersten Transaktion gefunden werden.

Name	Elter-Name	Größe	1. Sektor	ID
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17
Allocation	\	4,0 KB	8	6
Attributes	\	72,0 KB	1.200	8

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00152576	48	2B	00	04	80	00	20	00	48	46	53	4A	00	00	00	02	H+ € HFSJ
00152592	D2	CA	96	1B	D2	CA	88	8D	00	00	00	00	D2	CA	88	0B	ÒÊ– ÒÊˆ ÒÊˆ

Abbildung 77: Auszug der Block Daten der ersten Transaktion

Im Beispiel ist dies ein *Volume Header*. Um Transaktionen bezüglich der Datei „datei_a.txt“ zu finden, müssen die nachfolgenden Transaktionen ausgewertet werden. Durch die Werte aus dem *Journal Header* und der jeweiligen *Block List Header* und Block Info Einträge können die Positionen von der ersten bis zur letzten Transaktion bestimmt werden (in dieser Reihenfolge).

Die zweite Transaktion liegt am Offset 153088 des Journals.

Name	Elter-Name	Größe	1. Sektor	ID
.Trashes (1)	\	100 B		20
.DS_Store	\	8,0 KB	2.936	23
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00153088	FF	01	07	00	00	72	00	00	27	2D	30	3B	03	00	00	00	ÿ r '-0;
00153104	00	00	00	00	00	00	00	00	00	00	00	00	C6	87	0F	00	Æ‡
00153120	08	00	00	00	00	00	00	00	00	10	00	00	BF	BF	83	6F	¿¿fo
00153136	E8	0A	00	00	00	00	00	00	00	10	00	00	F9	0E	63	A9	è ù c©
00153152	02	00	00	00	00	00	00	00	00	02	00	00	BA	CA	74	26	°Êt&
00153168	E0	0A	00	00	00	00	00	00	00	10	00	00	22	09	B0	0F	à " °
00153184	F8	0A	00	00	00	00	00	00	00	10	00	00	DF	C0	64	FE	ø ßÀdþ
00153200	F0	0A	00	00	00	00	00	00	00	10	00	00	94	09	EF	44	ð ” ïD

Abbildung 78: Block List Header und Block Info Einträge der zweiten Transaktion

Aus dem *Block List Header* (gelbe Markierung) lässt sich entnehmen, dass sieben Block Info Einträge existieren. Der im Beispiel relevante Block Info Eintrag ist rot markiert. Die Block Daten beginnen am Offset 16588. Die Größe des Block Datenbereichs beträgt 4096 Bytes. Diese werden nach Metadaten, die zur gelöschten Datei „datei_a.txt“ gehören könnten, durchsucht.

Name	Elter-Name	Größe	1. Sektor	ID
.Trashes (1)	\	100 B		20
.DS_Store	\	8,0 KB	2.936	23
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00167184	00	62	00	06	00	00	00	1D	00	00	00	03	00	00	00	00	b
00167200	00	14	00	03	00	35	00	30	00	31	00	1C	00	00	00	1D	5 0 1
00167216	00	0B	00	64	00	61	00	74	00	65	00	69	00	5F	00	61	d a t e i _ a
00167232	00	2E	00	74	00	78	00	74	00	02	00	82	00	00	00	00	. t x t ,
00167248	00	00	00	18	D2	CA	88	51	D2	CA	88	51	D2	CA	88	51	ÒÊˆQÒÊˆQÒÊˆQ
00167264	D2	CA	88	51	00	00	00	00	00	00	01	F5	00	00	00	14	ÒÊˆQ õ
00167280	00	00	81	A4	00	00	00	01	00	00	00	00	00	00	00	00	¤
00167296	00	00	00	00	00	00	00	00	00	00	00	00	56	A4	D8	FC	V¤Øü
00167312	00	00	00	00	00	00	00	02	00	00	00	00	00	00	00	00	
00167328	00	00	00	00	00	00	00	64	00	00	00	00	00	00	00	01	d
00167344	00	00	00	AB	00	00	00	01	00	00	00	00	00	00	00	00	«
00167360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167376	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167392	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167408	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167424	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167440	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167456	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167472	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00167488	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Abbildung 79: Catalog Dateieintrag im Journal

Am Offset 167210 befindet sich ein Catalog Dateieintrag. Die Analyse der Catalog Dateieintragsstruktur (siehe Abschnitt Kategorie Metadaten, Tabellen 21-23) ergibt folgende Daten:

Parent ID ist 29. Diese CNID ist dem Papierkorb zuzuordnen. Der Name der zugehörigen Datei lautet „datei_a.txt“ mit der CNID 24. Diese Daten lassen sich der gesuchten bzw. gelöschten Datei zuordnen. Anhand der Extent Einträge im Catalog Dateieintrag können die Positionen der vor der Löschaktion mit der Datei allokierten Allocation Blocks ermittelt und analysiert werden. Gesamtzahl der Allocation Blocks beträgt 1 mit Allocation Block Nr. 171. Sucht man diesen auf, können Inhaltsdaten festgestellt werden:

Name	Elter-Name	Größe	1. Sektor	ID
.Trashes (1)	\	100 B		20
.DS_Store	\	8,0 KB	2.936	23
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00700416	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700432	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700448	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700464	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700480	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700496	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	61	aaaaaaaaaaaaaaaa
00700512	61	61	61	61	00	00	00	00	00	00	00	00	00	00	00	00	aaaa

Abbildung 80: Inhaltsdaten des Allocation Blocks 171

Ein weiterer Eintrag lässt sich im Journal außerhalb der noch nicht umgesetzten Transaktionen finden, der ebenfalls einen Catalog Dateieintrag zur Datei „datei_a.txt“ enthält. Dieser liegt im Journal im Offsetbereich 115554-115831:

Name	Elter-Name	Größe	1. Sektor	ID
.Trashes (1)	\	100 B		20
.DS_Store	\	8,0 KB	2.936	23
.journal	\	512 KB	24	16
.journal_info_block	\	4,0 KB	16	17

Partition | Datei | Vorschau | Details | Galerie | Kalender | Legende | Sync

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	ANSI ASCII
00115552	00	00	00	1C	00	00	00	02	00	0B	00	64	00	61	00	74	d a t
00115568	00	65	00	69	00	5F	00	61	00	2E	00	74	00	78	00	74	e i _ a . t x t
00115584	00	02	00	82	00	00	00	00	00	00	00	18	D2	CA	88	51	, ÒÊˆQ
00115600	D2	CA	88	51	D2	CA	88	51	D2	CA	88	51	00	00	00	00	ÒÊˆQÒÊˆQÒÊˆQ
00115616	00	00	01	F5	00	00	00	14	00	00	81	A4	00	00	00	01	õ ¤
00115632	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115648	00	00	00	00	56	A4	D7	D1	00	00	00	00	00	00	00	02	V¤×Ñ
00115664	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	64	d
00115680	00	00	00	00	00	00	00	01	00	00	00	AB	00	00	00	01	«
00115696	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115712	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115728	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115744	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115760	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115776	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115792	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115808	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00115824	00	00	00	00	00	00	00	00	00	30	00	00	00	02	00	15	0

Abbildung 81: Zweiter Catalog Dateieintrag im Journal

Aus der Catalog Dateieintragsstruktur kann entnommen werden, dass die Parent ID 2 lautet, was dem Root Verzeichnis des HFS Plus Volumes entspricht. CNID beträgt 24 und der Dateiname „datei_a.txt“, was den Metainformationen der Originaldatei entspricht. Es ist ebenfalls ein Extent Eintrag vorhanden. Es handelt sich hierbei auch um den Allocation Block 171 mit den Inhaltsdaten aus der Abbildung 80.

Im Ergebnis lassen sich aufgrund der Analyse auf Anwendungsebene Metadaten von bereits gelöschten Dateien erheben, so dass metadatenbasierend Inhaltsdaten wiederhergestellt werden können.

5.5.1.2 Analysetechniken

Aaron Burghardt und Adam J. Feldman haben 2008 eine Arbeit bezüglich der Wiederherstellung von gelöschten Dateien anhand des HFS Plus Journals veröffentlicht.[144] Die dort beschriebene Vorgehensweise für die Analyse des HFS Plus Journals für Dateiwiederherstellung wurde im oben genannten Beispiel angewandt und enthält folgende Schritte:

1. Analyse des Volume Headers
2. Position des *Catalog Files* aufsuchen anhand der Informationen aus dem Volume Header
3. Position des *Journal Files* bestimmen anhand der Informationen aus dem Volume Header
4. Journal File analysieren, beginnend von der ältesten Transaktion
5. Blocks sequentiell analysieren, um Kopien von Catalog Dateieinträgen im Journal File zu finden. Jeder interessante Eintrag muss im *Catalog File* auf Vorhandensein überprüft werden. Wenn der Eintrag aus dem Journal File nicht im *Catalog File* vorhanden ist, dann ist dies ein Hinweis, dass dieser zu einer gelöschten Datei gehört.
6. Wiederherstellungspotential ermitteln

Eine manuelle Auswertung des Journals ist sehr zeitaufwändig und bei größeren Datenmengen nicht praktikabel. Mittlerweile gibt es verschiedene Werkzeuge, die das Journal eines HFS Plus Dateisystems analysieren können.

Eine GUI-basierte Möglichkeit, das HFS Plus Journal automatisiert zu analysieren, bietet unter anderem das EnScript „HFSJournalParser“ für EnCase v7 von Kazamiya.[145] Das Skript durchsucht das Journal nach sämtlichen Transaktionen und interpretiert die gefundenen Metadaten. Die Ergebnisse werden als Lesezeichen in EnCase gespeichert. Als Option können Wiederherstellungsversuche von Dateien anhand der im Journal gefundenen Metadaten automatisiert unternommen werden.

Das Beispiel-Szenario hat gezeigt, dass Metadateninformationen gelöschter Dateien in Journal-Transaktionen gefunden werden können, auch wenn sie nicht mehr im *Catalog File* gespeichert sind. Dabei wurden noch offene Transaktionen ausgewertet, die etwa nach einem nicht ordnungsgemäßen Aushängen eines Datenträgers oder durch einen System-Crash im Journal gespeichert waren. Darüber hinaus können nicht nur offene Transaktionen im Journal von Bedeutung sein. Im Beispiel konnten auch Metadaten eines Catalog Dateieintrags aus dem Journal wiederhergestellt werden, der nicht mehr zu den gültigen Transaktionen zugehörig war.

Zu beachten ist, dass Allocation Blocks, auf die durch im Journal gefundene Metadaten verwiesen wird (Extents Einträge aus Catalog Dateieinträgen), bereits neu allokiert sein können oder zwischenzeitlich vom Zeitpunkt des Löschens bis zum Wiederherstellungsversuch mehrfach oder teilweise von neuen Dateien allokiert wurden.

[144] Burghardt und Feldman 2008
[145] http://www.kazamiya.net/en/HFSJournalParser, aufgerufen am 27.01.2016

Es ist auch die Konstellation vorstellbar, dass von einer gelöschten Datei, die stark fragmentiert war und Extents Overflow Einträge für die Allokation benötigte, nur der Catalog Dateieintrag im Journal gefunden wird und so die Datei nicht vollständig wiederhergestellt werden kann, da keine weiteren Anhaltspunkte für die übrigen Extents aus dem *Extents Overflow File* existieren.

5.5.2 File Event Store Database

Die *File Event Store Database* ist ebenfalls für die Protokollierung von Dateisystemänderungen bei HFS Plus Volumes zuständig. Die Datenbank ist auf jedem Volume, das mit einem Mac System verbunden ist, im Ordner *.fseventsd* gespeichert. Insbesondere *Spotlight* und *Time Machine* nutzen diese Datenbank, um zu ermitteln, welche Dateien neu sind oder ob Dateien veränderte Metadateneinstellungen haben.[146]

5.5.2.1 Analyse-Szenario

Als Beispiel dient das zuvor verwendete Image *„image_6_2“*. Ziel ist es, Informationen über die gelöschte Datei „datei_a.txt“ zu ermitteln, die von der *File Event Store Database* gespeichert wurden.

5.5.2.2 Analysetechniken

Die Datei *fseventsd-uuid* enthält den GUID der FSEvent Datenbank. Während auf einem „internen“ Mac System-Volume die UUID persistent ist, um vor Fehlfunktion zu schützen, ändern sich die FSEvent-UUIDs von externen Volumes bei jedem Einbinden ins System (Dies kann im System.log überprüft werden, wenn z. B. nach „fsevents“ gefiltert wird.). Auf nicht-HFS Plus Volumes wird der *.fsevents*-Ordner ebenfalls vom Mac System erstellt, aber ohne FSEvents Datenbank.

```
sh-3.2# ls -ilAr
total 40
22 -rw-------  1 Lars  staff   36 24 Jan 14:56 fseventsd-uuid
28 -rw-------  1 Lars  staff   70 24 Jan 14:56 000000000000e87b
27 -rw-------  1 Lars  staff   60 24 Jan 14:56 000000000000e87a
26 -rw-------  1 Lars  staff   70 24 Jan 14:55 000000000000d3c8
25 -rw-------  1 Lars  staff  113 24 Jan 14:55 000000000000d3c7
```

Abbildung 82: Inhalt des Verzeichnisses „.fsevent“ von „image_6_2“

Der GUID lautet im Beispiel:

```
sh-3.2# cat fseventsd-uuid
A408B7F3-5814-4BFC-95F1-35DDA719F2E9sh-3.2#
```

Abbildung 83: fseventsd-uuid von "image_6_2"

Der Ordner „.fsevent“ im Root Verzeichnis des Volumes beinhaltet Dateien, die gzipped sind und nur mit Root-Rechten entpackt werden können. Ein Überblick kann mit *#file ** gewonnen werden:

```
sh-3.2# file *
000000000000d3c7: gzip compressed data, from Unix
000000000000d3c8: gzip compressed data, from Unix
000000000000e87a: gzip compressed data, from Unix
000000000000e87b: gzip compressed data, from Unix
fseventsd-uuid:   ASCII text, with no line terminators
```

Abbildung 84: Inhalt von „.fsevent“ mit Information über die Komprimierung

[146] Apple: *FSEvents Reference* 2012

Um die Datenbankdateien auswerten zu können, müssen diese expandiert werden. Dies kann mit dem nativen Mac-Terminal-Programm *gunzip* oder ähnlichen Tools von Drittanbietern umgesetzt werden. Im folgenden Beispiel wurden die Archive mit *unar* in ein Zielverzeichnis extrahiert.

```
sh-3.2# find . -d 1 -iname '00*' -exec /Users/Lars/Desktop/unar/unar -o /Users/Lars/Desktop/target/
{} \;
./000000000000d3c7: Gzip
  000000000000d3c7... OK.
Successfully extracted to "/Users/Lars/Desktop/target/000000000000d3c7".
./000000000000d3c8: Gzip
  000000000000d3c8... OK.
Successfully extracted to "/Users/Lars/Desktop/target/000000000000d3c8".
./000000000000e87a: Gzip
  000000000000e87a... OK.
Successfully extracted to "/Users/Lars/Desktop/target/000000000000e87a".
./000000000000e87b: Gzip
  000000000000e87b... OK.
Successfully extracted to "/Users/Lars/Desktop/target/000000000000e87b".
```

Abbildung 85: Extraktion der FSEvent-Dateien

Der Inhalt kann nun für jede expandierte *fsevent*-Datei ausgewertet werden:

```
sh-3.2# xxd 000000000000d3c7
0000000: 3153 4c44 fbc6 848c 7800 0000 00dc 7400  1SLD....x.....t.
0000010: 0000 0000 0000 0000 022e 4453 5f53 746f  ..........DS_Sto
0000020: 7265 0058 8a00 0000 0000 0055 0080 002e  re.X.......U....
0000030: 5472 6173 6865 7300 7e74 0000 0000 0000  Trashes.~t......
0000040: 4800 0001 2e54 7261 7368 6573 2e41 4b75  H....Trashes.AKu
0000050: 7965 6b00 7a74 0000 0000 0000 8801 0001  yek.zt..........
0000060: 6461 7465 695f 612e 7478 7400 c6d3 0000  datei_a.txt.....
0000070: 0000 0000 1100 8000                      ........
```

Abbildung 86: Analyse des extrahierten Inhalts

Im Beispiel konnte ein Hinweis zur gelöschten Datei „datei_a.txt“ festgestellt werden, der mit den Informationen aus dem Journal korrespondiert.

Alternativ kann auch im Mac-Terminal mit *gzcat* der Inhalt der Datenbankdateien, ohne diese zu entpacken, ausgewertet werden.

Die im Rahmen der Arbeit getesteten Forensik-Tools EnCase 7.11.01, X-Ways Forensics 18.4, FTK 6.01 und BlackLight R3.1 konnten die Datenbankdateien expandieren.

5.6 „The Big Picture“

Im folgenden Abschnitt soll ein Überblick über den Zusammenhang der in der Arbeit behandelten Dateisystemkomponenten bei einer Dateierstellung und beim Löschen einer Datei gezeigt werden.

Erstellen einer Datei:

- Das Betriebssystem analysiert den Volume Header, um die nächste verfügbare CNID zu ermitteln.
- Danach wird das Allocation File darauf überprüft, wie viele zusammenhängende Allocation Blocks, die noch nicht allokiert sind, verfügbar sind. Dies geschieht basierend auf die Allocation Block Größe im Dateisystem, die Größe der Clumps und die Größe der Datei, die erstellt wird.

- Die Datei wird allokiert.
- Das Catalog File wird aktualisiert und die Datei erhält einen Catalog Dateieintrag. Der Catalog Dateieintrag enthält grundlegende Metadaten, wie Zeitstempel und Zugriffsrechte sowie Einträge, die die mit der Datei allokierten Dateneinheiten beschreiben und gegebenenfalls Metadaten, die in Resource Fork Daten gespeichert sind.
- Sollte die Datei stark fragmentiert sein und mehr als acht Extents benötigen, werden zusätzliche Extent Einträge im Extents Overflow File erstellt.
- Zusätzliche Metadaten werden im Attributes File gespeichert. Diese enthalten Informationen zum Beispiel von Drittherstellern oder weitere Sicherheits-informationen.

Wenn eine Datei gelöscht wird:

- Das Catalog File aktualisiert sich und der Catalog Dateieintrag der Datei wird gelöscht.
- Das Allocation File wird aktualisiert und markiert die Allocation Blocks der gelöschten Datei als verfügbar.
- Sollten zusätzliche Extent Einträge im Extents Overflow File vorhanden sein, werden diese ebenfalls durch Aktualisierung des Extents Overflow Files gelöscht.
- Zusätzliche Metadaten im Attributes File werden ebenfalls durch Aktualisierung dieses B-Trees gelöscht.
- Die Daten (Data Fork und Resource Fork) sind in den nicht mehr allokierten Allocation Blocks gespeichert, bis sie überschrieben werden.
- Der Volume Header wird aktualisiert.

Anhand eines Beispiels soll veranschaulicht werden, was bei einem Löschvorgang im HFS Plus Dateisystem passiert und welche Daten „zurückbleiben“.

Als Basis dient das Image „image_3_2“. Wie aus den Analyse-Szenarien bereits bekannt, sind im Root Verzeichnis die zwei Dateien „datei_a.txt“ und „datei_b.txt“ gespeichert. „datei_a.txt“ hat die CNID 24 und belegt laut Catalog File und Extents Overflow File insgesamt 12 Extents. Sie hat auch drei zusätzliche Attribute im Attributes File gespeichert. Der Catalog Dateieintrag mit Key-Struktur beginnt am Offset 9602 des Catalog Files. Der Eintrag im Extents Overflow File mit den zusätzlichen Extents Einträgen beginnt am Offset 4110 des Extents Overflow Files. Eine Übersicht über alle Extents der Datei „datei_a.txt“ kann der Tabelle 29 aus Abschnitt 5.3.3 entnommen werden. Die drei zusätzlichen Attribut Einträge beginnen ab Offset 8206 im Attributes File (siehe Analyse-Szenario im Abschnitt 5.3.4).

Die Datei „datei_a.txt“ wurde im Finder sofort gelöscht. Die Daten können im Beispielimage „image_3_3“ analysiert werden.

Eine Analyse des Catalog Files ergab, dass der Catalog Dateieintrag mit der CNID 24 sofort gelöscht wurde und ab Offset 9602 des Catalog Files nun der Catalog Dateieintrag der Datei „datei_b.txt“ beginnt.

Das Extents Overflow File wurde ebenfalls sofort aktualisiert. Eine Analyse des B-Tree Headers ergab, dass keine weiteren Nodes außer dem Header Node in diesem B-Tree existieren. Am

ursprünglichen Offset 4110 des Extents Overflow File Eintrages zu „datei_a.txt“ sind keine Daten gespeichert („nur noch Nullen“).

Auch das Attributes File wurde aktualisiert. Es befinden sich keine Attribut Daten der gelöschten Datei in diesem B-Tree. Am Offset 8206 beginnen nun die Attribut Einträge der Datei „datei_b.txt“.

Eine Überprüfung des Allokationsstatus im Allocation File ergab, dass alle Allocation Blocks, die in den ehemaligen Extents Einträgen der Datei „datei_a.txt“ aus „image_3_2“ beschrieben wurden, als nicht-allokiert markiert sind. Die Inhaltsdaten der Datei „datei_a.txt“ sind aber noch vorhanden.

Es wurde anschließend das Dateisystem Journal analysiert. Es sind mehrere Catalog Node Einträge zur Datei „datei_a.txt“ als nicht mehr gültige Transaktionen vorhanden. Da das Image „image_3_3“ ordnungsgemäß aus dem Mac System ausgehängt wurde, sind keine Transaktionen offen. Bestätigt wird dies durch Auswertung des Journal Headers, da hier die Werte für die erste und letzte Transaktion gleich sind.

6. Fazit

Ziel dieser Arbeit war es, das Referenzmodell der Dateisystemkategorien von Brian Carrier auf HFS Plus Dateisystemstrukturen anzuwenden. Zu diesem Zweck wurden die Dateisystemstrukturen von HFS Plus im Rahmen der Kategorien von Carrier behandelt und Analyse-Szenarien entwickelt, die forensisch analysiert wurden.

Im Ergebnis wurde festgestellt, dass das Referenzmodell von Carrier auf HFS Plus Dateisystemstrukturen anwendbar ist.

Es hat sich gezeigt, dass für die forensische Analyse die Metadaten einer Datei von besonderem Interesse sind. Diese werden in verschiedenen Dateien mit einer jeweils komplexen B-Tree-Struktur gespeichert und können unter Umständen teilweise in Resource Fork Daten abgelegt sein.

Was das Löschen von Dateien betrifft, so konnte anhand von Analysen der Szenarien festgestellt werden, dass eine metadatenbasierte Wiederherstellung von gelöschten Dateien in der Regel nicht möglich ist. Dies hängt mit der sofortigen Aktualisierung von B-Tree-Strukturen der für die Speicherung von Metadaten einer Datei verantwortlichen *Special Files* zusammen. Hierbei wurden in den Beispiel-Szenarien die Metadateneinträge der gelöschten Dateien sofort überschrieben.

Herausragende Bedeutung hat im Zusammenhang mit der Wiederherstellung von gelöschten Dateien die Analyse des HFS Plus Journals in der Kategorie Anwendung. So konnten im Journal Catalog Dateieinträge gelöschter Dateien sowohl in den gültigen als auch in den nicht mehr gültigen Dateisystemtransaktionen des Journals gefunden werden. Anhand dieser Metadaten konnte die gelöschte Datei wiederhergestellt werden. Grenzen bilden dabei die erneute Allokation der von der Datei früher belegten Allocation Blocks und eine starke Fragmentierung der Datei.

Abschließend kann festgestellt werden, dass dem Forensiker technische Mittel aus dem Open Source Bereich sowie auch kommerzielle Forensik-Programme für die Analyse von HFS Plus Dateisystemartefakten zur Verfügung stehen. Hervorzuheben sind hier TSK von Brian Carrier und EnCase 7.11.01 von Guidance Software, die hier präzise und umfängliche Analyseergebnisse lieferten.

Anhang

A.1 Python Skript

```
"""
Dieses Python Skript erstellt oder veraendert Dateien durch Uebergabe der Argumente fuer Dateiname, Zeichen
    und Dateigroesse.
"""
import sys

name = sys.argv[1] #Pfad- und Dateiname
char = sys.argv[2][0] #Zeichen
size = int(sys.argv[3]) #Dateigroesse
try: #Ausnahmebehandlung
    f = open(name, 'w')
except IOError:
    print ('Konnte Datei {0}nicht oeffnen!'.format(name))
else:
    f.write(char * size) #Schreibe in Datei x das Zeichen y bis die logische Dateigroesse z erreicht ist
    f.close()
```

A.2 Images-Übersicht

Bezeichnung	MD5	Beschreibung	Abschnitt
image_1	cf95c9b190f7152f252997d0773af1a2	Apple Partition Map	3.1.2
image_2	7a7d34c00bf71bf1cc018f103fb919ef	GPT Partition Map	3.1.4 5.1.5 5.2.2 5.2.3
image_3_1	d67e5b714695ced06764d90de9ff8c9d	Basisimage 3 – Fragmentierungs-beispiel	
image_3_2	802fbbc33c71063361f373665b8561b3	Basisimage 3 modifiziert durch Fragmentierung von „datei_a.txt“	5.3.2.6 5.3.3.3 5.3.4.1 5.3.5
image_3_3	8332999b93abaa0e15543beb971c36c1	Basisimage 3 modifiziert durch Löschen von „datei_a.txt“	5.6
image_4	d139838372afbd298906b8b789a1553d	HFS Komprimierung	5.3.6
image_5	5e278bac4308d7097574474272b9f0e0	Link Files	5.4.2
image_6_1	a17d50e19fb1e079ab43feb97cd3c36b	Journal – Basisimage 6	5.5.1.1
image_6_2	7b4662cc2cd5c42c34a3533c87e13393	Journal – modifiziert durch Löschen von „datei_a.txt“	5.5.1.1 5.5.2.1 5.5.2.2

A.3 Herstellung der Fragmentierung „image_3_2“

Schritt	Dateibezeichnung	Aktion	Größe nach Aktion in Bytes
1	datei_**a**.txt	erstellen	100.000
1	datei_**b**.txt	erstellen	6.000.000
2	datei_**a**.txt	vergrößern	3.100.000
3	datei_**b**.txt	verkleinern	5.975.424
4	datei_**c**.txt	erstellen	4096
4	datei_**d**.txt	erstellen	4096
4	datei_**e**.txt	erstellen	4096
4	datei_**f**.txt	erstellen	4096
4	datei_**g**.txt	erstellen	4096
4	datei_**h**.txt	erstellen	4096
5	datei_**c**.txt	löschen	-
5	datei_**a**.txt	vergrößern	3.104.096
6	datei_**h**.txt	löschen	-
6	datei_**a**.txt	vergrößern	3.108.192
7	datei_**d**.txt	löschen	-
7	datei_**a**.txt	vergrößern	3.112.288
8	datei_**g**.txt	löschen	-
8	datei_**a**.txt	vergrößern	3.116.384
9	datei_**e**.txt	löschen	-
9	datei_**a**.txt	vergrößern	3.120.480
10	datei_**f**.txt	löschen	-
10	datei_**a**.txt	vergrößern	3.124.576

Literaturverzeichnis

Apple: *FSEvents Reference* (2012). http://developer.apple.com/library/mac/documentation/Darwin/Reference/FSEvents_Ref/Reference/reference.html; zuletzt aktualisiert am 23.07.2012, zuletzt geprüft am 29.12.2015.

Apple: *hfs_format.h* (2012). http://www.opensource.apple.com/source/xnu/xnu-2050.18.24/bsd/hfs/hfs_format.h; zuletzt aktualisiert am 28.09.2012, zuletzt geprüft am 14.06.2015.

Apple: *IOApplePartitionScheme.h*. http://www.opensource.apple.com/source/IOStorageFamily/IOStorageFamily-24/IOApplePartitionScheme.h; zuletzt geprüft am 09.09.2015.

Apple: *Technical Note TN1150 - HFS Plus Volume Format*. https://developer.apple.com/legacy/library/technotes/tn/tn1150.html#//apple_ref/doc/uid/DTS10002989; zuletzt geprüft am 03.09.2015.

Apple: *Technical Note TN2166 - Secrets of the GPT* (2006). https://developer.apple.com/library/mac/technotes/tn2166/_index.html#//apple_ref/doc/uid/DTS10003927-CH1-SUBSECTION8; zuletzt aktualisiert am 11.06.2006, zuletzt geprüft am 03.09.2015.

Burghardt, Aaron; Feldman, Adam J.: *Using the HFS+ journal for deleted file recovery* (2008). In: *Digital Investigation* 5, S. S76-S82. DOI: 10.1016/j.diin.2008.05.013.

Carrier, Brian: *File System Forensic Analysis* (2005). 10. Aufl. Boston, Mass., London: Addison-Wesley.

Dewald, Andreas; Freiling, Felix C.: *Forensische Informatik* (2011). 1. Aufl. Norderstedt: Books on Demand.

Freiling, Felix; Gruhn Michael: What is essential data in digital forensic analysis? In: 9th International Conference on IT Security Incident Management & IT Forensics (2015). Los Alamitos, CA: IEEE Computer Society Conference Publishing Services (CPS).

Geschonneck, Alexander; Kraus, Helmut Computer-Forensik: *Computerstraftaten erkennen, ermitteln, aufklären* (2014). 6. aktualisierte und erweiterte Auflage. Hg. v. René Schönfeldt. Heidelberg, Germany: dpunkt.verlag (iX-Edition). http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10852943.

Unified Extensible Firmware Interface Forum: UEFI. http://www.uefi.org; zuletzt geprüft am 11.09.2015.

Intel: *UEFI*. http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-homepage-general-technology.html; zuletzt geprüft am 11.09.2015.

iohead: FileXray: *User Guide and Reference;* zuletzt geprüft am 14.06.2015.

Kuhlee, Lorenz; Völzow, Victor: *Computerforensik Hacks* (2012). Beijing: O'Reilly (Hacks series).

Levin, Jonathan: *Mac OS X and iOS Internals - To the Apple's Core* (2013). Indianapolis, Ind.: Wiley Wrox.

Pehnack, Andreas: *Synalize IT! Pro*.
https://www.synalysis.net; zuletzt geprüft am 12.11.2015.

Singh, Amit: *Mac OS X Internals - A Systems Approach* (2007). Upper Saddle River, NJ: Addison-Wesley. http://proquest.tech.safaribooksonline.de/0321278542.

Sleuthkit Wiki: *HFS*.
http://wiki.sleuthkit.org/index.php?title=HFS; zuletzt geprüft am 22.01.2016.

Tanenbaum, Andrew S.: *Moderne Betriebssysteme* (2009). 3. Aufl. Pearson Studio.

Varsalone, Jesse; Kubasiak, Ryan R.; Barr, Walter: *Macintosh OS X, iPod, and iPhone Forensic* (2009). Burlington, Mass.: Syngress.

Wikipedia: *Apple Partition Map*.
https://en.wikipedia.org/wiki/Apple_Partition_Map; zuletzt geprüft am 09.09.2015.

Wikipedia: *Hierarchical File System*.
https://en.wikipedia.org/wiki/Hierarchical_File_System, zuletzt geprüft am 08.11.2015.

Wikipedia: *Macintosh II*.
https://en.wikipedia.org/wiki/Macintosh_II; zuletzt geprüft am 09.09.2015.

Wikipedia: *Partition (Datenträger)*.
https://de.wikipedia.org/wiki/Partition_(Datenträger)#Bezeichnungen_und_Typen_unterschiedlicher_Partitionen; zuletzt geprüft am 08.09.2015.