



Hochschule
Augsburg University of
Applied Sciences

FORENSISCHE ANALYSE EINES EXT3 DATEISYSTEMS

MASTER INDUSTRIAL SECURITY
UNIVERSITY OF APPLIED SCIENCES AUGSBURG

Tabelle 1: Student

Student	Matrikelnummer
Michael Wager	2081894

Zusammenfassung

Dieses Dokument befasst sich mit dem Linux-Dateisystem EXT3 und wurde im Rahmen einer Seminararbeit für Masterstudierende im Fach *Einführung in die IT Forensik* erstellt. Es ist grob in zwei Teile aufgeteilt: Im ersten Teil wird einleitend auf die Geschichte von Dateisystemen unter Linux eingegangen und anschließend die grundlegenden Datenstrukturen sowie die Funktionsweise des EXT3 Dateisystems beschrieben. Im zweiten Teil beginnt dann die technische Analyse, welche forensische Konzepte zur Auswertung des Dateisystems vorstellt, sowie verschiedene Ansätze zur Wiederherstellung gelöschter Dateien untersucht. Zur Veranschaulichung der technischen Hintergründe beinhalten beide Teile praktische Beispiele, welche der Leser bei Bedarf auf einem gängigen Linux-System wie z.B. Ubuntu oder Tsurugi einfach nachvollziehen kann.

Inhaltsverzeichnis

1. Technische Hintergründe	1
1.1. Geschichte des Ext-Dateisystems	1
1.2. Basics	1
1.2.1. Superblöcke	2
1.2.2. Block Group Descriptor Tables	3
1.2.3. Block- und Inode-Bitmaps	3
1.2.4. Die Inode-Tabelle	4
1.2.5. Datenblöcke	4
1.3. Praktische Beispiele	4
1.3.1. Erstellen von Dateien	5
1.3.2. Löschen von Dateien	7
2. Forensische Analyse des EXT3 Dateisystems	8
2.1. Erstellen einer forensischen Arbeitskopie	8
2.2. Forensische Konzepte zur Auswertung	8
2.3. Wiederherstellen gelöschter Dateien	10
2.3.1. File-Carving	10
2.3.2. Das File-Journal	11
3. Abbreviations	13
4. Literatur	14
A. Appendix: Script zum Auslesen des Inhalts von Datenblöcken (JavaScript)	15

1. Technische Hintergründe

1.1. Geschichte des Ext-Dateisystems

Das sogenannte Extended file system (EXT) war das Erste einer Reihe von Dateisystemen welches speziell für Linux entwickelt wurde und damals das minix-Dateisystem ablöste. EXT in Version 1 ist mittlerweile allerdings veraltet und wird in aktuellen Linux-Distributionen nicht mehr verwendet. Ext3 ist die neuere Variante, bleibt von Grund auf jedoch exakt gleich wie Ext2, führt jedoch *file system journaling* ein. Die folgenden Varianten sind aktuell noch gängig und werden aktiv eingesetzt:

- ext2 - Führte separate Zeitstempel für Dateizugriffe, Inode- und Datenmodifikation ein. Bringt keine Unterstützung für journaling.
- ext3 - Führte journaling ein (und ist required!)
- ext4 - Aktuelle Version. Unterstützung für fast fsck, native filesystem encryption, journaling, jedoch auch für non-journaling

1.2. Basics

Das Ext-Dateisystem verwendet Blöcke als Basiseinheit zum Speichern von Daten [1], ähnlich wie Cluster in FAT- und NTFS-Dateisystemen. Ein Block kann 1024, 2048 oder 4096 Bytes groß sein. Sogenannte *Inodes* (Index-Nodes) werden zum Speichern von Metadaten verwendet, Blockgruppen zur optimierten logischen Strukturierung, Verzeichnisse um hierarchische Strukturen darstellen zu können, Block- und Inode-Bitmaps um allokierte Blöcke und Inodes zu kennzeichnen, sowie Superblöcke um allgemeine Informationen des Dateisystems zu speichern. Kopien von wichtigen Datenstrukturen wie z.B. den Superblöcken werden redundant vorgehalten und alle Daten welche mit Dateien assoziiert werden, sind aus Gründen der Performance gut lokalisiert [2].

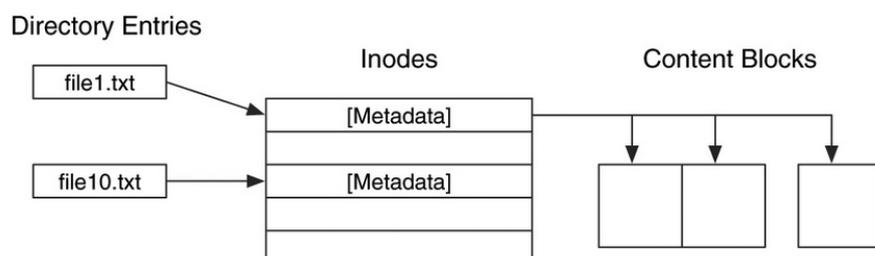


Abbildung 1: Zusammenhang zwischen Verzeichniseinträgen, Inodes und Inhaltsblöcken [2]

Abbildung 1 zeigt den Zusammenhang zwischen Verzeichniseinträgen, Inodes und Inhaltsblöcken. Für jede Datei (und für jedes Verzeichnis) existiert eine Inode, welche dazugehörige Metadaten, sowie Referenzen auf die Blöcke mit dem tatsächlichen Inhalt speichert.

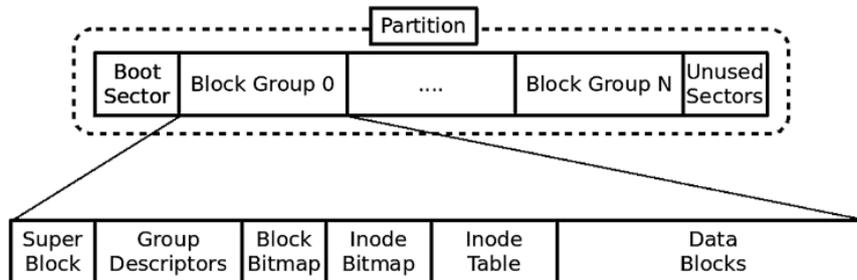


Abbildung 2: Layout des Dateisystems [3]

Abbildung 2 enthält eine graphische Darstellung des Layouts eines Ext-Dateisystems. Die Partition ist in Blockgruppen aufgeteilt und jede Blockgruppe enthält einen Superblock (optional), Group-Descriptors, die Block- und Inode-Bitmap, die Inode-Tabelle sowie die Datenblöcke. Auf diese Aufteilung innerhalb der Blockgruppen wird nun in den folgenden Unterkapiteln näher eingegangen.

1.2.1. Superblöcke

Da wir in den praktischen Beispielen (s.u.) ein Ext3 System auf einem USB Stick erstellt haben und dieser keinen OS Kernel enthält, befindet sich auf unserem Stick auch kein Boot-Code. Dieser würde sich sonst in den ersten 1024 Bytes befinden. Der *Superblock* befindet sich ab Offset 1024 des Dateisystems und hat eine Größe von 1024 Bytes, obwohl nicht alle davon verwendet werden. Hier könnte man somit leicht Daten verstecken. Er enthält Basisinformation wie die Blockgröße (z.B. 4096 Byte), die totale Anzahl an Blöcken, die Anzahl an Blöcken pro Gruppe, die Anzahl an Inodes per Blockgruppe, den Volume-Name, den letzten Schreibzugriff, die letzte Mountzeit sowie den Pfad in dem das Dateisystem zuletzt gemounted war. Abbildung 3 zeigt eine simple Analyse des Superblocks auf unserem USB Stick.

```

$ sudo dd if=/dev/sdb status=none bs=1024 count=1 skip=1 | hexdump -Cv
00000000  50 51 1d 00 00 30 75 00 00 dc 05 00 e3 ac 72 00 |PQ...0u.....r.|
00000010  45 51 1d 00 00 00 00 00 02 00 00 00 02 00 00 00 |EQ.....|
00000020  00 80 00 00 00 80 00 00 f0 1f 00 00 62 18 e9 61 |.....b..a|
00000030  62 18 e9 61 01 00 ff ff 53 ef 01 00 01 00 00 00 |b..a...S.....|
00000040  8b 0d e9 61 00 00 00 00 00 00 00 00 01 00 00 00 |...a.....|
00000050  00 00 00 00 0b 00 00 00 00 01 00 00 3c 00 00 00 |.....<...|
00000060  06 00 00 00 03 00 00 00 55 b0 dc 54 5f 8a 41 21 |.....U..T_.A|
00000070  9e 62 14 36 74 7b 65 4e 00 00 00 00 00 00 00 00 |.b.6t{eN.....|
00000080  00 00 00 00 00 00 00 00 2f 6d 65 64 69 61 2f 75 |...../media/u|
00000090  73 62 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |sb.....|
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 fe 03 |.....|
000000d0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000000e0  08 00 00 00 00 00 00 00 00 00 00 00 6a 17 9f 1e |.....j...|
000000f0  b0 3a 4c da af 04 c2 69 7b 76 59 f6 01 01 00 00 |.:L...i{vY....|
00000100  0c 00 00 00 00 00 00 00 8b 0d e9 61 08 06 00 00 |.....a....|
00000110  09 06 00 00 0a 06 00 00 0b 06 00 00 0c 06 00 00 |.....|
00000120  0d 06 00 00 0e 06 00 00 0f 06 00 00 10 06 00 00 |.....|
00000130  11 06 00 00 12 06 00 00 13 06 00 00 14 06 00 00 |.....|
00000140  15 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 08 |.....|
00000150  00 00 00 00 00 00 00 00 00 00 00 00 20 00 20 00 |.....|
00000160  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
00000170  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|

```

Abbildung 3: Analyse des Superblocks

Die ersten 4 Bytes repräsentieren die Anzahl an Inodes: [1]:

50 51 1d 00 -> Little endian -> 00 1D 51 50 ergibt 1921360 Inodes(Vergleiche mit Abbildung 4).

Die zweiten 4 Bytes repräsentieren die Anzahl an Blöcken: **00 30 75 00 => Little endian => 00 75 30 00 = ergibt 7680000 Blöcke.**

In der Abbildung ist außerdem noch der Pfad, in welchem das Dateisystem zuletzt gemounted war, markiert: **/media/usb**.

1.2.2. Block Group Descriptor Tables

Im Block nach dem Superblock folgt die Group-Descriptor-Tabelle. Diese beschreibt wo sich Daten wie z.B. die Inode-Tabelle oder die Bitmaps innerhalb einer Blockgruppe befinden.

1.2.3. Block- und Inode-Bitmaps

Die Block- und Inode-Bitmaps speichern den Allokationsstatus von Blöcken, sowie Inodes innerhalb der Blockgruppe. Sie geben also an, welche Blöcke- bzw Inodes innerhalb der Gruppe belegt sind und welche nicht. Jedes Bit repräsentiert hier den Status eines Blocks wobei "1"bedeutet, der Block ist verwendet, "0"bedeutet der Block ist frei bzw. verfügbar. Die Inode-Bitmap funktioniert ähnlich, mit dem Unterschied dass die Bits hier auf den Allokationsstatus einer Inode-Tabelle verweisen.

1.2.4. Die Inode-Tabelle

Die Inode-Tabelle wird verwendet um jedes Verzeichnis, jede Datei, symbolischen Link oder Spezialdatei zu beschreiben. Die folgenden Metadaten werden u.a. in den Inodes gespeichert:

- Dateigröße
- Speicherort (sog. Blockpointer)
- Eigentümer (Linux User-Ids)
- Gruppe (Linux Group-Ids)
- Dateityp
- Berechtigungen
- Zugriffs- und Änderungs- und Löszeitpunkt (Gespeichert als typische UNIX-Zeitstempel, also die Nummer an Sekunden seit dem 1. Januar 1970 UTC)

Der Dateiname wird nicht in Inodes gespeichert. Eine Datei kann ein Socket, ein Puffer, ein symbolischer Link oder eine normale Datei sein.

In einer Inode befinden sich vordefinierte Datenstrukturen um auf die ersten 12 Blöcke zu referenzieren, welche die Daten der Datei enthalten (*Blockpointer*). Außerdem gibt es weitere Pointer auf Blöcke, die bei größerem Dateinhalt (> 48KB bei Blocksize 4096) dann einfach weitere Pointer zu weiteren Datenblöcken enthalten.

1.2.5. Datenblöcke

Als letztes innerhalb einer Blockgruppe folgen die Datenblöcke, dessen Adressen innerhalb der Inodes gespeichert werden. Hier befinden sich die tatsächlichen Daten.

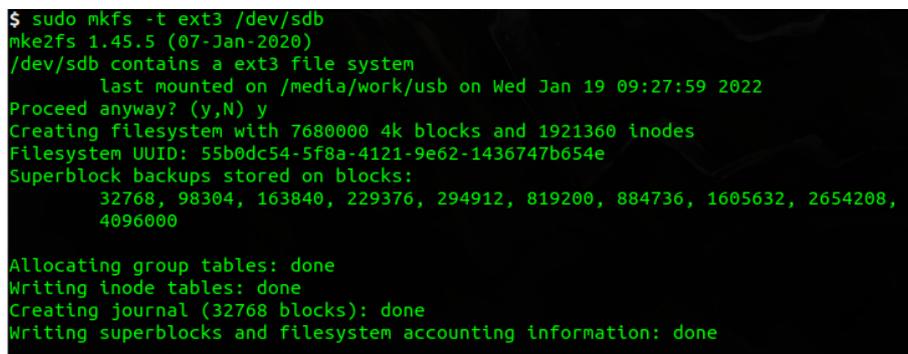
1.3. Praktische Beispiele

Alle praktischen Beispiele zu diesem Dokument wurden unter der Tsurugi Linux-Distribution (Version 5.14.6) auf einem 32GB USB Stick durchgeführt und hier völlig transparent, je nach Kontext, präsentiert und sollten somit unter allen gängigen Linuxsystemen 1-zu-1 nachgemacht werden können.

```
1 $ uname -a
2 Linux lab 5.14.6-tsurugi #1 SMP Mon Sep 20 21:45:06 CEST
   2021 x86_64 x86_64 x86_64 GNU/Li
```

Als erstes wurde der USB Stick mit dem Ext3 Dateisystem frisch formatiert:

```
1 # find mounted devices and their paths:
2 lsblk
3
4 # unmount the device: (The usb stick here is "/dev/sdb")
5 sudo umount /dev/sdb
6
7 # Create the filesystem:
8 sudo mkfs -t ext3 /dev/sdb
```



```
$ sudo mkfs -t ext3 /dev/sdb
mke2fs 1.45.5 (07-Jan-2020)
/dev/sdb contains a ext3 file system
   last mounted on /media/work/usb on Wed Jan 19 09:27:59 2022
Proceed anyway? (y,N) y
Creating filesystem with 7680000 4k blocks and 1921360 inodes
Filesystem UUID: 55b0dc54-5f8a-4121-9e62-1436747b654e
Superblock backups stored on blocks:
   32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
   4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

Abbildung 4: Erstellen eines EXT3 Dateisystems auf einem USB Stick

Im Abbildung 4 ist zu sehen wie das Dateisystem auf dem USB Stick unter `/dev/sdc` erstellt wird. Es werden 7680000 4KB Blöcke mit 1921360 Inodes erstellt (32GB). Für das Dateisystem wird eine Universal Unique Identifier (UUID) generiert, ein Journal erstellt, sowie die Blöcke in denen sich die Superblock-Backups befinden, ausgegeben.

1.3.1. Erstellen von Dateien

Folgendes Listing erzeugt eine Datei:

```
1 # Assuming the fs is mounted on /media/usb
2 cd /media/usb
3 touch mydocument.txt
4 echo "Hallo welt." > mydocument.txt
```

Mit Hilfe des Kommandos `ls -li` können wir uns die Inode-ID ausgeben lassen um diese dann mit Hilfe des SleuthKit ([4]) Kommandos `istat` näher zu betrachten. Abbildung 5 zeigt die Ausgabe der Manual-Page für das Tool.

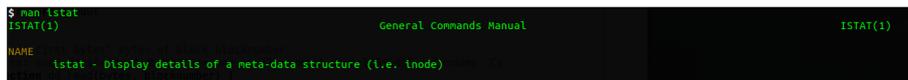


Abbildung 5: Manual page *istat*

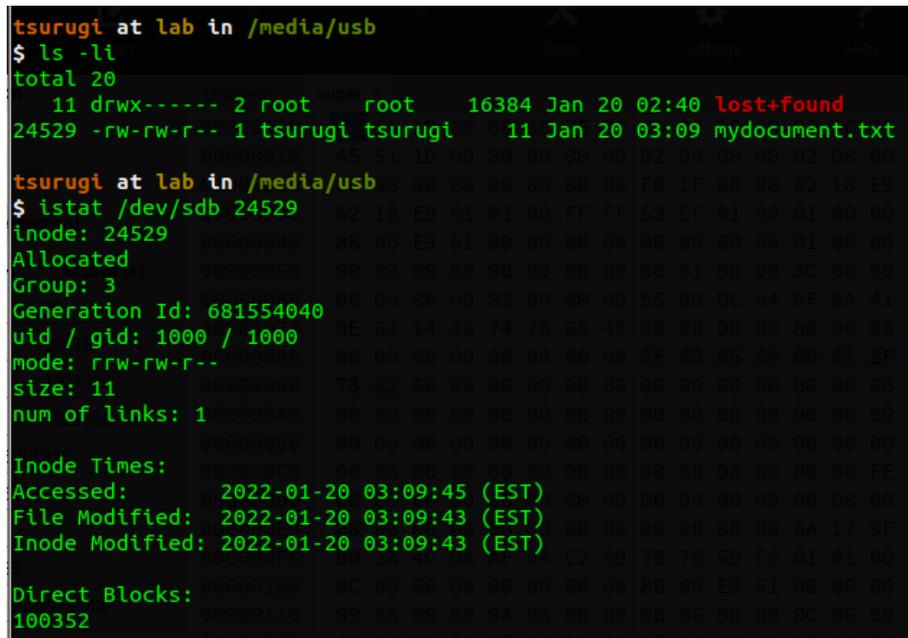


Abbildung 6: Informationen einer Inode

Wir sehen in Abbildung 6, dass unsere zuvor erstellte Datei *mydocument.txt* zur Inode 24529 gehört. Sie befindet sich in Blockgruppe 3, hat eine Größe von 11 Bytes (Inhalt: *Hallo welt.*) und der Inhalt der Datei befindet sich in Block Nummer 100352.

Mit Hilfe von *dd* wollen wir uns nun den Inhalt des Blocks 100352 ansehen, um zu verifizieren ob dort auch die Daten unserer Datei liegen:

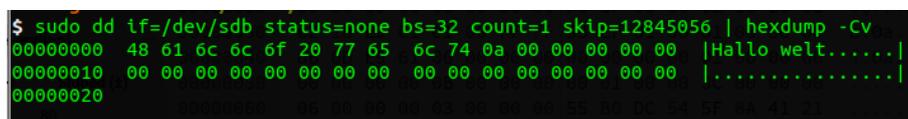


Abbildung 7: Rohen Inhalt eines Blocks mit *dd* auslesen

Da jeder Block in unserem Dateisystem 4096 Bytes groß ist, beginnt der Inhalt dieser Datei an Byteoffset 411041792 (Blocknummer x Blockgröße). Wir kopieren mit *dd* immer 32 Byte, müssen daher mit Hilfe des *skip*-Arguments angeben, ab wann wir mit dem Kopieren starten wollen ($411041792/32 = 12845056$). Und siehe da, der Inhalt der Datei wird korrekt ausgegeben!

1.3.2. Löschen von Dateien

Wir löschen die Datei mit folgendem Kommando:

```
1 rm -rf mydocument.txt
```

Wir wissen noch die Inode ID 24529. Abbildung 8 zeigt die Analyse dieser Inode mittels *istat* nach dem Löschen der Datei:

```
$ istat /dev/sdb 24529
inode: 24529
Not Allocated
Group: 3
Generation Id: 681554040
uid / gid: 1000 / 1000
mode: rrw-rw-r--
size: 0
num of links: 0

Inode Times:
Accessed:      2022-01-20 11:40:18 (EST)
File Modified: 2022-01-20 11:40:24 (EST)
Inode Modified: 2022-01-20 11:40:24 (EST)
Deleted:       2022-01-20 11:40:24 (EST)

Direct Blocks:
```

Abbildung 8: Inode-Inhalt nach dem Löschen auslesen

Die Inode gilt nun nicht mehr als allokiert (*Not Allocated*), die Dateigröße wurde auf 0 Byte gesetzt und die Referenz zum Inhaltsblock entfernt (Blockpointers zeroed). Außerdem kam ein *Deleted*-Datensatz hinzu. Dennoch können wir immer noch die Daten der vorherigen Blocknummer mittels *dd* wie oben beschrieben auslesen und bekommen den Inhalt der Datei. Es wird also nicht der Inhalt, sondern nur die Referenz in der Inode gelöscht und somit auch der Speicherplatz freigegeben. Dieser kann nun zu späterem Zeitpunkt überschrieben werden.

2. Forensische Analyse des EXT3 Dateisystems

2.1. Erstellen einer forensischen Arbeitskopie

Normalerweise würden wir vor einer Analyse immer zuerst eine forensische Arbeitskopie erstellen. Aus Gründen der Einfachheit wird hier jedoch darauf verzichtet und bei sämtlichen Analysen direkt auf das Blockdevice (*/dev/sdb*) zugegriffen.

Folgende Kommandos könnten verwendet werden:

```
1 $ sudo dd if=/dev/sdb of=usb.dd bs=512 conv=noerror, sync  
   status=progress  
2  
3 # dc3dd is a forensics extension of dd (e.g. hash based  
   verification integrated)  
4 dc3dd if=/dev/sdb hof=/path/to/my/backup.raw hash=sha256
```

2.2. Forensische Konzepte zur Auswertung

Im diesem Kapitel werden einige Konzepte zur Auswertung von Ext-Dateisystemen vorgestellt. Wir wissen, dass im Superblock wichtige Informationen über das Dateisystem gespeichert sind und können somit dort ansetzen. Sollte der Superblock korrupt sein, ist es auch relativ einfach nach Backups zu suchen um dann diese zu verwenden.

Wir könnten den Superblock wie oben bereits beschrieben manuell analysieren, mit Hilfe des Tools *fsstat* ist es jedoch einfacher, sich grundlegende Information ausgeben zu lassen. Das Tool gibt Metadaten, sowie Inhaltsinformationen aus und anschließend Informationen zu jeder Blockgruppe. Diese Informationen können für weitere Untersuchungen sehr hilfreich sein. Abbildung 9 zeigt die Ausgabe des Tools für unseren USB Stick. Wir sehen z.B. auch, welche Blöcke in welcher Gruppe liegen, eine Information die besonders relevant sein kann.

```
$ sudo fsstat /dev/sdb
FILE SYSTEM INFORMATION
-----
File System Type: Ext3
Volume Name:
Volume ID: 4e657b743614629e21418a5f54dcb055

Last Written at: 2022-01-21 03:31:07 (EST)
Last Checked at: 2022-01-20 02:21:47 (EST)

Last Mounted at: 2022-01-21 03:31:07 (EST)
Unmounted properly
Last mounted on: /media/usb

Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode, Dir Index
InCompat Features: Filetype, Needs Recovery,
Read Only Compat Features: Sparse Super, Large File,

Journal ID: 00
Journal Inode: 8

METADATA INFORMATION
-----
Inode Range: 1 - 1921361
Root Directory: 2
Free Inodes: 1921347

CONTENT INFORMATION
-----
Block Range: 0 - 7679999
Block Size: 4096
Free Blocks: 7515361

BLOCK GROUP INFORMATION
-----
Number of Block Groups: 235
Inodes per group: 8176
Blocks per group: 32768

Group: 0:
  Inode Range: 1 - 8176
  Block Range: 0 - 32767
  Layout:
    Super Block: 0 - 0
    Group Descriptor Table: 1 - 2
    Data bitmap: 1025 - 1025
    Inode bitmap: 1026 - 1026
    Inode Table: 1027 - 1537
    Data Blocks: 1538 - 32767
  Free Inodes: 8165 (99%)
  Free Blocks: 0 (0%)
  Total Directories: 2

Group: 1:
  Inode Range: 8177 - 16352
  Block Range: 32768 - 65535
```

Abbildung 9: Ausgabe von fsstat für unseren USB Stick

Für die weitere Auswertung von Inhalten, ist es wichtig den Allokationsstatus von Blöcken zu kennen. Ist ein Block belegt, so ist das ein Hinweis darauf, dort

verwertbare Daten zu finden. Diesen Allokationsstatus zu finden ist ein 3-Schritte-Prozess ([2]):

1. Feststellen, zu welcher Blockgruppe der Block gehört
2. Suchen des Eintrags der Gruppe in der Group-Descriptor-Tabelle um festzustellen, wo ihre Block-Bitmap gespeichert ist
3. Zuletzt wird die Block-Bitmap analysiert und das dem betreffenden Block zugeordnete Bit untersucht. Wenn das Bit gesetzt ist, ist der Block allokiert

Der Allokationsstatus kann auch für das Wiederherstellen gelöschter Dateien hilfreich sein, denn mit dessen Hilfe kann die Suche nach bestimmten Inhalten (z.B. Keywords) auf gewissen Blockgruppen beschränkt werden, was viel Zeit sparen kann.

Bisher haben wir den Inhaltsbereich analysiert. Ein weiterer Ansatz ist es, den Metadatenbereich zu verwenden. In unserem praktischen Beispiel haben wir bereits gesehen, dass nach dem Löschen einer Datei immer noch die Inode ausgelesen werden kann. Die Größe und Blockreferenz wird zwar gelöscht, dennoch gibt es einen *Deleted*-Datensatz mit dem Zeitstempel der Löschung, sowie weitere oben beschriebenen Informationen sind noch intakt. Diese Informationen in den Inodes können hilfreich für eine weitere Auswertung sein.

Weitere Möglichkeiten der Auswertung basierend auf der Dateinamen- oder Applikationskategorie werden in [2] beschrieben, darauf wird jedoch hier nicht weiter eingegangen. Abschließend befassen wir uns im nächsten Kapitel mit dem Wiederherstellen gelöschter Dateien.

2.3. Wiederherstellen gelöschter Dateien

Zurück zu unserem Beispiel oben, wir wissen ja noch die Inode ID 24529. Außerdem kennen wir den Inhalt „Hallo Welt“. Somit könnten wir als erste Idee zur Auswertung einfach nach dem Inhalt suchen:

```
1 $ sudo strings /dev/sdb | grep "Hallo"
```

Dieses Kommando findet zwar den Inhalt noch, obwohl die Datei gelöscht wurde. Das sagt uns aber nichts über den Speicherort und bei verschlüsselten oder kodierten Inhalten würde uns das auch nicht weit bringen. Außerdem kennen wir normalerweise auch nicht die Inode. Wir benötigen bessere Methoden zur Auswertung und Datenwiederherstellung. Zwei davon werden im Folgenden präsentiert und erläutert.

2.3.1. File-Carving

Die Wiederherstellung von Dateien ist bei Ext3-Dateisystemen nicht einfach. Bei Ext2 werden die Inode-Werte nicht gelöscht, wenn die Datei gelöscht wird, so dass

die Blockzeiger noch vorhanden sind. Die Zeitwerte werden ebenfalls aktualisiert, wenn die Datei gelöscht wird, so dass man weiß, wann die Löschung erfolgt ist.

Bei Ext3 ist der Zusammenhang zwischen dem Dateinamen und dem Inode noch vorhanden, aber die Blockpointer werden gelöscht. Um die Daten wiederherzustellen, ist es möglich sog. File-Carving-Techniken anzuwenden. Gruppeninformationen lassen sich aus Performancegründen hier zu unserem Vorteil nutzen. Die Zuweisung von Blöcke in Blockgruppen ist jedoch abhängig vom Betriebssystem, es könnte also auch ein Carving des gesamten Platzes notwendig sein: es kann also ein zeitintensives Unterfangen sein.

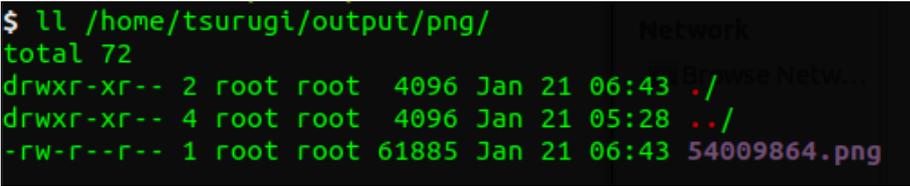
File-Carving bedeutet das Suchen nach bestimmten Mustern (wie z.B. JPEG header Informationen) im rohen Binärdaten und somit das Zusammenfassen von unstrukturierten Inhalten zu einer Datei. Ein Tool dafür soll hier vorgestellt werden: *Foremost* [5]

Foremost ist ein Tool zur Wiederherstellung gelöschter Daten unter Verwendung von gängigen Datei-headern, footern sowie internen Datenstrukturen. Die Config-Datei unter `/etc/foremost.conf` enthält binäre Signaturen zur Erkennung vieler gängiger Dateitypen.

Für das praktische Beispiel wurde eine *png*-Datei angelegt und anschließend via *rm* gelöscht. Folgendes Kommando wurde zur Wiederherstellung verwendet:

```
1 sudo foremost -t jpg,png -v -q -d -b 4096 -i /dev/sdb -o  
/home/tsurugi/output
```

Nach ca. zwei Stunden war *foremost* fertig und hat den Screenshot korrekt wieder hergestellt:



```
$ ll /home/tsurugi/output/png/  
total 72  
drwxr-xr-- 2 root root 4096 Jan 21 06:43 ./  
drwxr-xr-- 4 root root 4096 Jan 21 05:28 ../  
-rw-r--r-- 1 root root 61885 Jan 21 06:43 54009864.png
```

Abbildung 10: Das Tool *foremost* speichert wiederhergestellte Dateien im *output* Ordner

Eine weitere Methode zur Wiederherstellung gelöschter Daten wird im nächsten Kapitel vorgestellt.

2.3.2. Das File-Journal

Das File-Journal wurde in Ext3 eingeführt und stellt die Integrität eines Dateisystems sicher, indem es eine Art Logbuch von Schreibzugriffen speichert. Dies ist u.a.

hilfreich bei einem Systemabsturz oder Stromausfall. Mit ein bisschen Glück lassen sich in diesem Journal noch alte Inode-Kopien gelöschter Dateien finden, welche immer noch die Block-Referenzen beinhalten. Durch diesen Ansatz kann der Inhalt wiederhergestellt werden.

Ein Tool zur Automatisierung der Wiederherstellung von Daten in Ext-Dateisystemen, welches auf dem oben beschriebenen File-Journal Ansatz basiert, soll hier abschließend vorgestellt werden: *ext4magic* [6]. Abbildung 11 zeigt das Tool in Aktion. Nach erfolgreicher Wiederherstellung wird hier der Inhalt mittels *cat* ausgegeben (Inode 24529).

```
root at lab in /media
$ ext4magic /dev/sdb -M
Warning: Activate magic-scan or disaster-recovery function, may be some command line options ignored
"RECOVERDIR" accept for recoverdir
Filesystem in use: /dev/sdb

Using internal Journal at Inode 8
Activ Time after : Fri Jan 21 03:10:05 2022
Activ Time before : Fri Jan 21 03:14:33 2022
Inode 2 is allocated
-----
RECOVERDIR/lost+found
RECOVERDIR/RECOVERDIR
RECOVERDIR/index.js#
RECOVERDIR/
MAGIC-1 : start lost directory search
MAGIC-2 : start lost file search
-----
RECOVERDIR/MAGIC-2/text/plain/I_0000024529.txt#
MAGIC-2 : start lost in journal search
-----
RECOVERDIR/MAGIC-2/text/plain/I_0001921361.txt#
MAGIC function for ext3 not available, use ext4magic 0.2.4 instead
ext4magic : EXIT_SUCCESS

root at lab in /media
$ cat RECOVERDIR/MAGIC-2/text/plain/I_0000024529.txt
I_0000024529.txt I_0000024529.txt#

root at lab in /media
$ cat RECOVERDIR/MAGIC-2/text/plain/I_0000024529.txt
Hallo welt.
```

Abbildung 11: Wiederherstellung gelöschter Daten mit dem Tool *ext4magic* [6]

3. Abbreviations

EXT Extended file system

UUID Universal Unique Identifier

4. Literatur

- [1] D. Poirier, *The Second Extended File System - Internal Layout*. Adresse: <https://www.nongnu.org/ext2-doc/ext2.pdf>.
- [2] B. Carrier, *File System Forensic Analysis*. Adresse: <https://learning.oreilly.com/library/view/file-system-forensic/0321268172/pref00.html>.
- [3] K. D. Fairbanks, *An analysis of Ext4 for digital forensics*. Adresse: <https://github.com/skeledrew/ext4-raw-reader/blob/2fc1f623e548f8c7b7e9f11eefbc0792242493d4/DFRWS2012-13.pdf>.
- [4] *The SleuthKit*. Adresse: <https://sleuthkit.org/>.
- [5] *Foremost - a tool to recover lost files*. Adresse: <https://www.kali.org/tools/foremost/>.
- [6] *CLI Tool ext4magic*. Adresse: <https://github.com/gktrk/ext4magic>.

A. Appendix: Script zum Auslesen des Inhalts von Datenblöcken (JavaScript)

```
1  /**
2  * Simple script for reading out the raw bytes of a given
   blocknumber
3  *
4  * Usage: node index.js $BLOCKNUMBER
5  */
6  const { execSync } = require('child_process')
7  const BLOCKSIZE = 4096
8
9  function exec(cmd) {
10     const stdout = execSync(cmd)
11     return stdout
12 }
13
14 // read some bytes of a specific block
15 function dd_read(bytes, blocknumber) {
16     return exec(`sudo dd if=/dev/sdb status=none bs=${
       bytes} count=1 skip=${(BLOCKSIZE * blocknumber) /
       bytes} | hexdump -Cv` )
17 }
18
19 const blocknumber = parseInt(process.argv[2])
20 const data = dd_read(512, blocknumber).toString()
21
22 console.log(data)
```