

Synthetisch erzeugtes Dateisystem

Entwicklung eines Tools zur syntethischen Erzeugung von FAT32-Asservaten

Hausarbeit

zu Modul 111 - Datenträger-Forensik



vorgelegt von: Michael Koll

Matrikelnummer:

Prüfer:

© 2018

Inhaltsverzeichnis

Abkürzungsverzeichnis	3
1. Einleitung	4
2. FAT32 - Forensische Bewertung	5
2.1. Grundlagen	5
2.2. Kategorie: Dateisystem	7
2.3. Kategorie: Inhalt	8
2.4. Kategorie: Metadaten	9
2.5. Kategorie: Dateinamen	12
2.6. Datenstrukturen	12
2.6.1. Bootsektor	12
2.6.2. FAT32 FSINFO	15
2.6.3. File Allocation Table (FAT)	16
2.6.4. Directory Entries	17
3. Design	20
3.1. Dateisystemkomponente	21
3.2. Ablaufkomponente	23
3.3. Konfigurationskomponente	24
3.4. Wichtige Konzepte	26
3.5. Implementierte UseCases	27
3.5.1. RawWrite	27
3.5.2. CreateImage	28
3.5.3. FAT32CreateBootSector	28
3.5.4. CreateDir	29
3.6. Fehler und Erweiterungen	29
4. Fallbeispiele	32
4.1. Installation und Setup	32
4.2. Dateioperationen	36
4.3. Existierendes Dateisystem	40
5. Fazit	42

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

Literatur	43
Eidesstattliche Erklärung	44
Verzeichnis der Listings	45
Anwendungsfälle	46
Abbildungsverzeichnis	47
Tabellenverzeichnis	48
A. Anhang	49
A.1. Aufgabenstellung	50

Abkürzungsverzeichnis

EOC End-of-Clusterchain.

EOF End-of-File.

FAT File Allocation Table.

LFN Long File Name.

SFN Short File Name.

1. Einleitung

Für forensische Analysen ist es unabdingbar gewisse Szenarien zu üben oder für Simulationszwecke nachstellen zu können. Bei kriminaltechnischen Untersuchungen spielen Asservate von sichergestellten Datenträgern eine wichtige Rolle. In dieser Hausarbeit wird ein Prototyp zur Erstellung von synthetischen Asservaten entwickelt, der es ermöglicht forensische relevante Anwendungsfälle und Artefakte auf einem Datenträger zu erzeugen. Dabei ist die Wiederholbarkeit und Konsistenz entscheidend, damit jederzeit der Zustand des Datenträgers und der darauf befindlichen Daten bekannt ist.

Für die Implementierung des Prototyps wurde das FAT32-Dateisystem gewählt, da dieses aufgrund der wenigen Datenstrukturen einen relativ einfachen Aufbau hat. In Kapitel 2 wird eine Einführung in das Dateisystem FAT32 und die Datenstrukturen gegeben. An den relevanten Stellen werden die möglichen UseCases beschrieben, welche später als Artefakte auf dem Datenträger hinterlegt werden könnten.

In Kapitel 3 wird das entwickelte Konzept des Tools vorgestellt. Dabei wird der grundlegende Aufbau der drei Hauptkomponenten Dateisystem, Konfiguration und Ablaufsteuerung beschrieben. Zusätzlich werden die bereits implementierten Anwendungsfälle vorgestellt und ein Ausblick auf mögliche Erweiterungen gegeben.

Zum Schluss wird in Kapitel 4 eine praktische Demonstration des Tool durchgeführt, indem der aktuelle Implementierungsstand vorgestellt und auf Besonderheiten hingewiesen wird.

2. FAT32 - Forensische Bewertung

Um synthetische forensische Asservate zu erstellen ist ein tieferes Verständnis des Dateisystems und der Besonderheiten essentiell. In diesem Kapitel wird der Aufbau, die Datenstrukturen und das Zusammenspiel dieser erläutert. Gleichzeitig wird versucht auf forensisch relevante Punkte einzugehen und diese zu bewerten. Diese Themen werden im Folgenden unter dem Begriff „UseCases“ geführt und in den späteren Kapiteln als mögliche Schritte beider Asservatserstellung betrachtet und implementiert. Eine Auflistung aller UseCases ist in Kapitel 5 zu finden.

Das FAT Dateisystem (siehe [3]) wurde von Microsoft entwickelt und war lange Zeit das Standarddateisystem der MS-DOS- und Windows-Betriebssysteme. Mit der Zeit wurde die Spezifikation weiterentwickelt und im Jahr 1996 für das Betriebssystem Windows 95 das FAT32 Dateisystem eingeführt. Ziel war die Unterstützung größerer Datenträger bei gleichzeitiger Weiternutzung des vorhandenen Codes. FAT32 unterstützt je nach Sektorengröße Datenträger bis zu 2 TiB einer Sektorengröße von 512 Bytes und 16 TiB bei einer Sektorengröße von 4096 Bytes. Im Microsoft-Umfeld wurde das FAT-Dateisystem mittlerweile durch das NTFS-Dateisystem abgelöst. FAT-Dateisysteme sind allerdings aufgrund der geringen Komplexität häufig auf SD-Karten und USB-Sticks zu finden. Die geringe Komplexität des FAT32-Dateisystems beruht auf der geringen Anzahl an Datenstrukturen und wurde daher als Referenz für diese Hausarbeit gewählt.

Die Beschreibung in den folgenden Kapiteln beruht hauptsächlich auf der offiziellen Spezifikation des FAT32-Dateisystems (siehe [3]) und der Beschreibung und Kategorisierung der Daten von Brian Carrier (siehe [1, S. 156–198])

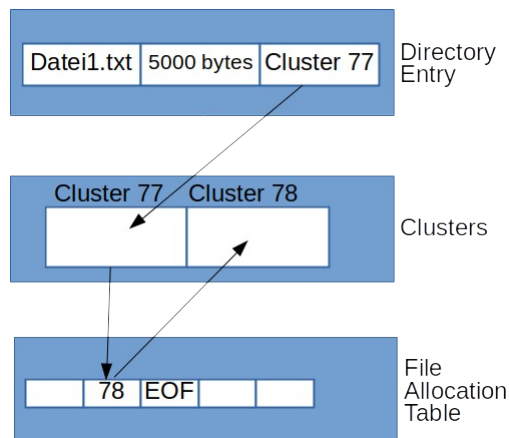
2.1. Grundlagen

Die beiden wichtigsten Datenstrukturen des FAT-Dateisystems sind die File Allocation Table (FAT) und Directory Entries (Verzeichniseinträge). Das Grundkonzept des Dateisystems ist, dass jeder Datei und jedem Verzeichnis eine Datenstruktur, genannt Directory Entry, zugewiesen wird, die den Datei- oder Verzeichnisnamen, die Größe, die Startadresse des Dateiinhalts und Metadaten wie Zeitstempel enthält.

Der Datei- und Verzeichnisinhalt wird in Data Units bzw. Clustern gespeichert. Falls für eine Datei oder ein Verzeichnis mehr als ein Cluster benötigt wird, können die folgenden Cluster durch die FAT-Datenstruktur gefunden werden.

Die FAT enthält für jedes Cluster im Dateisystem einen Eintrag, der den Allokationsstatus und, falls vorhanden, den nächsten Cluster benennt.

Abbildung 2.1 zeigt exemplarisch das Zusammenspiel der verschiedenen Datenstrukturen. Der Verzeichniseintrag enthält neben den Eigenschaften der Datei den Startcluster des Dateiinhalts (in diesem Fall Cluster 77). Der Eintrag des Cluster 77 in der FAT-Tabelle



enthält einen Verweis auf Cluster 78, der Eintrag des Cluster 78 beinhaltet eine End-of-File-Markierung. Der Dateiinhalt der `Datei1.txt` ist dementsprechend in den Clustern 77 und 78 enthalten.

Der physikalische Aufbau des FAT-Dateisystems gliedert sich in drei Bereiche (siehe Abbildung 2.2):

- **Reserviert** Enthält den Bootsektor und die FSInfo-Datenstruktur
- **FAT** Enthält die File Allocation Table und die Backupkopie dieser (falls vorhanden)
- **Daten** Datenbereich mit Directory Entries und Dateiinhalten

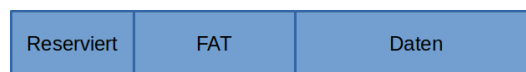


Abbildung 2.2.: Physikalischer Aufbau des FAT-Dateisystems

In den folgenden Kapiteln werden die Daten des Dateisystems nach den Kategorien von Carrier beschrieben.

2.2. Kategorie: Dateisystem

Daten der Kategorie Dateisystem können im Bootsektor (oder BIOS Parameter Block, vgl. [3, S. 7]) gefunden werden. Der Bootsektor (detaillierte Struktur in Kapitel 2.6.1) befindet sich im reservierten Bereich des Dateisystems im ersten Sektor (Sektor 0) und enthält essentielle Daten über die physikalische Struktur und den Aufbau des Dateisystems, also die Position der FAT und des Datenbereichs, ebenso wie die Position des Backup Bootsektors. Der reservierte Bereich enthält weiterhin die FSINFO-Datenstruktur (detaillierte Beschreibung in Kapitel 2.6.2), welche Informationen über die Anzahl der freien Cluster und den zuletzt beschriebenen Cluster enthält.

Die essentiellen Positionsinformationen des Bootsektors sind:

Backup Bootsektor Position des Backup Bootsektors ist als Sektoradresse im Bootsektor enthalten (Microsoft empfiehlt die feste Adresse Sektor 6)

FAT-Position Die File Allocation Table beginnt in dem Sektor nach dem reservierten Bereich. Die Größe des reservierten Bereichs wird im Bootsektor festgelegt. Die Größe einer FAT wird in Sektoren im Bootsektor hinterlegt. Die Backup FAT folgt der ersten FAT.

Datenbereich Die Position des Datenbereichs kann durch $\text{Größe des reservierten Bereichs} + \text{Anzahl der FATs} * \text{Größe einer FAT}$ ermittelt werden.

UseCase 1: Manipulieren der Strukturinformationen

Durch gezieltes Verändern der Daten im Bootsektor kann die Lokalisierung der verschiedenen Datenstrukturen erschwert werden. Zum Beispiel kann durch einen falschen Größenwert des reservierten Bereichs weder die FAT, noch der Datenbereich ohne manuelle Analyse gefunden und gelesen werden. Zu beachten ist, dass bzw. ob die Zeichenketten auch im Backup-Bootsektor geändert werden bzw. wurden

Der Bootsektor enthält weitere, nicht-essentielle Daten, die allerdings zur forensischen Analyse interessant sein können. Unter anderem ist dies ein String *OEMName* mit acht Zeichen, der typischerweise das zum Erstellen des Dateisystems genutzte Tool enthält (z. B. *MSDOS5.0* oder *mkdosfs*) oder den Namen des Gerätes bei Flash-Speicherkarten. Weiterhin existiert ein Feld für eine Seriennummer, die durch einige Betriebssysteme mit dem Erstellungsdatum des Dateisystems gefüllt wird, ebenso wie ein Feld für das genutzte Dateisystem (z. B. *"FAT32"*). Die letzte relevante Zeichenkette ist das Volumelabel, das typischerweise beim Erstellen des Dateisystems durch den Benutzer gewählt werden kann. Dieser wird sowohl im Bootsektor, als auch im

Wurzel-Verzeichniseintrag (Root-Directory-Entry) hinterlegt. Diese Daten sind für das Lesen und Nutzen des Dateisystems nicht essentiell, können für eine forensische Analyse aufgrund der enthaltenen Informationen interessant sein.

UseCase 2: Manipulieren von Zeichenketten

Durch Verändern der Zeichenketten (OEMName, Seriennummer, Dateisystemtyp, Volumelabel) können gezielt Falschinformationen hinterlegt werden (z. B. das Volumelabel *Urlaub* statt *Steuer* oder das Verschleiern des genutzten Tools). Weiterhin wäre es möglich in diesen Feldern Informationen zu *verstecken*, also z. B. die korrekte Größe des reservierten Bereichs, falls dieser manipuliert wurde (siehe UseCase 1). Zu beachten ist, dass bzw. ob die Zeichenketten auch im Backup-Bootsektor geändert werden bzw. wurden.

UseCase 3: Bootcode-Slack

Die Bytes 90 bis 509 sind typischerweise für den Bootcode vorgesehen. Da allerdings Bootcode nicht zwingend benötigt wird, kann dieser Platz zum Speichern von Daten oder Informationen genutzt werden.

UseCase 4: Reservierter-Bereich-Slack

Da der reservierte Bereich meistens größer als die benötigten Datenstrukturen (Bootsektor, FSINFO und Backupbootsektor) ist, kann der verbleibende Platz zum Verstecken von Daten genutzt werden.

2.3. Kategorie: Inhalt

Die Daten der Kategorie Inhalt geben Informationen über die Position von Datei- und Verzeichniseinhalten. Die Daten eines FAT-Dateisystems werden in Clustern geordnet. Ein Cluster besteht aus einer festgelegten Anzahl an Sektoren aus der 2er Potenz, wobei der erste Cluster eines FAT-Dateisystems immer 2 ist. Das Wurzelverzeichnis war in FAT12 und FAT16-Dateisystemen immer in den ersten beiden Clustern des Datenbereichs. Bei FAT32-Dateisystemen wird die Position des Wurzelverzeichnisses im Bootsektor hinterlegt, um z. B. beschädigte Sektoren umgehen zu können.

UseCase 5: Position des Wurzelverzeichnisses

Durch Manipulieren der Positionsangabe des Wurzelverzeichnisses im Bootsektor ist es nicht mehr bzw. nur durch hohen Aufwand möglich die Verzeich-

nisstruktur auszulesen. Durch diese Technik kann ein Dateisystem unbrauchbar gemacht werden. Vorstellbar wäre z. B. als Position irgendeinen beliebigen Cluster anzugeben und den korrekten Wert in einem der Zeichenketten oder in einem Slackbereich zu hinterlegen.

Durch die File Allocation Table wird der Allokationsstatus der Cluster gespeichert. Für jeden Cluster existiert also ein korrespondierender Tabelleneintrag. In einem FAT32-Dateisystem ist jeder Eintrag 32 Bit groß, wobei nur 28 Bit verwendet werden. Falls der Tabelleneintrag 0 ist, bedeutet dies, dass der Cluster nicht durch eine Datei oder ein Verzeichnis alloziert ist. Falls der Cluster beschädigt ist enthält der Tabelleneintrag `0x0fff fff7`. Alle anderen Werte bedeuten, dass der Cluster zugeordnet ist. Weitere mögliche Werte werden in Kapitel 2.4 (Metadaten) erläutert.

UseCase 6: Allokationsstatus eines Clusters ändern

Durch Ändern des Allokationsstatus eines Clusters können vorhandene Daten "versteckt" werden. Das Ändern eines Tabelleneintrags eines belegten Clusters auf `0x0000 0000` (frei) oder `0x0fff fff7` (beschädigt) verschleiert die Existenz dieser Daten.

UseCase 7: FAT-Slack

Da für eine FAT eine ganze Anzahl an Sektoren reserviert wird, kann zwischen der ersten FAT und der FAT-Kopie ein ungenutzter Bereich entstehen, der es erlaubt Daten zu speichern.

UseCase 8: Volume- oder Partitionsslack

Wenn das Volume oder die Partition nicht vollständig durch das Dateisystem ausgenutzt wird entsteht ein freier Bereich, der **Volumeslack** genannt wird (vgl. [1, S. 132]). Bei einem FAT-Dateisystem kann dies z. B. durch die Wahl der Clustergröße entstehen, wenn der die Größe des Datenbereichs kein Vielfaches der Clustergröße ist.

2.4. Kategorie: Metadaten

Metadaten beschreiben Eigenschaften einer Datei oder eines Verzeichnisses, wie z. B. die Position der Daten, Zeitstempel und Berechtigungen. Diese Informationen

sind in FAT-Dateisystemen in Directory Entries und in der FAT (speziell Informationen über die Struktur einer Datei/eines Verzeichnisses) zu finden. Verzeichnisse werden in FAT-Dateisystemen wie ein spezieller Dateityp behandelt.

Beim Anlegen eines Verzeichnisses wird im Elternverzeichnis ein Directory Entry angelegt. Jedes Directory Entry hat ein Attribut, welches den Typ des Eintrags bezeichnet. Die essentiellen Attribute sind Directory Long File Name (LFN) und Volume Label. Falls kein Attribut gesetzt ist, ist der Eintrag für eine Datei. Nicht-essentielle Attribute, die je nach Betriebssystem unterschiedlich interpretiert werden, sind Read Only, Hidden, System und Archive. Ein Directory Entry enthält weiterhin das Startcluster des Verzeichnisses.

Jedes Directory Entry enthält drei Zeitstempel: Erstellt, Letzter Zugriff und Letzte Änderung. Diese Zeitstempel haben unterschiedliche Genauigkeiten und nur der Erstellt-Zeitstempel ist verpflichtend. Der Zeitpunkt oder die Operation wann welcher Zeitstempel aktualisiert werden muss ist nicht festgelegt und betriebssystemspezifisch. Dies hat zur Folge, dass bei der Auswertung der Zeitstempel die Aussagekraft nur mit vorheriger Kenntnis des Betriebssystemverhaltens bewertet werden kann. Die Zeitstempel werden unter Beachtung der lokalen Zeitzone erstellt.

UseCase 9: Ändern der Zeitstempel

Durch das Ändern der Zeitstempel können forensische Analysen erschwert oder verhindert werden, bei denen die zeitliche Abfolge entscheidend ist.

In einem Directory Entry wird das erste Byte mit dem ersten Buchstaben des Verzeichnisnamens beschrieben. Falls das Verzeichnis oder die Datei gelöscht werden wird das erste Byte auf den Wert `0xe5` geändert, damit gilt das Directory Entry als nicht-alloziert.

UseCase 10: Allokationsstatus eines Directory Entry ändern

Durch das Überschreiben des ersten Bytes eines Directory Entry mit `0xe5` wird das Directory Entry als nicht-alloziert markiert und von Betriebssystemen nicht mehr gelesen. Somit ist es möglich einen Eintrag zu verstecken, obwohl der Inhalt und die Metadaten noch vorhanden sind. Eine zweite Möglichkeit ist das Setzen des ersten Bytes auf `0x00`. Dieser spezielle Wert besagt, dass das Directory Entry und alle folgenden Directory Entries ebenfalls nicht alloziert sind.

UseCase 11: Reallokation von Entry Adressen

Die Startadresse eines Verzeichnisses, die in einem Directory Entry angegeben wird, kann verändert werden, so dass der Zeiger auf einen anderen Cluster zeigt, obwohl die Daten des Verzeichnisses weiterhin vorhanden sind. Solche Fälle können nur gefunden werden, indem alle Datencluster analysiert und anhand der `..`-Verzeichnisse^a die korrekte Zuordnung überprüft wird.

^a Jedes Directory Entry enthält eine `..`-Verzeichnis, welches auf das aktuelle Verzeichnis zeigt und ein `..`-Verzeichnis, welches die Clusternummer des Elternclusters beinhaltet.

Da eine Datei oder Verzeichnis nicht auf die Clustergröße beschränkt sind, werden durch die FAT sogenannte Cluster Chains erstellt. Wie bereits erwähnt, enthält der FAT-Eintrag des korrespondierenden Clusters die Adresse des nächsten Clusters des Verzeichnisses oder der Datei. Beim Lesen eines Eintrags wird aus dem Directory Entry das Startcluster ermittelt. Anschließend wird dieses Cluster gelesen und dann in dem FAT-Eintrag geprüft, ob weitere Cluster zu diesem Eintrag gehören. In diesem Fall enthält der FAT-Eintrag die Nummer des nächsten zugehörigen Clusters. Falls es keinen weiteren Cluster gibt, enthält die FAT eine End-of-File (EOF)-Markierung. Microsoft Betriebssysteme nutzen als EOF- oder End-of-Clusterchain (EOC)-Markierung den Wert `0x0fffffff`, möglich ist allerdings jeder Wert, der keine gültige Clusteradresse darstellt.

UseCase 12: Verändern der Clusterchain

Durch das Verändern der FAT-Einträge kann eine Clusterchain manipuliert und die folgenden Daten versteckt werden. Dies könnte z. B. der Fall bei einem Verzeichnis sein, welches über mehrere Cluster geht. Wenn in diesem Fall in dem FAT-Eintrag des ersten Clusters eine EOC-Markierung gesetzt wird, wird der Rest des Verzeichnisses nicht mehr eingelesen und damit verschleiert. Eine weitere Möglichkeit wäre die Adresse des nächsten Clusters zu ändern. Durch geschicktes Setzen auf einen weiteren Directory Entry könnte somit die Manipulation effektiv verschleiert werden.

UseCase 13: Clusterslack

Da für jedes Directory Entry immer vollständige Cluster alloziert werden, kann der verbleibende Platz zum Verstecken von Daten genutzt werden. So kann z. B. ein Verzeichnis mit nur wenigen Inhalten angelegt und der restliche Platz des Clusters zum Verstecken von Daten genutzt werden. Bei Windows Betriebssystemen kann sich zum Nutzen gemacht werden, dass das Verzeichnis

¹ Der genutzte Wert ist abhängig von dem jeweiligen FAT-Treiber bzw. Betriebssystem

nicht weiter gelesen wird, sobald ein Directory Entry gefunden wurde, welches ausschließlich aus Nullen besteht (vgl. [1, S. 174]).

UseCase 14: Volume-Label-Clusterchain

Da ein Volume-Label einen normalen Directory Entry darstellt, kann die Startclusteradresse und der zugehörige FAT-Eintrag manuell angepasst werden, um eine Clusterchain zum Hinterlegen von Daten zu erstellen.

2.5. Kategorie: Dateinamen

Die Dateinamen wurden zum Teil bereits im vorherigen Titel beschrieben, da diese unmittelbar mit den Metadaten in Form der Directory Entries verbunden sind. Normale Directory Entries speichern Dateinamen in einem 8.3-Format (8 Zeichen für den Dateinamen und 3 Zeichen für die Dateieindung). Falls der Dateiname länger ist oder spezielle Zeichen enthält werden sogenannte LFN-Einträge genutzt. Dies sind ebenfalls Directory Entries, die das Attribut LFN gesetzt haben. Einträge dieser Art enthalten ausschließlich einen Teil des Dateinamens und keinen Startcluster oder Zeitstempel. Aus diesem Grund muss zu jedem LFN-Eintrag ein normales Directory Entry existieren. Die Dateinamen in LFN-Einträgen sind im Gegensatz zu den SFN-Einträgen in Unicode gespeichert und ermöglichen somit die Unterstützung eines erweiterten Zeichensatzes.

2.6. Datenstrukturen

In diesem Kapitel soll ein detaillierter Einblick in die FAT32-Datenstrukturen gegeben werden, um die grobe Beschreibung aus den vorherigen Kapiteln zu ergänzen. Die Tabellen wurden aus der FAT32-Spezifikation übertragen (siehe [3])

2.6.1. Bootsektor

Name	Offset (bytes)	Größe (bytes)	Essentiell	Beschreibung
------	-------------------	------------------	------------	--------------

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

BS_jmpBoot	0	3	Nein	Sprunganweisung zum Bootcode. Der Bootcode wird normalerweise hinter dem Bootsektor in dem restlichen Platz des Sektors 0 hinterlegt.
BS_OEMName	3	8	Nein	Frei wählbare Zeichenkette, typischerweise das genutzte Tool, um den Datenträger zu formatieren
BPB_BytesPerSec	11	2	Ja	Bytes pro Sektor, muss mit der Sektorengröße des Datenträgers übereinstimmen
BPB_SecPerClus	13	1	Ja	Anzahl der Sektoren pro Cluster, erlaubte Werte sind 1, 2, 4, 8 16, 32, 64 und 128
BPB_RsvdSecCnt	14	2	Ja	Anzahl der reservierten Sektoren am Anfang des Volumes, für FAT32 typischerweise 32
BPB_NumFATs	16	1	Ja	Anzahl der FAT-Datenstrukturen, empfohlener Wert 2
BPB_RootEntCnt	17	2	Ja	Nur für FAT12/FAT16, für FAT32: 0
BPB_TotSec16	19	2	Ja	Nur für FAT12/FAT16, für FAT32: 0
BPB_Media	21	1	Nein	0xf8 für nicht-entfernbarer Datenträger, 0xf0 für entfernbaren Datenträger. Der Wert muss im niedrigsten Byte der ersten FAT-Tabelle ebenfalls eingetragen werden
BPB_FATSz16	22	2	Ja	Nur für FAT12/FAT16, für FAT32: 0
BPB_SecPerTrk	24	2	Nein	Sektoren pro Spur des Datenträgers
BPB_NumHeads	26	2	Nein	Anzahl der Köpfe des Datenträgers
BPB_HiddenSec	28	4	Nein	Anzahl der versteckten Sektoren vor der Partition des Dateisystems
BPB_TotSec32	32	4	Ja	Anzahl der Sektoren des Volumes (MBR, BootSektor, FAT und Datenbereich)

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

BPB_FATSz32	36	4	Ja	Größe einer FAT in Sektoren
BPB_ExtFlags	40	2	Ja	Bits 0-3: Anzahl der aktiven FATs, Bits 4-6: Reserviert, Bit 7: 0 falls FAT gespiegelt wird, 1 falls nicht, Bits 8-15: Reserviert
BPB_FSVer	42	2	Ja	Versionssnummer der genutzten FAT-Spezifikation
BPB_RootClus	44	4	Ja	Clusteradresse des Root Directories. Empfohlen: 2
BPB_FSInfo	48	2	Nein	Sektoradresse der FSInfo-Datenstruktur
BPB_BkBootSec	50	2	Nein	Sektoradresse des Backup-Bootsektors, empfohlen: 6
BPB_Reserved	52	12	Nein	Reservierter Bereich, sollte mit 0x00 gefüllt sein
BS_DrvNum	64	1	Nein	0x00 für Floppy-Disks, 0x80 für Festplatten
BS_Reserved1	65	1	Nein	Reservierter Bereich, sollte mit 0x00 gefüllt sein
BS_BootSig	66	1	Nein	Erweiterte Bootsignatur 0x29, Hinweis, dass die nächsten 3 Felder des Bootsektors vorhanden sind
BS_VolID	67	4	Nein	Seriennummer des Volumes, kann frei gesetzt werden
BS_VolLab	71	11	Nein	Frei wählbares Volume-Label. Dieser Wert wird ebenfalls als Directory Entry im Root Directory hinterlegt
BS_FilSysType	82	8	Nein	Sollte auf den String FAT32 gesetzt werden

Tabelle 2.1.: Datenstruktur des Bootsektors eines FAT32-Dateisystems

Der Bootsektor legt die Grundstruktur des Dateisystems fest. Essentielle Felder werden benötigt, um die Daten des Dateisystems auslesen zu können. Nicht-essentielle Felder werden nur in bestimmten Fällen oder bei der Nutzung bestimmter FAT-Treiber und Medientypen benötigt und können zur Ablage von versteckten Informationen genutzt werden.

UseCase 15: Verstecken von Daten im Bootsektor

Zusätzlich zu dem Verstecken von Daten in den Zeichenketten (siehe UseCase 2) können in den reservierten Bereichen des Bootsektors (Offset 52 und 65) insgesamt 13 Bytes untergebracht werden. Zusätzlich ist es möglich nicht-essentielle Felder, wie z. B. die Anzahl der Datenträgerköpfe mit Informationen zu füllen, da diese nicht bzw. nur von bestimmten FAT-Treibern gelesen werden.

2.6.2. FAT32 FSINFO

Name	Offset (bytes)	Größe (bytes)	Essentiell	Beschreibung
FSI_LeadSig	0	4	Nein	Anfangssignatur zur Identifizierung der FSI-Struktur (0x41615252)
FSI_Reserved1	4	480	Nein	Reservierter Bereich für Erweiterungen
FSI_StrucSig	484	4	Nein	Mittleres Signaturfeld (0x61417272)
FSI_Free_Count	488	4	Nein	Anzahl der freien Cluster auf dem Volume, 0xffffffff falls nicht bekannt
FSI_Nxt_Free	492	4	Nein	Nächstes freies Cluster, damit nicht das komplette Dateisystem durchsucht werden muss
FSI_Reserved2	496	12	Nein	Reservierter Bereich für Erweiterungen
FSI_TrailSig	508	4	Nein	Endsignatur 0xaa550000

Tabelle 2.2.: FAT32 FSInfo-Struktur

Die FSInfo-Struktur gibt Informationen über den aktuellen Zustand des Dateisystems in Form der Anzahl der freien Cluster und der Adresse des nächsten freien Clusters. Da diese Informationen aber nicht zwingend vorhanden sein oder aktualisiert werden müssen, kann keine Gewähr auf die Korrektheit dieser Daten gegeben werden.

UseCase 16: FSInfo-Slack

In den beiden reservierten Feldern der FSInfo-Struktur können bis zu 492 Bytes an Informationen hinterlegt werden. Da das Vorhandensein einer FSInfo-Struktur nicht von allen FAT-Treiber vorausgesetzt wird bzw. nicht ausgewertet wird, können im Extremfall sogar 508 Bytes verwendet werden. Die Endsignatur sollte in jedem Fall erhalten bleiben.

UseCase 17: FSInfo-Manipulation

Durch das gezielte Ändern der Daten in der FSInfo-Struktur können vorherige Aktivitäten auf dem Datenträger verschleiert werden. Zum Beispiel kann die Adresse des nächsten freien Cluster verändert werden, um die Position des letzten schreibenden Zugriffs zu verschleiern.

2.6.3. File Allocation Table (FAT)

Die File Allocation Table besteht aus aufeinanderfolgenden 32-Bit-Feldern, wobei die Anzahl der Cluster im Dateisystem entspricht. Für jeden Cluster existiert also ein korrespondierender FAT-Eintrag.

Da die Clusteradressierung erst bei 2 beginnt, werden die ersten beiden Felder der FAT dazu verwendet eine Kopie des Medientyps und den "DirtyStatus (falls der Datenträger nicht korrekt entfernt wurde), wobei bei Felder nicht essentiell sind.

Jeder weitere 32-Bit Eintrag kann folgende Werte enthalten:

0x0000 0000 Der zugehörige Cluster ist nicht alloziert

0x0fff fff7 Der zugehörige Cluster ist beschädigt

größer als 0x0fff fff8 EOF- bzw. EOC-Markierung, markiert den letzten Cluster einer Clusterchain

jeder andere Wert Adressiert den nächsten Cluster in der Clusterchain

Normalerweise enthält ein FAT32-Dateisystem zwei FAT-Strukturen, wobei die genaue Anzahl im Bootsektor festgelegt wird. Diese folgen direkt aufeinander. Das Vorhandensein der Backuptabellen ermöglicht einen weiteren UseCase:

UseCase 18: FAT Backupkopie als echte Referenz

Vorausgesetzt der eingesetzte FAT-Treiber bzw. das Betriebssystem führen keine Konsistenzprüfung der beiden FAT-Kopien durch: Wenn wie in UseCase 6 der Allokationsstatus von Clustern manipuliert wird, könnte die FAT Backupkopie unverändert erhalten bleiben. Dies verschleiert die Informationen auf den ersten Blick, ermöglicht aber das Wiederherstellen der korrekten Clusterchains und FAT-Einträge.

2.6.4. Directory Entries

Directory Entries werden zum Speichern von Informationen über Dateien und Verzeichnisse benötigt.

Name	Offset (bytes)	Größe (bytes)	Essentiell	Beschreibung
DIR_Name[0]	0	1	Ja	Erstes Zeichen des Dateinamens, ansonsten 0xe5: nicht alloziert, 0x00: Dieses und alle weiteren Directory Entries sind nicht alloziert
DIR_Name	1	10	Ja	Zeichen 2-11 des Dateinamens im 8.3 Format in ASCII
DIR_Attr	11	1	Ja	Dateiattribute <ul style="list-style-type: none"> • ATTR_READ_ONLY 0x01 • ATTR_HIDDEN 0x02 • ATTR_SYSTEM 0x04 • ATTR_VOLUME_ID 0x08 • ATTR_DIRECTORY 0x10 • ATTR_ARCHIVE 0x20 • ATTR_LONG_NAME 0x0f Die höchsten beiden Bits sind reserviert und sollten immer 0 sein.
DIR_NTRes	12	1	Nein	Nur für Windows NT, ansonsten 0x00

DIR_CrtTimeTenth	13	1	Nein	Millisekunden des Erstellt-Zeitstempels, wobei die Granularität 2 Millisekunden sind
DIR_CrtTime	14	2	Nein	Erstellt-Zeitstempel (Stunden, Minuten, Sekunden)
DIR_CrtDate	16	2	Nein	Erstellt-Zeitstempel (Tag)
DIR_AccDate	18	2	Nein	Zugriffs-Zeitstempel (Tag)
DIR_FstClusHI	20	2	Ja	Die beiden höchsten Bytes der Clusteradresse des ersten Clusters
DIR_WrtTime	22	2	Nein	Änderungs-Zeitstempel (Stunden, Minuten, Sekunden)
DIR_WrtDate	24	2	Nein	Änderungs-Zeitstempel (Tag)
DIR_FstClusLO	26	2	Ja	Die beiden niederwertigsten Bytes der Clusteradresse des ersten Clusters
DIR_FileSize	28	4	Ja	32-Bit DWORD der Dateigröße (0 für Directories)

Tabelle 2.3.: FAT32 Directory Entry

Wenn ein Directory Entry vom Typ DIRECTORY erstellt wird, werden gleichzeitig in dem Verzeichnis, also in dem Startcluster, die sogenannten *Dot-Entries* in Form eigener Directory Entries angelegt: zeigt auf das aktuelle Verzeichnis (den aktuellen Cluster), .. auf das Elternverzeichnis (also den Cluster, der das Elternverzeichnis enthält). Dies ist z. B. wichtig, falls Cluster realloziert wurden, da anhand dieser Werte verwaiste Verzeichnisinhalte gefunden werden können.

Da mit den normalen Directory Entries (oder auch Short File Name (SFN)-Entries) nur Dateinamen im ASCII-Format und mit einer Länge von 11 Zeichen (8 im Dateinamen, 3 für die Dateiendung) gespeichert werden können, existiert die LFN-Struktur. Diese ist ein spezieller Typ von Directory Entries, der nur den Zweck hat lange Dateinamen und Sonderzeichen zu speichern. Es ist möglich mehrere LFN-Entries miteinander zu verketten, wobei diese immer vor dem zwingenden SFN-Entry gespeichert werden². LFN-Entries werden benötigt um lange Dateinamen, Groß- und Kleinschreibung und Sonderzeichen zu speichern, daher werden diese Einträge in UNICODE codiert.

Die Struktur eines LFN-Entries ist in der folgenden Tabelle dargestellt.

²Carrier hat festgestellt, dass ScanDisk keinen Fehler feststellt, wenn zu einem LFN-Entry kein normaler Directory Entry vorhanden ist (siehe [1, S. 177])

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

Name	Offset (bytes)	Größe (bytes)	Essentiell	Beschreibung
LDIR_Ord	0	1	Ja	Sequenznummer des LFN
LDIR_Name1	1	10	Ja	Zeichen 1-5 des Dateinamens (zwei Byte pro Zeichen)
LDIR_Attr	11	1	Ja	muss zwingend <code>ATTR_LONG_NAME</code> sein
LDIR_Type	12	1	Nein	0 bezeichnet ein directory entry für lange Dateinamen, alle anderen Werte werden bisher nicht genutzt
LDIR_Chksum	13	1	Ja	Prüfsumme des kurzen Dateinamens
LDIR_Name2	14	12	Ja	Zeichen 6-11 des Dateinamens (zwei Byte pro Zeichen)
LDIR_FstClusLO	26	2	Nein	Immer 0
LDIR_Name3	28	4	Ja	Zeichen 12 und 13 des Dateinamens (zwei Byte pro Zeichen)

Tabelle 2.4.: Long File Name Direcorey Entry

3. Design

In diesem Kapitel sollen die Überlegungen zum Design des Prototypen dargestellt und erläutert werden. Es wurden folgende Ziele für das zu entwickelnde Tool definiert:

Reproduzierbarkeit Das Ergebnis, also das erstellte Asservat, muss wiederholbar sein. Die Ausführung derselben Schritte muss zu demselben Asservat führen.

Überprüfbarkeit Die durchgeführten Schritte müssen auf korrekte Ausführung überprüft werden können.

Nutzbarkeit Die Definition der Schritte muss intuitiv sein, so dass diese ohne tiefere Kenntnis des Dateisystems erfolgen können.

Abstraktion Das Design sollte so gewählt werden, dass die Erweiterung und Anpassung für weitere Dateisysteme mit möglichst geringem Aufwand durchgeführt werden kann. Dabei sollte beachtet werden, dass Schritte in allen Einzelheiten definiert werden können, es aber auch möglich sein muss einzelne Parameter automatisch zu kalkulieren.

Als Kernelemente des Tools wurden folgende Komponenten identifiziert:










Dateisystemkomponente Die Software benötigt eine Komponente, die die Verwaltung der Dateisystemoperationen übernimmt. Dazu gehören im Falle von FAT32 die Schreib- und Lesezugriffe auf den Datenträger, das Verwalten der FAT-Einträge und der Clusterinhalte.

Ablaufkomponente Die Software benötigt eine Komponente, die den Ablauf zur Erstellung eines Asservats steuert. Diese sollte aufgrund späterer Erweiterbarkeit möglichst abstrakt gehalten werden. Die Konsistenz- und Erfolgsprüfung soll Teil dieser Komponente sein.

Konfigurationskomponente Diese Komponente soll die Konfiguration aller Parameter und des Ablaufs beinhalten, um diesen zu speichern und später wiederholen zu können.

Die erstellte Python-Software wird im folgenden **SyntheticDisk** (kurz: **SynDisk**) genannt und ist auf GitHub unter <https://github.com/michkoll/syntheticdisk> und der GPL-v3-Lizenz veröffentlicht. Der in dieser Hausarbeit beschriebene Versionsstand ist unter der Releasenummer 0.1 veröffentlicht (siehe <https://github.com/michkoll/syntheticdisk/tree/0.1>)

Die Struktur des Repositories ist folgende:

Repository	
 disktools	Datenträgerverwaltung, Schreibe- und Leseoperationen
 doc	Dokumentation
 filesystem	Operationen und Verwaltung des FAT-Dateisystems
 helper	Hilfsskripte zur Erstellung von Konfigurationsdateien, etc.
 yaml	
 model	Definition der Datenstrukturen
 tests	Testfälle
 util	globale Hilfsfunktionen
 workflow	Ablaufsteuerung und -definition

In den folgenden Kapiteln wird ein Überblick über den aktuellen Stand der Implementierung gegeben. Dabei werden vor allem wichtige Konzepte und Elemente vorgestellt. Auf eine detaillierte Beschreibung des Quellcodes wird aufgrund des Umfangs verzichtet. Dies kann der Dokumentation im Repository bzw. der Docstring-Dokumentation entnommen werden. Die dargestellten Klassendiagramme stellen nur einen Auszug der Klassen und deren Inhalte dar, beschränkt auf die wesentlichen Elemente.

3.1. Dateisystemkomponente

In den Dateien der Dateisystemkomponente sind alle Operationen, die das Datenträgerabbild direkt (Package **disk**) oder das Dateisystem betreffen (Package **filesystem**) implementiert. Die Dateisystemkomponente basiert im Wesentlichen auf den **FATTools** von Max Pat (siehe [2]³). Das Tool wurde auf die Besonderheiten, die für diese Arbeit benötigt werden, angepasst. Dies sind vor allem Anpassungen zur

³Das GitHub-Repository enthält keine Angaben über die Lizenz. Die Erlaubnis zur Nutzung, Modifikation und Veröffentlichung geänderter Komponenten wurde per Mail eingeholt. Grundsätzlich gelten laut dem Autor die Bedingungen der GPLv3

manuellen Steuerung von Parametern, wie den zu beschreibenden Cluster, Sektor oder FAT-Eintrag.

disktools.disk

Die Klasse **Disk** stellt den Zugriff auf ein Datenträgerabbild bereit. Dafür wurden Methoden zum Schreiben und Lesen und Hilfsmethoden wie das Setzen der aktuellen Position und Cachingmethoden implementiert.

Disk
<pre> __init__(self, filename: str, mode: str, buffering: int) seek(self, offset: int, whence: int, force: int) write(self, s: bytearray) read(self, size: int, offset: int) tell(self) </pre>

Abbildung 3.1.: KlasseDisk

Mit Hilfe der Funktionen der Klasse Disk (siehe Abbildung 3.1) ist es möglich Daten auf das Image **filename** zu schreiben oder von diesem zu lesen. Dabei muss vor einer Aktion mit Hilfe von **seek()** die Position in Bytes gesetzt werden.

filesystem.fat32

Dieses Package stellt alle für das Dateisystem relevanten Operationen bereit. Zur Erstellung oder dem Einlesen eines neuen FAT32-Dateisystems sind die Klassen **FAT32_Boot** und **FAT32FSINFO** relevant. Diese beiden Klassen enthalten die Datenstrukturen der Dateisystemstrukturen und stellen grundsätzliche Operationen zur Verfügung. Hauptsächlich werden diese Klassen, gemeinsam mit der Hilfsklasse **FAT32Creator** zum Erstellen eines neuen Images genutzt. Zum Verständnis der Funktionsweise sollten die Hinweise auf Seite 26 gelesen werden.

Das folgende Listing zeigt exemplarisch die Erstellung eines neuen FAT32-Dateisystems auf einem existierenden leeren Imagefile (siehe **helper/createFatManual.py**):

Listing 3.1: createFatManual.py - Neue FAT-Strukturen schreiben

```

1 from disktools.disk import Disk
2 from filesystem.fat32 import FAT32Creator
3
4 FILENAME = "../tests/images/Testimage.img"
5
6 # Create instance of Disk object for reading and writing access to image file
7 # The image file has to be created before

```

```

8 disk = Disk(FILENAME, 'r+b', 0)
9
10 # Creates new fat boot sector and fsinfo objects with example values and writes
11 # structures to disk
12 fat = FATCreator.mkfat32(stream=disk, size=disk.size,
13     wBytesPerSector=512, uchSectorsPerCluster=2,
14     chOemId='Test', sVolumeLabel='Demonstrate')

```

Die Verwaltung aller Operationen des FAT-Dateisystems findet in den folgenden Klassen des Packages `filesystem.fat32` statt:

FAT Implementiert die Funktionen der FAT-Tabelle, wie z. B. suchen eines freien Clusters, allozieren eines neuen FAT-Eintrags und lesende und schreibende Operationen der FAT-Tabelle

Chain Mit dieser Klasse werden Clusterchains verwaltet, wie z. B. Suchen einer zusammenhängenden Folge von freien Clustern und allozieren von Cluster und FAT-Eintrag

Handle Verwaltet einen Slot eines Directory Entries und dient als Mappingklasse zwischen Chains und Dirstable

FATDirentry Stellt das Layout eines Directory Entries (SFN und LFN) zur Verfügung

Dirstable Verwaltet mehrere Directory Entries und bietet Funktionen zum Erstellen, Ändern und Löschen neuer Dateien und Verzeichnisse. Ebenso sind Such-, Auflistungs- und Sortierfunktionen implementiert

3.2. Ablaufkomponente

Die Ablaufkomponente ermöglicht es einen Workflow zu definieren und diesen auszuführen. Mit den im nächsten Kapitel vorgestellten Konfigurationsdateien kann ein Workflow definiert und wiederholt ausgeführt werden. Alle relevanten Klassen sind im Package `workflow` enthalten.

Die Klasse `workflow.Workflow` dient als oberstes Steuerungsobjekt. Dieser Klasse werden `WorkflowSteps` hinzugefügt und der Ablauf geregelt ausgeführt. Jede konkrete Implementierung eines Schrittes muss von der abstrakten Klasse `WorkflowStep` abgeleitet werden. Die Implementierung muss die Methoden `validate()`, `execute()` und `check()` implementieren. Diese Methoden werden für jeden Schritt im Ablauf ausgeführt. Der Ablauf eines Workflows ist in der folgenden Abbildung schematisch dargestellt:

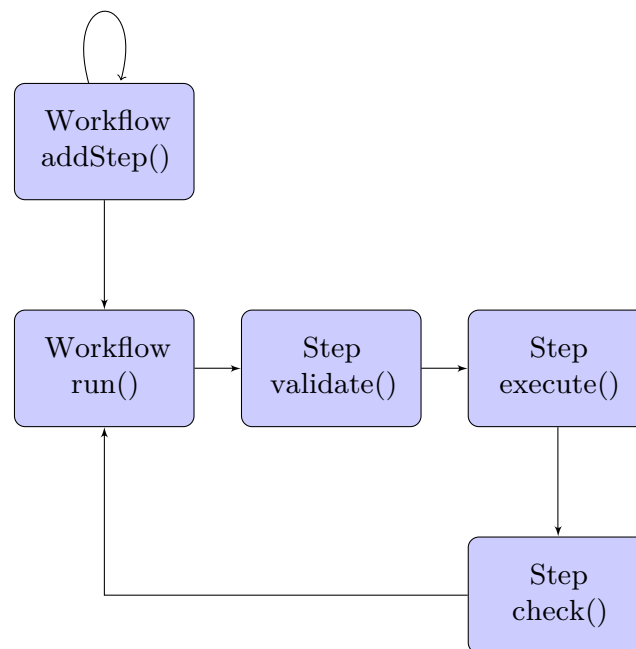


Abbildung 3.2.: Ablauf des Workflows

3.3. Konfigurationskomponente

Um den oben beschriebenen Ablauf zu steuern wurde eine Konfiguration mittels YAML und der Serialisierung der Workflowelemente gewählt. Dadurch ist es, neben dem einfachen Laden eines Workflows, diesen auch programmatisch zu erstellen und anschließend zu speichern. Dies ist exemplarisch im folgenden Beispiel dargestellt.

Listing 3.2: Workflowkonfiguration erstellen

```

1 import ...
2
3 # Initialize workflow
4 workflow = Workflow()
5
6 # Create raw writing step, writing to sector 1
7 rawStep = RawWriteStep(workflow, content=b'Testing', description="Parent",
8   position=1, positionType=PositionType.SECTOR)
9 # Create step for loading boot sector from config file and write to disk
10 fatStep = FAT32CreateBootSectorStep(workflow,
11   pathToConfig="/datadisk/Repos/github/syntheticdisc/tests/yaml/fat32.yaml")
12
13 # Adding steps to workflow
14 workflow.addStep(rawStep)
15 workflow.addStep(fatStep)
16
17 # Write workflow to yaml config file
18 yaml = ruamel.yaml.YAML()

```

```

19 with open('yaml/workflowTest.yml', 'w') as fout:
20     yaml.dump(workflow, fout)

```

Die erstellte Konfigurationsdatei ist im folgenden Listing dargestellt:

Listing 3.3: Workflow-Konfigurationsdatei

```

1 !workflow
2 steps:
3 - !rawWrite
4   description: Parent
5   position: 1
6   content: !!binary
7   | VGVzdGluZw==
8   positionType: 2
9 - !FAT32CreateBootSector
10  description: Write FAT32 BootSector
11  pathToConfig: /datadisk/Repos/github/syntheticdisc/tests/yaml/fat32.yml

```

Das Laden der Konfiguration und das Starten eines Workflows sind im folgenden Listing zu sehen:

Listing 3.4: Workflow deserialisieren und starten

```

1 def runWorkflow(wfConfig, disk):
2
3     #create disk object from path
4     disk = Disk(disk, 'r+b', 0)
5
6     # yaml factory
7     yaml = ruamel.yaml.YAML()
8
9     # deserialize workflow
10    with open(wfConfig, 'r') as fin:
11        workflow = yaml.load(fin)
12
13    # run workflow
14    workflow.run(disk)

```

Wie in Listing 3.3 zu sehen können die Parameter für die Erstellung eines neuen FAT Bootsektors ebenfalls aus einer Konfigurationsdatei geladen werden. Der Bootsektor und die FSINFO-Struktur können ebenfalls in eine YAML-Datei serialisiert werden. Ein Beispiel zur Erstellung einer Konfigurationsdatei ist in dem Skript `helper/createFat32BootYaml.py` zu finden. Die Mappingklassen für die Parameter liegen in dem `Packagemodel`.

Einige Fallbeispiele und eine Übersicht der bereits implementierten Schritte sind in Kapitel 4 dargestellt.

3.4. Wichtige Konzepte

Um ein effizientes Mapping der Datenstrukturen zu ermöglichen, wurde auf dem Konzept der FATTools von Max Pat aufgesetzt und dieses erweitert. FAT-Datenstrukturen enthalten ein Dictionarylayout, welches als Schlüsselement den Offset des jeweiligen Feldes enthält und als Wert eine Map mit dem Namen des Parameters und der Formatierung (Länge). Dieses Layout wird mittels überschriebenen `__getattr__` und `__setattr__` dazu verwendet die Parameter als Member der Klasse zu registrieren.

Das Vorgehen wird anhand der FSINFO-Struktur demonstriert.

Listing 3.5: FSINFO `__init__`

```

1 def __init__(self, s = None, offset = 0, stream: Disk = None):
2     logging.debug("Init FAT32 FSINFO")
3     self._i = 0
4     self._pos = offset
5     self._buf = s or bytearray(512)
6     self.stream = stream
7
8     # Copy layout to dict_kv
9     self._kv = self.layout.copy()
10
11    # Copy layout to dict_vk
12    # with values holdung { name: offset }
13    self._vk = {}
14    for k, v in self._kv.items():
15        self._vk[v[0]] = k
16    # Maps layout and buffer s to member variables
17    getattr(self, v[0])

```

In der `__init__`-Methode wird das Layout-Dictionary in das Dictionary `_vk` überführt, wobei dieses die Werte { Name: Offset} enthält. Anschließend wird für jeden Parameter die überschriebene `getattr`-Methode aufgerufen. Essentiell ist es, dass der Buffer `_buf` durch `s` gefüllt wird, ansonsten werden nur Null-Werte hinterlegt.

Listing 3.6: common `__getattr__`

```

1 def common_getattr(c, name):
2     '''
3     Decodes parameter layout to class member variables
4
5     :param c: Class object
6     :type c: object
7     :param name: Name of paramter
8     :type name: str
9     :return:
10    :rtype:

```

```

11  """
12  # Get offset for parameter
13  i = c._vk[name]
14  # Get format for parameter
15  fmt = c._kv[i][1]
16  # Unpack parameter value as bytearray from buffer
17  cnt = struct.unpack_from(fmt, c._buf, i + c._i)[0]
18  # Sets attribute to object
19  setattr(c, name, cnt)
20  return cnt

```

Die überschriebene `getattr`-Methode liest nun mit Hilfe des Offsets `i` und des Formats `fmt` den Wert des Parameters aus dem Buffer `_buf`. Anschließend wird der Parameter über die `setattr`-Methode als Klassenvariable definiert.

Ein weiteres wichtiges Konzept ist das Packen der Parameter in `struct`-Objekte⁴. Um eine manuelle Transformation von Datenstrukturen in Byteobjekte zu vermeiden enthält jede Klasse einer Datenstruktur eine `pack`-Methode, die die Klassenvariablen in den Buffer `_buf` transformiert. Anschließend ist es möglich diesen Buffer in das Imagefile zu schreiben.

Listing 3.7: pack

```

1  def pack(self):
2  "Update internal buffer"
3  # Iterate over layout dict in _kv
4  for k, v in self._kv.items():
5      # Update _buf at calculated position
6      self._buf[k:k+struct.calcsize(v[1])] = struct.pack(v[1], getattr(self, v[0]))
7  return self._buf

```

3.5. Implementierte UseCases

Im folgenden werden die bereits implementierten UseCases bzw. Steps vorgestellt. Dabei werden vor allem die verschiedenen Möglichkeiten der Parametrisierung beschrieben. Der Parameter `description` kann für alle Schritte definiert werden und dient als beschreibende Zeichenkette in der Konfiguration.

3.5.1. RawWrite

Der `rawSteps.RawWriteStep` stellt eine einfache Möglichkeit dar eine Zeichenkette an eine bestimmte Position des Datenträgers zu schreiben. Dabei wird keine Prüfung

⁴<https://docs.python.org/3.7/library/struct.html>

von vorhandenen Einträgen durchgeführt. Dadurch ist es möglich z. B. SlackSpace auszunutzen. Da keine Prüfung durchgeführt wird ist es allerdings auch möglich die essentiellen FAT-Strukturen zu überschreiben und das Dateisystem unbrauchbar zu machen.

Name	Essentiell	Typ	Beschreibung
content	Ja	Bytearray	Inhalt der auf den Datenträger geschrieben werden soll
position	Nein	int	Position an der geschrieben werden soll. (In Kombination mit <code>positionType</code>). Standarwert: 0
positionType	Nein	int	1 : Byte (Standard), 2 : Sektor

Tabelle 3.1.: RawWriteStep Parameter

3.5.2. CreateImage

Der `diskSteps.CreateImageStep` ist ein Hilsschritt zur Vorbereitung des Zielimages. Dazu gibt es die Möglichkeit entweder ein vorhandenes Image zu kopieren oder ein neues Image mit einer definierten Größe zu erstellen.

Name	Essentiell	Typ	Beschreibung
srcDisk	Nein	String	Pfad zum Quellimage, falls eine Kopie angelegt werden soll
destDisk	Ja	String	Pfad zum Zielimage
diskSize	Nein	int	Größe des zu erstellenden Images in Bytes

Tabelle 3.2.: CreateImageStep Parameter

Die Kombination der Parameter bestimmt die Art und Weise der Ausführung. Falls `srcDisk` nicht gesetzt ist, wird anhand von `destDisk` und `diskSize` ein neues Image erstellt. Falls `srcDisk` und `destDisk` definiert sind, wird der Kopiermodus verwendet.

3.5.3. FAT32CreateBootSector

Dieser Schritt erstellt anhand einer vorhandenen Bootsektor-Konfigurationsdatei eine neue FAT-Struktur. Dabei wird sowohl der Bootsektor, die FSInfo-Struktur und eine neue FAT erstellt.

Name	Essentiell	Typ	Beschreibung
pathToConfig	Ja	String	Pfad zur Konfigurationsdatei des Bootsektors

Tabelle 3.3.: FAT32CreateBootSectorStep Parameter

3.5.4. CreateDir

Der `fatSteps.CreateDirStep` ermöglicht das Anlegen eines neuen Verzeichnisses auf dem Datenträger. Dabei werden verschiedene Möglichkeiten zur Steuerung und Definition der Metadaten und Attribute angeboten.

Name	Essentiell	Typ	Beschreibung
fullpath	Nein	String	vollständiger Pfad des anzulegenden Verzeichnisses, <code>parentDir</code> und <code>dirName</code> werden aus diesem erstellt
parentDir	Nein	String	vollständiger Pfad zum Elternverzeichnis. Falls nicht definiert, wird das Verzeichnis im Root-Directory erstellt.
dirName	Ja	String	Name des Verzeichnisses
deleted	Nein	String	Markiert das Verzeichnis als gelöscht. Dabei wird nur das erste Zeichen des Directory Entries verändert. Die Inhalte bleiben erhalten, sofern sie nicht durch weitere Vorgänge realloziert werden.

Tabelle 3.4.: CreateDirStep Parameter

3.6. Fehler und Erweiterungen

Da der aktuelle Stand des Tools nur ein Prototyp ist werden im Folgenden aktuell bekannte Probleme erläutert und Ideen zur Weiterentwicklung aufgeführt.

Master Boot Record und Partitionstabelle Derzeit ist es nur möglich ein Image mit einem FAT32-Dateisystem zu erzeugen, ohne Master Boot Record bzw. einer Partitionstabelle. Es ist zwar möglich ein Offset für das Dateisystem zu hinterlegen, dies konnte aber nicht mehr ausführlich getestet werden, weshalb vorerst nur Images unterstützt werden, die ausschließlich ein FAT32-Dateisystem enthalten

FAT2 Aktualisierung Derzeit wird die Backupkopie der FAT-Tabelle nicht aktualisiert. Dazu muss für eine weitere Entwicklung entschieden werden, ob die Aktualisierung nach der Spezifikation erfolgen soll oder z. B. konfiguriert werden kann, dass die Kopie andere Einträge enthält, als die Originaltabelle (siehe UseCase 18)

Exception- und Fehler-Handling Aufgrund von Zeitmangel konnte kein umfassendes Fehlerhandling implementiert werden. Dies äußert sich z. B. bei der Übergabe von Parametern des falschen Typs. Es wurde versucht diese Fehler soweit wie möglich abzufangen, indem eine Typkonvertierung durchgeführt wird. Hier müsste in einer weiteren Entwicklung ein gesamtheitliches und einheitliches Konzept implementiert werden.

Ablaufdefinition Ursprünglich war es geplant das gesamte Ablaufkonzept als Ansible-Modul zu implementieren. Das Modul sollte dann im Gegensatz zum normalen Ablauf von Ansible immer auf dem Host operieren und mittels der implementierten Dateisystemschnittstelle das Image beschreiben. Der Vorteil dieser Lösung wäre gewesen, dass die gesamte Ablaufkonfiguration und -durchführung auf das bewährte Ansible-Konzept gesetzt hätte und damit das Fehlerhandling und die robuste Konfiguration von Ansible genutzt werden könnte. Die Einarbeitung in die Ansible-Sourcen und die Implementierung eines eigenen Moduls waren innerhalb des Zeitrahmens allerdings nicht möglich, sodass eine eigenständige Ablaufkonfiguration implementiert wurde. Diese Refactoringmaßnahme wäre für zukünftige Entwicklungsschritte eine umfassende aber sinnvolle Anpassung.

Abstraktion des Dateisystems Derzeit greifen die dateisystemabhängigen Ablaufschritte direkt auf die Dateisystemkomponente, also die FAT-Komponente zu. Hier wäre es möglich eine weitere Abstraktionsschicht zu implementieren, damit die Ablaufschritte unabhängig vom Dateisystem konfiguriert werden können. Die Vorstellung ist, dass in einer zukünftigen Implementierung z. B. nur ein `CreateFileStep` existiert und ein unterliegendes Interface je nach gewähltem Dateisystem die korrekte Komponente verwendet.

Automatische Validierung Es wurden Vorbereitungen im Ablauf zur automatischen Erfolgskontrolle der einzelnen Schritte vorgenommen, um sichergehen zu können, dass die Daten wie konfiguriert auf dem Datenträger gespeichert sind. Die einzelnen Validierungsschritte wurden allerdings noch nicht implementiert. Vorgesehen ist, dass die Validierung jedes einzelnen Schritts erst zum Ende des Gesamtablaufs getätigt wird. Dabei müssen allerdings Abhängigkeiten betrachtet werden, wie z. B. die Reallokation eines vorher erstellten und

gelöschten Directory Entries. Eine endgültige Lösung für dieses Problem konnte bis zu diesem Punkt nicht entwickelt werden.

Detaillierte Konfiguration Da es für die Erstellung eines Asservats unabdingbar ist jedes Detail konfigurieren zu können, müssen die Konfigurationsmöglichkeiten deutlich erweitert werden. So muss es z. B. möglich sein konkrete Directory Entries auszuwählen und entweder den zugehörigen FAT-Eintrag, den Directory Entry selbst oder das Datencluster zu manipulieren. Dabei kann das Dateisystem aber in einen inkonsistenten Zustand geraten der darauffolgende Schritte, wie z. B. das Allokieren eines Clusters nicht mehr ermöglicht oder ungewünschte Ergebnisse liefert. Dieses Verhalten könnte entweder durch eine wohlüberlegte Reihenfolge der Ablaufschritte oder durch eine parallele Speicherung der korrekten Werte ermöglicht werden, so dass immer ein Referenzsystem und ein Zielsystem existieren, damit ein reibungsloser Ablauf gewährleistet werden kann.

4. Fallbeispiele

In diesem Kapitel soll die praktische Anwendung des SyntheticDisc-Tools präsentiert werden. Dazu wird als Erstes die Installation und Basiskonfiguration vorgestellt (siehe Kapitel 4.1). Anschließend werden im zweiten Beispiel (siehe Kapitel 4.2) die wichtigsten Datei- und Verzeichnisoperationen demonstriert. Das letzte Beispiel (Kapitel 4.3) demonstriert das Einlesen und Verändern eines existierenden Images und die dabei zu beachtenden Besonderheiten.

Alle Fallbeispiele wurden als Musterfälle im Repository hinterlegt und können dort nachvollzogen werden. Die Images wurden aus Platzgründen nicht im Repository gespeichert.

4.1. Installation und Setup

Um eine fallbezogene Erstellung von Asservaten zu ermöglichen und die Bedienung zu vereinfachen wurden einige Hilfsskripte implementiert. Mit diesen ist es möglich eine definierte Ordnerstruktur für einen *Fall* anzulegen, inklusive der benötigten Konfigurationsdateien und Skripte.

Als Erstes wird ein Projektordner angelegt, in den der Inhalt des Repositories geklont wird. Anschließend wird SyntheticDisk in der virtuellen Pythonumgebung installiert.

Listing 4.1: Installation

```
1 mkoll:/$ mkdir SynTest
2 mkoll:/$ cd SynTest
3 # Clone repository
4 mkoll:SynTest$ git clone git@github.com:michkoll/syntheticdisk.git
5 mkoll:SynTest$ cd syntheticdisk
6 # Virtuelle Umgebung aktivieren
7 mkoll:syntheticdisk$ source venv/bin/activate
8 # Syntheticdisk in der Venv-Umgebung installieren/aktualisieren
9 mkoll:syntheticdisk$ pip3 install .
10 Processing /datadisk/SynTest/syntheticdisk
11 Installing collected packages: syntheticdisc
12 Found existing installation: syntheticdisc 0.1
13 Uninstalling syntheticdisc-0.1:
```

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

```
14 Successfully uninstalled syntheticdisc-0.1
15 Running setup.py install for syntheticdisc ... done
16 Successfully installed syntheticdisc-0.1
17 # Überprüfen der installierten Packages
18 mkoll:syntheticdisc$ pip3 list
19 Package            Version
20 -----
21 pip                 10.0.1
22 ruamel.yaml         0.15.64
23 setuptools          39.1.0
24 syntheticdisc       0.1
```

Anschließend kann, unter Angabe des Elternverzeichnisses und dem Fallnamen ein neuer Fall angelegt werden. Die erstellte Ordner- und Dateistruktur ist im folgenden Listing dargestellt.

Listing 4.2: Neuen Fall anlegen

```
1 # Neuen Fall ex-setup im Ordner cases anlegen
2 mkoll:syntheticdisc$ python3 createNewCase.py cases ex-setup
3 mkoll:syntheticdisc$ tree cases/
4 cases/
5   ex-setup
6     conf
7       config.yml
8     disk
9     files
10    log
11    prepare.py
12    run.py
13    scripts
14      createFat32BootYaml.py
15      createWorkflowYaml.py
16      __init__.py
17      mkimage.py
18      runWorkflow.py
19
20 6 directories, 8 files
```

Für die Konfiguration des Falls sind die beiden Skripte `createFat32BootYaml.py` und `createWorkflowYaml.py` essentiell. Es ist zwar möglich die Konfigurationsdateien manuell zu erstellen, mit den beiden Skripten ist dies aber deutlich nutzerfreundlicher. Dazu müssen die beiden Skripte nach den Wünschen angepasst werden. Anschließend werden diese mit Hilfe des Skripts `prepare.py` als Konfiguration für den Fall hinterlegt.

In der Datei `createFat32BootYaml.py` werden die einzelnen Parameter für den Bootsektor und die FSInfo-Struktur hinterlegt. Dabei kann jeder Wert einzeln kon-

figuriert werden. Es ist allerdings zu beachten, dass keine Validierung der Werte stattfindet. Angaben, die zu einem korrupten Dateisystem führen, können ebenfalls eingetragen und auf den Datenträger geschrieben werden. Es ist ratsam fehlerhafte Bootsektoren als letzten Schritt auf den Datenträger zu schreiben.

Die Erstellung der Ablaufkonfiguration ist mit Hilfe des Skripts `createWorkflowYaml.py` möglich. Dabei werden die einzelnen Schritte erstellt und konfiguriert und anschließend einem Workflow-Objekt hinzugefügt. Das folgende Listing zeigt den konfigurierten Workflow für den Fall `ex-setup`.

Listing 4.3: Workflowkonfiguration ex-setup

```

1 def createWorkflowYaml(yamlPath):
2     # Initialize workflow
3     workflow = Workflow()
4
5     # EXAMPLE AREA - DELETE BEFORE EXECUTION
6     # Create RawWriteStep
7     rawStep = RawWriteStep(workflow, content=b'Testing', description="Write test
→ string", position=1, positionType=PositionType.SECTOR) ←
8     # Create step for loading boot sector from config file and write to disk
9     fatStep = FAT32CreateBootSectorStep(workflow, ←
→ pathToConfig=os.path.join(os.path.dirname(yamlPath), "fat32.yml"))
10
11     # Adding steps to workflow
12     workflow.addStep(rawStep)
13     workflow.addStep(fatStep)
14     # EXAMPLE AREA - DELETE BEFORE EXECUTION
15
16     # Write workflow to yaml config file
17     yaml = ruamel.yaml.YAML()
18     with open(yamlPath, 'w') as fout:
19         yaml.dump(workflow, fout)

```

Nachdem die gewünschte Konfiguration in den beiden Skripten hinterlegt wurde können mit der Ausführung des Skripts `prepare.py` die Konfigurationsdateien erstellt werden. Zusätzlich wird, je nach Konfiguration in der Datei `conf/config.yml` ein leeres Image erstellt oder ein vorhandenes Image kopiert. Die Parameter in der Datei `config.yml` müssen je nach Fall angepasst werden. Standardmäßig wird ein leeres Image mit 256MB Größe angelegt.

Listing 4.4: Konfigurationsdateien erstellen

```

1 mkoll:~$ cd cases/ex-setup/
2 mkoll:ex-setup$ python3 prepare.py conf/config.yml
3 2018-09-27 14:38:17,155 [MainThread ] [INFO ] Created image with size 268435456: ←
→ /datadisk/SynTest/syntheticdisk/cases/ex-setup/disk/dest.img
4 2018-09-27 14:38:17,157 [MainThread ] [INFO ] Created workflow configuration: ←
→ /datadisk/SynTest/syntheticdisk/cases/ex-setup/conf/workflow.yml

```

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

```
5 2018-09-27 14:38:17,168 [MainThread ] [INFO ] Created boot sector config in path: ←
./datadisk/SynTest/syntheticdisk/cases/ex-setup/conf
6 2018-09-27 14:38:17,168 [MainThread ] [INFO ] Finished case preparation
```

Der Fall ist nun vorbereitet und kann wie im folgenden Listing dargestellt ausgeführt werden. Dabei werden alle konfigurierten Schritte auf dem Zieldatenträger ausgeführt.

Listing 4.5: Ablauf ausführen

```
1 mkoll:ex-setup$ python3 run.py conf/config.yml
```

In dem hier vorgestellten Workflow wird nur ein neues Image erstellt und dieses mit den konfigurierten Parametern mit einem FAT32-Dateisystem beschrieben. Das Ergebnis kann entweder manuell mit einem Hex-Editor oder mit dem Tool `fsstat` überprüft werden:

Listing 4.6: Ex-Setup fsstat Ausgabe

```
1 mkoll:ex-setup$ fsstat -f fat32 disk/dest.img
2 FILE SYSTEM INFORMATION
3 -----
4 File System Type: FAT32
5
6 OEM Name: EX-SETUP
7 Volume ID: 0x499602d2
8 Volume Label (Boot Sector): M111
9 Volume Label (Root Directory):
10 File System Type Label: FAT32
11 Next Free Sector (FS Info): 4130
12 Free Sector Count (FS Info): 524254
13
14 Sectors before file system: 0
15
16 File System Layout (in sectors)
17 Total Range: 0 - 524287
18 * Reserved: 0 - 31
19 ** Boot Sector: 0
20 ** FS Info Sector: 1
21 ** Backup Boot Sector: 6
22 * FAT 0: 32 - 2079
23 * FAT 1: 2080 - 4127
24 * Data Area: 4128 - 524287
25 ** Cluster Area: 4128 - 524287
26 *** Root Directory: 4128 - 4129
27
28 METADATA INFORMATION
29 -----
30 Range: 2 - 8322566
31 Root Directory: 2
```

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

```
32
33 CONTENT INFORMATION
34 -----
35 Sector Size: 512
36 Cluster Size: 1024
37 Total Cluster Range: 2 - 260081
38
39 FAT CONTENTS (in sectors)
40 -----
41 4128-4129 (2) -> EOF
```

4.2. Dateioperationen

In dem zweiten Beispiel wird auf einem neuen Image ein FAT32-Dateisystem angelegt und anschließend verschiedene Dateioperationen durchgeführt. Anhand dieses Beispiels sollen die aktuellen Möglichkeiten zur Ablaufsteuerung demonstriert werden. Die Erstellung des Cases und die Konfiguration wurde wie im vorherigen Kapitel durchgeführt, der Fall ist im Repository unter dem Pfad `cases/ex-diroperation` abgelegt.

Die erstellte Datei- und Verzeichnisstruktur sieht nach Ausführung des Workflows folgendermaßen aus:

Listing 4.7: Verzeichnisstruktur ex-diroperation

```
1 mkoll:ex-diroperation $ ls -r -f fat32 disk/dest.img
2 d/d 4: Parent
3 + d/d 38: ChildTime
4 ++ r/r 133: time.txt
5 + d/d 40: ChildDel
6 ++ d/d 229: nodelete
7 ++ d/d * 230: _elete
8 + d/d 42: ChildRealloc
9 ++ r/r 293: newfile.txt
10 ++ r/r * 295: LongDeletedFi
11 d/d 5: short
12 + r/r 69: content.txt
13 d/d 8: ThisIsALongDirName
14 v/v 8322563: $MBR
15 v/v 8322564: $FAT1
16 v/v 8322565: $FAT2
17 V/V 8322566: $OrphanFiles
```

Um die unterschiedlichen Möglichkeiten zu demonstrieren wurde der Inhalt der Datei `content.txt` aus der Datei `files/content.txt` kopiert (Attribut: `contentFile`).

Im Gegensatz dazu wurde der Inhalt der Datei `time.txt` direkt als Bytearray übergeben (Attribut: `content`). Dadurch ist es möglich eine Workflowkonfiguration zu erstellen, die rekursiv ein gegebenes Verzeichnis durchläuft und die Inhalte auf das Zielimage überträgt. Zusätzlich dazu können die Metadaten wie im folgenden Beispiel gezeigt angepasst werden.

Wie in UseCase 9 beschrieben ist es ein forensischer Anwendungsfall Zeitstempel zu manipulieren, um die Timeline eines Asservats zu verändern. In diesem Beispiel liegt der Zeitstempel der letzten Änderung vor dem Erstellt-Zeitstempel, was auf eine Manipulation hindeuten würde.

Listing 4.8: CreateDirStep: Zeitstempel manipulieren

```
1 childTimeDir = CreateDirStep(workflow, fullPath="/Parent/ChildTime",
2   description="Create subdir for time manipulation")
3 fileTime = CreateFileStep(workflow, fullPath="/Parent/ChildTime/time.txt",
4   cDate="2000-01-01 12:00:00",
5   mDate="2000-01-01 11:00:00", aDate="2000-01-01 00:00:00", content=512*'Time')
6 workflow.addStep(childTimeDir)
7 workflow.addStep(fileTime)
```

Das Ergebnis kann z. B. mit `istat` überprüft werden

Listing 4.9: Ergebnis: Zeitstempel manipulieren

```
1 mkoll:ex-diroperations$ istat -f fat32 disk/dest.img 133
2 Directory Entry: 133
3 Allocated
4 File Attributes: File, Archive
5 Size: 2048
6 Name: time.txt
7
8 Directory Entry Times:
9   Written:  2000-01-01 11:00:00 (CET)
10  Accessed: 2000-01-01 00:00:00 (CET)
11  Created:  2000-01-01 12:00:00 (CET)
12
13 Sectors:
14 4138 4139 4140 4141
```

Ein weiterer forensischer Anwendungsfall ist File- oder Clusterslack, welcher in UseCase 13 beschrieben wurde. Dabei werden entweder gezielt Daten hinter den Daten abgelegt, die einer Datei zugeordnet sind oder ein Cluster wird nur zum Teil durch eine Reallokation überschrieben. Der zweite Fall wird in den folgenden Listings dargestellt:

Listing 4.10: CreateDirStep: Zeitstempel manipulieren

```

1 childRealDir = CreateDirStep(workflow, fullPath="/Parent/ChildRealloc",
  ↳ description="Create subdir for reallocation")
2 sectorContent = 512 * 'A' + 512 * 'B' + 512 * 'C'
3 deletedFile = CreateFileStep(workflow,
  ↳ fullPath="/Parent/ChildRealloc/LongDeletedFileName.txt",
4   content=sectorContent, deleted=True description="Create file for deletion")
5 newFile = CreateFileStep(workflow, fullPath="/Parent/ChildRealloc/newfile.txt",
6   description="Create reallocation file", content="Das ist die Datei, die den
  ↳ Cluster neu belegt.")
7 workflow.addStep(childRealDir)
8 workflow.addStep(deletedFile)
9 workflow.addStep(newFile)

```

Im Ergebnis sieht man, dass der erste LFN-Eintrag der `LongDeletedFileName.txt` durch die neue Datei `newfile.txt` überschrieben wurde, der Dateiname kann nicht mehr vollständig hergestellt werden. Da der Directory Entry für die Datei aber erhalten geblieben ist, kann die Information über den vormals zugeordneten Start-Cluster weiterhin gelesen werden. Allerdings wurde der zugehörige FAT-Eintrag durch die neue Datei überschrieben, daher ist eine Zuordnung weiterer Cluster nicht mehr möglich. Die verbliebenen Inhalte müssten manuell gesucht werden.

Listing 4.11: istat: LongDeletedFileName.txt

```

1 mkoll:ex-diropertions$ istat -f fat32 disk/dest.img 295
2 Directory Entry: 295
3 Not Allocated
4 File Attributes: File, Archive
5 Size: 1536
6 Name: _ONGDE~1.TXT
7
8 Directory Entry Times:
9   Written: 2018-09-27 19:27:02 (CEST)
10  Accessed: 2018-09-27 00:00:00 (CEST)
11  Created: 2018-09-27 19:27:02 (CEST)
12
13 Sectors:
14 4148 4149

```

Listing 4.12: istat: LongDeletedFileName.txt

```

1 mkoll:ex-diropertions$ istat -f fat32 disk/dest.img 293
2 Directory Entry: 293
3 Allocated
4 File Attributes: File, Archive
5 Size: 46
6 Name: newfile.txt
7
8 Directory Entry Times:
9   Written: 2018-09-27 19:27:02 (CEST)

```

Entwicklung eines Tools zur syntethischen Erzeugung von FAT32-Asservaten

In Abbildung 4.1 ist zu sehen, dass der FAT-Eintrag des Clusters 12 (Sektor 4148+4149) bereits durch die neue Datei mit einer EOF-Markierung versehen wurde. Cluster 13 ist laut FAT-Tabelle nicht zugeordnet.

[illegible]

Nichtsdestotrotz können in den Clustern 12 und 13 die Reste der `LongDeletedFileName.txt` gefunden werden, da die neue Datei einen deutlich kürzeren Inhalt hat, siehe Abbildung 4.2. Der blaue Bereich markiert die Daten von `newfile.txt`. Der grüne und rote Bereich sind die Cluster 12 und 13 mit den nicht zugeordneten, aber noch vorhandenen Daten von `LongDeletedFileName.txt`.

[illegible]

© Michael Koll

4.3. Existierendes Dateisystem

In diesem Beispiel soll demonstriert werden, wie ein existierendes Image eingelesen und manipuliert werden kann. Das Vorgehen unterscheidet sich nicht grundsätzlich von den vorherigen Beispielen. Da aber z. B. die EOF-Markierung nicht fest definiert ist, muss beim Einlesen eines vorhandenen Images dieses vorkonfiguriert werden. Standardmäßig ist dieser Wert auf `0xffff ffff` gesetzt.

Für dieses Beispiel wird das Image `ueb-fat32.dd` aus dem Studienbrief 3 genutzt. Dieses wird, nach Anlage eines neuen Falles `ex-existing` in den Ordner `disk` kopiert und als `src.img` abgelegt⁵.

Abbildung 4.3.: Auszug FAT ueb-fat32.dd

Wie in Abbildung 4.3 an den Markierungen zu sehen sind die EOF-Markierungen in diesem Fall auf den Wert `0xffff ffff` definiert. Dieser Parameter wird auf Ebene des Workflows definiert, da dieser Wert für jeden Schritt essentiell ist. In Listing 4.15 wird der Wert `workflow.fatLast` definiert, dies ist ebenso möglich für `workflow.fatReserved` und `workflow.fatBad`.

Listing 4.13: CreateWorkflowYaml.py: EOF-Markierung setzen

```
1 def createWorkflowYaml(yamlPath):
2     # Initialize workflow
3     workflow = Workflow()
4     workflow.fatLast = 0xFFFFFFFF
5
6     copyStep = CreateImageStep(workflow, srcDisk="disk/src.img",
7     destDisk="disk/dest.img")
8     workflow.addStep(copyStep)
9
10    delStep = CreateFileStep(workflow, fullPath="/Kontakte/besenfelder.txt",
11    deleted=True)
12    workflow.addStep(delStep)
```

Die erstellte Konfigurationsdatei enthält anschließend den definierten Wert in der letzten Zeile.

⁵Falls die Datei an einem anderen Ort oder unter einem anderen Dateinamen gespeichert ist muss dies in der `config.yml` angepasst werden

SYNTHETISCH ERZEUGTES DATEISYSTEM

Entwicklung eines Tools zur synthetischen Erzeugung von FAT32-Asservaten

Listing 4.14: fat32.yml: EOF-Markierung setzen

```
1 !workflow
2 steps:
3 - !createImage
4   description: Create new blank image file
5   srcDisk: disk/src.img
6   destDisk: disk/dest.img
7   diskSize: 0
8 - !createFile
9   description: Default description
10  fullPath: /Kontakte/besenfelder.txt
11  parentDir:
12  fileName:
13  deleted: true
14  mDate:
15  cDate:
16  aDate:
17  content:
18  contentFile:
19 fatLast: 268435455
```

Anschließend kann der Workflow wie gewohnt ausgeführt werden. In diesem Beispiel wird die Datei `/Kontakte/besenfelder.txt` gelöscht. Folgendes Listing zeigt die Metadaten der Datei, an denen zu sehen ist, dass diese nicht mehr zugeordnet ist.

Listing 4.15: Ergebnis: EOF-Markierung setzen

```
1 mkoll:ex-existing$ stat -f fat32 disk/dest.img 42
2 Directory Entry: 42
3 Not Allocated
4 File Attributes: File, Archive
5 Size: 0
6 Name: _ESENF~1.TXT
7
8 Directory Entry Times:
9 Written: 2018-09-28 08:12:36 (CEST)
10 Accessed: 2018-09-28 00:00:00 (CEST)
11 Created: 2018-09-28 08:12:36 (CEST)
12
13 Sectors:
14 8188
```

5. Fazit

Im Rahmen dieser Hausarbeit wurde das Tool SyntheticDisk entwickelt, um die unzähligen Möglichkeiten zum Hinterlegen von forensisch relevanten Artefakten auf einem FAT32-Datenträger wiederholbar und konsistent zu ermöglichen. Die Schwierigkeit bei der Umsetzung war es vor allem das richtige Maß an Konfigurierbarkeit und Bedienbarkeit zu finden. Um alle vorgestellten UseCases umzusetzen bedarf es einen tiefen Eingriff in das Dateisystem. Das manuelle Überschreiben oder Manipulieren bestimmter Bereiche kann zu einem inkonsistenten Zustand führen bei dem es nicht mehr möglich ist spezifizierte FAT-Operationen durchzuführen. Der aktuelle Entwicklungsstand stellt ein Rahmenwerk dar, um solch ein Tool zu entwickeln. Es wurden einige Basisoperationen zur Demonstration implementiert und die wichtigsten Abstraktionsschichten zur Erweiterung des Tools erstellt.

Essentiell für eine weitere Entwicklung wäre die automatische Kontrolle des Endergebnisses. Da die hintereinander durchgeführten Operationen häufig Einfluss aufeinander haben, hat die Kontrolle der Einzelschritte keine hinreichende Aussagekraft über das Endergebnis. Diese Validierung konnte nicht mehr erarbeitet werden, allerdings wurden Vorbereitungen für diese in der Ablaufsteuerung implementiert.

Literatur

- [1] Brian Carrier. *File system forensic analysis*. eng. 10. print. Carrier, Brian (VerfasserIn). Upper Saddle River, NJ: Addison-Wesley, 2011. 569 S. ISBN: 0-32-126817-2.
- [2] maxpat78. *FATtools. Facilities to read and write FAT filesystems with Python*. Version 6a7d815121a32de39fce9b7c7833f81eb9aad9c8. maxpat78URL: %5Curl%7Bhttps://github.com/maxpat78/FATtools%7D (besucht am 24. 08. 2018).
- [3] Microsoft Corporation. *Microsoft Extensible Firmware Initiative FAT32 File System Specification. FAT: General Overview of On-Disk Format*. Hrsg. von Microsoft Corporation. Version 1.03. 2000.

Verzeichnis der Listings

3.1. createFatManual.py - Neue FAT-Strukturen schreiben	22
3.2. Workflowkonfiguration erstellen	24
3.3. Workflow-Konfigurationsdatei.	25
3.4. Workflow deserialisieren und starten.	25
3.5. FSINFO __init__	26
3.6. common_getattr	26
3.7. pack	27
4.1. Installation	32
4.2. Neuen Fall anlegen	33
4.3. Workflowkonfiguration ex-setup	34
4.4. Konfigurationsdateien erstellen	34
4.5. Ablauf ausführen	35
4.6. Ex-Setup fsstat Ausgabe	35
4.7. Verzeichnisstruktur ex-diroperation	36
4.8. CreateDirStep: Zeitstempel manipulieren	37
4.9. Ergebnis: Zeitstempel manipulieren	37
4.10. CreateDirStep: Zeitstempel manipulieren	38
4.11. istat: LongDeletedFileName.txt.	38
4.12. istat: LongDeletedFileName.txt.	38
4.13. CreateWorkflowYaml.py: EOF-Markierung setzen.. . . .	40
4.14. fat32.yml: EOF-Markierung setzen	41
4.15. Ergebnis: EOF-Markierung setzen	41

Anwendungsfälle

1.	Manipulieren der Strukturinformationen	7
2.	Manipulieren von Zeichenketten	8
3.	Bootcode-Slack	8
4.	Reservierter-Bereich-Slack	8
5.	Position des Wurzelverzeichnisses.	8
6.	Allokationsstatus eines Clusters ändern	9
7.	FAT-Slack	9
8.	Volume- oder Partitionsslack	9
9.	Ändern der Zeitstempel	10
10.	Allokationsstatus eines Directory Entry ändern	10
11.	Reallokation von Entry Adressen	11
12.	Verändern der Clusterchain	11
13.	Clusterslack	11
14.	Volume-Label-Clusterchain	12
15.	Verstecken von Daten im Bootsektor.	15
16.	FSInfo-Slack	16
17.	FSInfo-Manipulation	16
18.	FAT Backupkopie als echte Referenz.	17

Abbildungsverzeichnis

2.1. Zusammenspiel der FAT-Datenstrukturen.	6
2.2. Physikalischer Aufbau des FAT-Dateisystems.	6
3.1. Klasse <code>Disk</code>	22
3.2. Ablauf des Workflows	24
4.1. FAT-Tabelle nach Ablauf	39
4.2. Inhalt Cluster 12 und 13	39
4.3. Auszug FAT ueb-fat32.dd	40

Tabellenverzeichnis

2.1. Datenstruktur des Bootsektors eines FAT32-Dateisystems	14
2.2. FAT32 FSInfo-Struktur	15
2.3. FAT32 Direcorey Entry	18
2.4. Long File Name Direcorey Entry	19
3.1. RawWriteStep Parameter	28
3.2. CreateImageStep Parameter	28
3.3. FAT32CreateBootSectorStep Parameter.	29
3.4. CreateDirStep Parameter	29

A. Anhang

A.1. Aufgabenstellung

Thema 15: Synthetisch erzeugtes Dateisystem

T: Ziel ist es, Asservate zu erzeugen, deren Daten und Metadaten konsistent und unter voller Kontrolle eines Anwenderprogramms erzeugt werden. Gedanklich soll ein einfaches Dateisystem, wie z. B. FAT32 angelegt werden, indem die Daten des Dateisystems auf einen Datenträger gebracht werden, indem von einem Anwendungsprogramm direkt die entsprechenden logischen Sektoren beschrieben werden. Damit soll es z. B. möglich sein, gezielt Inhaltsdaten und Metadaten mit den gewünschten Werten abzulegen. Dazu ist es notwendig, die Mechanismen und Funktionsweise des Dateisystems vorab zu beschreiben.

P: Stellen Sie dar, wie Sie sich so ein Anwenderprogramm vorstellen:

- Konfigurationsdatei zur Ablage der Daten für gewünschte Daten und Metadaten
- Ablaufsteuerung zum Anlegen von Systemdaten (z. B. FAT, MBR) und Nutzerdaten
- Konsistenzprüfung des Resultats

Zeigen Sie experimentell anhand eines Datenträgerimages, wie Sie die Konsistenzprüfung durchführen würden. Implementieren Sie das Anwenderprogramm (ggf. nur in Auszügen).