FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Btrfs Filesystem Forensics

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Andreas Juch, Bakk. techn.
Matrikelnummer 0460615

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Martin Mulazzani, M.Sc.

Wien, 17.03.2014

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Btrfs Filesystem Forensics

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Andreas Juch, Bakk. techn.
Registration Number 0460615

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Martin Mulazzani, M.Sc.

Vienna, 17.03.2014

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Andreas Juch, Bakk. techn.
Kalvarienberggasse 6/23, 1170 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

Many people supported me during my studies, countless talks about topics in my field and also more general discussions made me the person I am today. I'd like to thank them here for their influence and the exchange of ideas.

Especially I'd like to thank my partner Kerstin for her support and her patience during the writing of this thesis. She helped me through countless weekends of writing and programming and always encouraged me to focus the goal of finishing the thesis besides my day job.

Also I'd like to thank my advisors for providing feedback to my thesis and providing a constant stream of interesting new ideas to consider.

I'd also like to thank my parents for enabling me to study both in forms of financial support and also by educating me to the person I am now.

Abstract

The Btrfs (B-tree file system) is a steadily evolving new filesystem for Linux with advanced features not covered by existing filesystems in Linux. It brings new features such as snapshots, subvolumes, it's own volume management and uses checksums extensively. The context of this thesis lies in the field of Digital Forensics and is aimed at the development of novel forensic methods to extract data from forensic filesystems collected during an investigation.

This thesis therefore defines six distinct artifact types that shall be retrieved from such filesystem images. These artifacts are believed to be among the most important data an investingator typically wants to extract during an investigation. These artifacts cover – among others – the metadata of files and the contents of deleted files.

The main question to answer is to what extent the individual artifacts can be recovered from the filesystem data and why this is the case. This question is answered after performing an extensive literature research as well as reverse-engineering the Btrfs disk format, which has not yet been covered in detail by the scientific community to an extent that is required by this thesis. After this process an analysis of the data structures was conducted with the goal of describing them well enough to specify the actual forensic methods suitable to extract the desired artifacts. These forensic methods are later implemented in the de-facto Open Source standard forensics toolkit "The Sleuthkit". This makes it possible to evaluate the methods using test filesystem images. Besides serving the purpose of evaluating the forensic methods, the implementation enables forensic investigators to perform forensic analysis of Btrfs filesystems, which was not possible before due to the lack of tool support.

The results show that five out of the total six artifacts can be extracted by using existing Btrfs filesystem data, thus getting a complete result based on the data. The last artifact, namely the contents of deleted files, are extracted based on heuristics due to incomplete data. The evaluation and related literature shows that in practice this also yields good results.

Kurzfassung

Btrfs (B-tree file system) ist ein ständig weiterentwickeltes, neues Dateisystem für Linux mit fortschrittlichen Funktionen, die derzeit von keinem existierenden Linux-Dateisystem abgedeckt werden. Es unterstützt Snapshots, Subvolumes, hat sein eigenes Volume-Management und verwendet weitgehend Prüfsummen. Der Kontext dieser Arbeit liegt in der Digitalen Forensik und das Ziel der Arbeit ist die Entwicklung neuartiger forensischer Methoden um Daten aus Dateisystemen zu extrahieren, die im Rahmen forensischer Ermittlungen sichergestellt wurden.

Diese Arbeit definiert sechs verschiedene Artefakte die aus den Dateisystemdaten extrahiert werden sollen. Diese Artefakte sind unter den wichtigsten Daten, an denen ein forensicher Ermittler während einer Analyse interessiert ist. Diese Artefakte beinhalten unter anderem die Metadaten von Dateien und den Inhalt gelöschter Dateien.

Die Hauptfrage dieser Arbeit ist inwiefern diese Artefakte aus den Dateisystemdaten extrahiert werden können und warum. Um diese Frage zu beantworden, wird zunächst eine eingehende Literaturrecherche und reverse-engineering des Btrfs Dateisystemformats – welches nicht im für diese Arbeit benötigten Ausmaß wissenschaftlich dokumentiert ist – durchgeführt. Anschließend werden die gefundenen Datenstrukturen analysiert um forensische Methoden spezifizieren zu können, die die oben genannten Artefakte extrahieren können. Diese forensischen Methoden werden dann im de-facto wichtigsten forensischen Open Source Toolkit „The Sleuthkit“ implementiert. Dies ermöglicht sowohl die Evaluierung der forensischen Methoden als auch die forensische Analyse von Btrfs Dateisystemen im Rahmen von Ermittlungen, was vorher auf Grund der fehlenden Werkzeuge nicht möglich war.

Die Ergebnisse zeigen, dass von den sechs Artefakten fünf auf Grund von explizit vorhandenen Daten gefunden werden können, was zu einem vollständigen Ergebnis führt. Das letzte Artefakt, die Inhalte von gelöschten Dateien, wird auf Basis von Heuristiken extrahiert, da die Daten unvollständig sind. Die Ergebnisse zeigen aber, dass das in der Praxis trotzdem zu guten Resultaten führt.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

The dynamic and steadily evolving field of computing has the invariant property that new software creates a demand for better hardware and better hardware leads to the usage of more hardware resources. Both hardware and software influence each others. This is also true for the storage specific disciplines of computing: Larger storage media creates a demand for adoption of the operating systems. New hardware such as solid state drives render previous design criteria of filesystems obsolete since locality in storage is becoming less important due to the lack of rotating disks.

Traditional Unix filesystems [15] are approaching a point at which relevant innovation cannot happen due to their design and partially obsolete concepts (their conceptual origin is the Berkeley Fast File System [28]). Adoption to the new requirements needs new concepts which are not happening due to the need of backwards compatibility in filesystems. This is also true for Linux filesystems. The currently de-facto default filesystem in Linux distributions is Ext (version 3 or 4). It has evolved from the original ext-filesystem which was merged into the Linux kernel in 1993 [37]. Since then it has evolved in the boundaries of being backwards-compatible to the earlier revisions of the filesystem. It has reached a high level of stability and since Ext3 it is has journaling support to tolerate power-failures in a graceful way. Although there has ever been innovation in the new versions of the filesystem, the huge steps towards a modern filesystem couldn't happen without breaking compatibility. So the filesystem remains a very stable and performance optimized solution for most of today's needs, but the future of linux storage is lying somewhere else. [34]

The ZFS filesystem released with Solaris 10 in 2006 was a huge step in the right direction. It solved storage limitations by using 128 bit addresses, included a decent volume manangement to administer space efficiently and provided better abstraction from the underlying storage hardware. It also provided an easy way to create snapshots, introduced checksums for metadata and file contents and allowed many separate filesystems in a pool. [7] In short Sun Microsystems

did lots of things right. Due to the license of the filesystem code, Linux couldn't profit from it, though.

In 2007 Chris Mason from Oracle jumped to the rescue and started the Btrfs project [34]. It reuses most key concepts of ZFS but is written especially for Linux. Since then Btrfs is maturing (Lu et al. compared the evolution of Linux filesystems during eight years and found that even the bug count of mature filesystems doesn't converge [27]) and probably will attract more users since the current Linux filesystems don't offer attractive features needed for larger storage sizes and seem to be one generation behind ZFS.

Digital Forensics are needed whenever the forensic analysis of a filesystem needs to be performed. The encounters of a specific filesystem type increase with the increasing general usage of a filesystem. Thus it can be expected that the demand for forensic methods will increase with the use of Btrfs filesystem. Forensic investigation of a Btrfs volume is currently not possible (feasible) due to the missing tool support. There are currently no scientific publications which describe forensic methods for Btrfs and there are no automated tools to analyzing Btrfs filesystems in a forensic manner.

1.2 Aim of the Thesis

The goal for this master's thesis will be the initial development of forensic methods for the Btrfs file system in order to be able to perform file system analysis. Brian Carrier defines file system analysis as:

> File system analysis examines data in a volume (i.e., a partition or disk) and interprets them as a file system. There are many end results from this process, but examples include listing the files in a directory, recovering deleted content, and viewing the contents of a sector. [10, Chapter 8]

This includes a forensic description of the relevant structures used to store data in Btrfs. Based on this forensic description, methods can be developed to extract forensically relevant data from these structures. To limit the scope of the thesis, no specialized methods to perform tree forensics will be developed. Tree forensics is the application of forensic methods to restore older states of a tree structure. Tree forensics could be an interesting approach, but would widen the scope of the thesis too much.

Additionally to the forensic methods, a prototypical implementation of these methods should be created to make the forensic analysis of Btrfs filesystems possible. This serves two purposes: with just the specified methods it is not feasible to perform analysis in larger scale since too many repetitive manual steps would be necessary, second an implementation makes it possible to evaluate the forensic methods on sample test data.

Prior to defining methods to extract data from Btrfs filesystem images, a definition of what to extract is necessary: The relevant artifacts to collect. Artifacts in the sense of digital forensics is data of interest in forensic investigations. The following enumeration of artifacts is considered in this thesis.

A1 Filesystem Metadata The metadata of the filesystem which includes the layout, parameters and the locations of important structures that the filesystem keeps.

A2 Filesystem Structure The structure of a filesystem includes the filesystem hierarchy, the type of the artifacts (directories, files, etc.).

A3 File Metadata The metadata of files such as size, modification timestamps and file names.

A4 Allocation Status The allocation status of data units in the filesystem including their usage type. This includes the information whether a data unit is used by the filesystem or if it is unused as well as the information what is stored in the allocation unit if it is allocated.

A5 Contents of Allocated Files The files stored in a filesystem are generally the most relevant artifact for forensic investigations. They can be in two states: allocated and unallocated. Allocated files might seem out of scope in the first place because they are covered by the filesystem implementation anyway, but they are relevant because forensic investigations must cover them too and forensic tools should use documented procedures to read the files in a correct way, while filesystem code usually optimize on performance and their algorithms are often not documented.

A6 Contents of Unallocated Files Files which are not allocated by the filesystem are even more interesting. They were deleted and the filesystem implementation doesn't show them any more. Usually their contents remain intact on disk until they are overwritten with new contents. It is not necessarily the case that unallocated data can be restored fully, although experience from other filesystems shows that it is possible that unallocated data can be recovered to some extent.

1.3 Methodological Approach

The development of the forensic methods will include literature research as far as possible since the proposal of the thesis already showed that forensics for Btrfs has not had any significant scientific perception up to now. The lack of scientific works will have to be compensated by original research and the studying of software documentation and source code.

After documenting the Btrfs filesystem structures and theoretically being able to read the contents of the filesystem, the definition of forensic methods can take place. This means that the procedure of aggregating the interesting artifacts is described in textual form.

With the methods defined, the implementation of the forensic methods will be started. The implementation shall be able to retrieve the stated artifacts.

1.4 Structure of the Thesis

This work consists of five chapters. The first chapter being this intoduction.

The second chapter is about the foundations of Btrfs. It starts by a general introduction of the concepts of Btrfs and continues with an in-depth analysis of the On-Disk format. This analysis is the basis for the implementation of the prototype later on. The Analysis describes how data is stored on the physical disk in a Btrfs filesystem. This part is naturally quite technical and very specifying, but nevertheless it is the essential foundation of any sequent task. The section also states the differences between Btrfs and other popular filesystems. This is important to understand the novelty of some features along with the implications that result for the later implementation. The last part of this chapter is a short introduction into Digital Forensics with the relevant steps that lead to a forensic filesystem image that is the base for the later analysis.

The third chapter presents The Sleuthkit as the chosen platform for implementation along with the justification of this choice. There were quite a number of positive and negative aspects about it but the positive side had a bigger impact and most of the negative aspects were neglectible or circumventable in some kind. A description of the relevant API of TSK introduces the area of code which needs to be tackled when implementing support for a new filesystem. Because the API documentation lacks some points, it is described here from a practical point of view. Then the chapter describes the forensic methods which are an application of the analysis part of the previous chapter. Those methods represent the conceptional proceeding to gain artifacts from a Btrfs filesystem. The second part of this chapter covers the implementation details of the defined forensic methods in the scope of the Sleuthkit API. It describes key points from the code along with design decisions that leaded to the specific implemenation. This section includes a lot of references to code samples in the appendix to make the implementation clear to the technically educated reader. This section requires a certain degree of being able to read C source code to understand, but it is not required to fully understand this section and the referenced source code to get a good idea of the conceptional ideas behind this thesis.

The fourth chapter evaluates the suitability of the implementation to retrieve the artifacts defined in the introduction. Thus it is structured after the artifacts to retrieve. The chapter finishes with possible future work in the context of the topic.

The fifth chapter concludes the thesis and recaptures the main achievements made in the course of the thesis.

The two appendices contain the technical details that disturb the flow of reading in the chapters. The first appendix contains the data structures of the Btrfs on-disk format and the second appendix contains the source code examples referenced by other chapters.

CHAPTER 2

Background

This chapter starts with the background knowledge needed to develop forensic methods for the Btrfs filesystem. The first section covers the core features and the general architecture of the Btrfs filesystem, also it compares the new Btrfs features to the state before Btrfs existed. The next section analyzes the Btrfs filesystem in a technical way as the book "Filesystem Forensic Analysis" by Brian Carrier does. The goal is to relate the various filesystem structures to logical groups in order to understand them better. Those descriptions often contain references to the appendix to make it clear how the information is physically stored in the filesystem. The last section covers Digital Forensics, the scientific field in which the thesis is embedded. It starts by an introduction into the field and presents the steps that are needed to obtain a forensic filesystem image which is the base for the application of the forensic methods developed later.

2.1 Btrfs Concepts and Features

Trees

Btrfs is build around trees, it exclusively uses trees for storing anything but actual file contents (with the exception of inline extents). The tree algorithm used in Btrfs is copy on write (COW) friendly b-trees [33]. The whole Btrfs filesystem can be seen as a huge forest of trees (see Figure 2.1). The trees are used as a form of generic data storage where every stored item has an identifier consisting of three elements: object ID, type and offset. This triple consitutes the key used to identify elements inside every tree [3]. Depending on which tree a key is encountered, the key's elements can have a different meaning. This is especially true for the offset. For reading trees, the special meaning of the key's elements is not required.

The trees can have multiple nodes that each have a level associated with them. The leaf level is level 0. Each node of the tree starts with a header (see Table A.5). The header provides an information about the current node (number of entries, level, etc.). The entries of a node are either block pointers or items. In the leaf nodes actual items (see Table A.7) exist, the inner nodes contain block pointers (see Table A.6). Block pointers point to other nodes. Every node

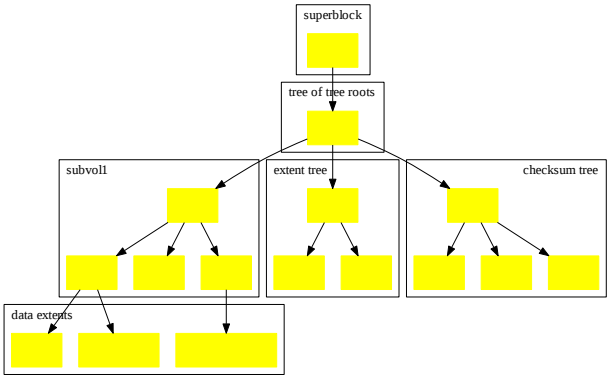

Figure 2.1: A forest of trees. [34]

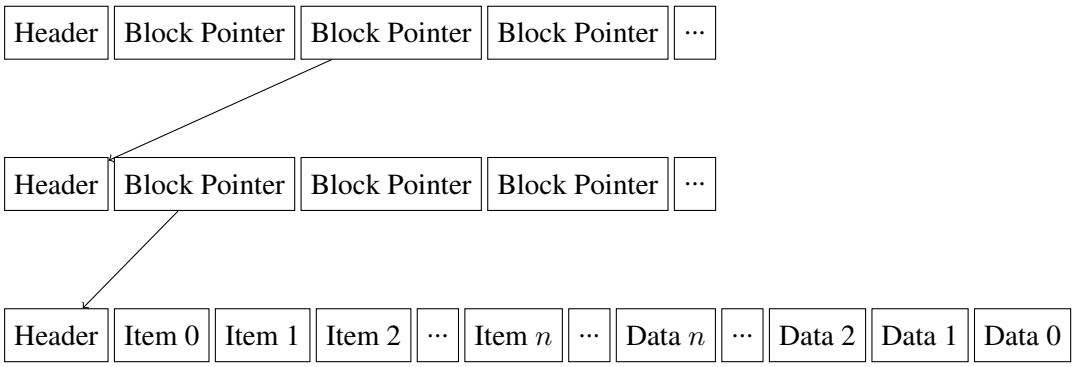

Figure 2.2: A path in the tree structure.

content (item or block pointer) has a key (see Table A.4) associated with it which constitutes the ordering inside the tree. Figure 2.2 provides an example for a path from root to leaf inside such a tree.

COW-friendly trees have more interesting features such as shadowing, a technique used to update trees and the ability to clone trees.

Shadowing The tree is basically just a plain b-tree, the difference lies mostly in the way updates are made. Btrfs uses shadowing when updates to the tree are made. While shadowing is Rodeh's name for this update method, copy-on-write (COW) is mostly used in context of Btrfs. COW ensures that nodes are not updated in-place and thus simply avoiding the need for any locking during reads and ensuring data integrity at unexpected system halts. Every write operation causes the tree to be updated up to the root node (see fig. 2.3). This concept ensures that a reference to the top node is everything needed to address a complete filesystem tree at a specific point of time. A shadowing operation always creates a new version of the whole tree. The path

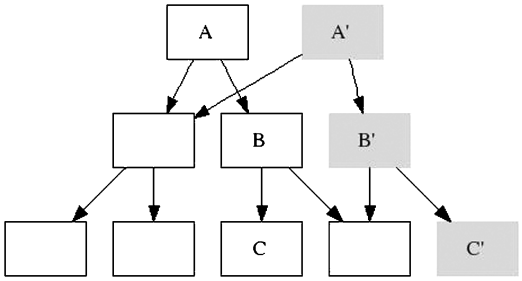

Figure 2.3: The update path for changing node C in a b-tree using shadowing. [33]

Figure 2.4: Cloning of a Tree: before and after. [34]

from the original root node references the tree with the old version of the changed node while the new root node references a tree with the new version of the changed node. With shadowing, new trees are created with every write operation but are written to the disk in form of checkpoints every 30 seconds by default. [33] [34]

Cloning and Reference Counting While shadowing creates new tree versions on checkpoint, it is desireable for a filesystem to clone the complete filesystem tree. Such a copy is easy to take since only a new root node needs to be inserted into the tree to have a clone of the tree, see Figure 2.4. Since not being referenced by the currently traversed tree is not a sufficient condition for considering a node unused because it could be referenced by another tree, reference counting is used to perform this task. Every node has a reference count variable which indicates by how many other nodes it is referenced. Taking a clone typically increases this value by one for every node since every node is part of another tree after cloning. Lazy reference counting avoids having to change every node tough and only increases the reference count of the child nodes of the changed node. [33]

Addressing

The Btrfs filesystem uses logical and physical addresses, both in essence being 64 bit values thus Btrfs is able to address 8 exabytes of storage. [3] Logical addresses are used everywhere in the filesystem as abstraction layer, serving as virtual address space. A logical address identifies a position in the filesystem. Every logical address can be translated into at least one physical address which consists of an offset and a device id needed to find the referenced data on a hard-

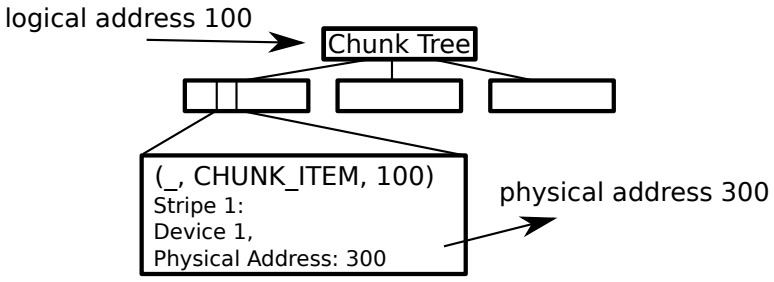

Figure 2.5: Translation of a logical address.

disk. Since a Btrfs volume can consist of multiple disks, a logical address can link to one or more disks. This comes handy when RAID levels are used, since in a RAID-1 volume a logical address can be translated into two physical addresses, while on a RAID-0 volume a logical address typically would only reference one disk.

Logical to Physical Address The translation from logical to physical addresses is done using the chunk tree (see page 12) and for bootstrapping purposes, some parts of the chunk tree are mirrored in the superblock (see Table A.3). The logical address (la) is being translated into a physical address (pa) by searching for matching ranges in the chunk tree (or bootstrap data). Each ChunkItem is referenced by a Key whose offset field (o_k) is the first logical address of the chunk. The ChunkItem specifies the size of the chunk and the physical offset (o_p). If la is between the starting address and the starting address plus size (s), the logical address is found. The ChunkItemStripe data of this ChunkItem is then used to calculate the offset for the specified device: $pa = o_p + (la - o_k)$. Figure 2.5 shows a high level view of the process. For the reverse process of translating physical addresses into logical addresses, the Dev Tree is used (see page 13).

Physical to Logical Address The physical addres (pa) is translated into a logical address (la) by searching the smallest key geq (dev_id, 204, pa) in the Dev Tree. The found DevExtent item (dev_id, 204, o) is then then used to calculate the logical address. It contains the logical address la_d. The result is: $la = la_d + (o - pa)$.

Volume Management

> A volume is a collection of addressable sectors that an Operating System (OS) or application can use for data storage. The sectors in a volume need not be consecutive on a physical storage device; instead, they need to only give the impression that they are. A hard disk is an example of a volume that is located in consecutive sectors. A volume may also be the result of assembling and merging smaller volumes. [10, Chapter 4]

In our context Volume management is the technique of managing the space of one or multiple physical storage devices in form of a virtual storage device. This can involve redundancy algorithms, plain remapping of the address space and combinations of both.

Current State The Linux kernel offers two techniques for abstracting storage: mdadm (RAID) and LVM. RAID allows the combination of individual storage hardware to logical arrays which usually include some sort of redundance (except RAID-0). This can be used to avoid data loss by adding redundancy to a storage system. The Linux kernel's RAID subsystem offers optimized implementations of various RAID-levels.

LVM operates on storage media and does a similar thing. It combines partitions to volume groups which can then be used to create logical volumes (comparable to partitions). LVM logical volumes can easily be resized, snapshots of volumes can be taken and even RAID functionality has been implemented in LVM.

These two approaces are in practice often combined: A system with two disks can have a RAID-1 mirroring configured and a LVM volume group on top of it to allow flexible allocation of space to various areas of the filesystem. These two storage abstraction techniques can be combined in many ways and are independent of the filesystem used.

The Btrfs Approach Btrfs could be used on top of a RAID or LVM volume like any other filesystem, but the Btrfs authors chose to reimplement volume management in Btrfs. This has a number of advantages due to limitations of the layering approach of RAID and LVM.

A RAID array with two-disk mirroring can easily prevent the loss of data when one disk fails: the other copy is used and the array is marked as degraded. But when one copy has corrupt data it is not defined which copy is used then. Due to the absence of checksums it cannot be determined which copy is valid and which one is corrupt. In the worst-case the valid data is replaced with the corrupt data. This is 50% chance. If Btrfs is used on top of a RAID-1 volume, it has no control over that recovery mechanism. Even Btrfs checksums would only allow the detection of corrupt data, the repairing is done by a lower level which cannot access that information. So the Btrfs authors chose to reimplement volume management to gain control over the disks and to ensure data integrity using checksums. In the stated RAID-1 example the process of a two-disk Btrfs volume would be different: The Btrfs code would notice the mismatching checksum of the file and replace it with the valid data with the correct checksum. This way data loss is prevented and integrity is ensured by using checksums. [3]

Recovering from data corruption/disk failure in RAID is a time consuming process as the whole array needs to be resynced. When a filesystem on top of a RAID volume is only 1% full, the remaining 99% of the disk need to be synced too since the RAID implementation is filesystem agnostic and does not know which areas of the disk are used and which areas are free space. Btrfs would only write used areas to the corrupt/new disk because it knows which parts of the filesystem are used.

Subvolumes

Subvolumes are a technique to create multiple filesystems in a single Btrfs volume.

Current State Current Linux filesystems keep one filesystem per partition or logical volume. That filesystem has a mount point in the filesystem tree and all of the filesystem's files are (apart from access control mechanisms) visible to the user. The traditional answer for isolation of different areas of storage in a system is partitioning with the disadvantage of having to know the partition's sizes at install time or using LVM to gain more flexibility to resize the logical volumes later. However both approaches lack a certain flexibility and require lots of manual steps to be carried out before a new partion or logical volume can be used. The worst-case is dividing a mounted partition into two partition involves umounting, fsck, resizing partition, creating a new partition, creating a new filesystem, mounting both. This is not only work intensive and error prone but also requires umounting the filesystem which could mean service interruption.

The Btrfs Approach Subvolumes are different in that matter. A subvolume is a filesystem in the traditional sense, but multiple of those subvolumes can be inside a volume. A subvolume is technically just a separate filesystem tree. [3] Since the filesystem tree stores the filesystem-specific data of the filesystem, each filesystem tree can be changed separately and only the common trees (Chunk Tree, etc.) need to be updated. A subvolume is a separate logical partition for filesystems. Each filesystem can hold completely independent files that are not related to other subvolumes. Since creation of subvolumes is a very quick and unproblematic step, it can help to avoid other partitioning methods. The process of creating a subvolume just involves creating a new filesystem tree and this is very fast.

Snapshots

Snapshots are a space-efficient and fast method to make copies of subvolumes.

Current State Taking snapshots of data is not supported in current Linux filesystems, but can be done using tools like rsnapshot, which creates snapshots of files using hard links. This is not very fast and has disadvantages due to the use of hard links. If a file in a snapshot is changed, it is changed in every snapshot due to being hard-linked. LVM implements filesystem-agnostic snapshots as well. Their method of snapshotting is to save the data changed in the original logical volume to the snapshot logical volume, thus the snapshot grows in its size when the original filesystem is modified.

The Btrfs Approach Btrfs snapshots are very simple as taking a snapshot requires just a new copy of the filesystem tree which is ref-counted which is the requirement for taking snapshots [34]. The snapshotted tree will end up with a new root node.

Checksums

Current State Current Linux Filesystems don't create checksums for data stored on a disk. Ext4 uses checksums for it's journal, but data doesn't profit from that.

	File System	Content	Metadata	File Name	Application
ExtX	Superblock, group descriptor	Blocks, block bitmap	Inodes, inode bitmap, extended attributes	Directory entries	Journal
NTFS	$Boot, $Volume, $AttrDef	Clusters, $Bitmap	$MFT, $MFT-Mirr, $STANDARD_INFORMATION, $DATA, $ATTRIBUTE_LIST, $SECURITY_DESCRIPTOR	$FILE-_NAME, $IDX_ROOT, $IDX_ALLOCATION, $BITMAP	Disk Quota, Journal, Change Journal
Btrfs	Superblock, Chunk Tree, Dev Tree, Tree of Tree Roots	Extents, Inline Extents	Extent Tree, Checksum Tree	FS-Tree	

Figure 2.6: The data categories of ExtX, NTFS and Btrfs. As in [10, Chapter 8]

The Btrfs Approach Btrfs checksums all of its data [3]. At mount-time when the superblock is read, it's checksum is verified. Every tree header contains the checksum of the current node, which means that every tree along with its contents is checksummed. For data a checksum tree to store checksums of data extents. Checksums are essential to detect corrupt data and metadata.

2.2 Btrfs Analysis

This section provides a deep introduction on how Btrfs works in terms of data storage. This is relevant for forensic investigations relating to Btrfs filesystems. This section also will compare the Btrfs filesystem to other filesystems: NTFS, ExtX and ZFS. This part is loosely structured as the filesystem specific parts in the book Filesystem Forensic Analysis by Brian Carrier [10, Chapter 12]. In table 2.6 there is a comparison of the data types of those three filesystems which allows to relate Btrfs data structures to the relevant data layers to gain a faster understanding about Btrfs.

Filesystem Category

The filesystem category is defined by Carrier as a "category [that] contains the general data that identify how this file system is unique and where other important data are located." "Analysis of data in the file system category is required for all types of file system analysis because it is during this phase that you will find the location of the data structures in the other categories." [10, Chapter 8]

In the case of Btrfs this includes the Superblock, Chunk Tree, Dev Tree and the Tree of Tree Roots. The justification for the inclusion of the numerous trees in this category is that address translation is necessary for operating on other categories of the filesystem data and thus those trees are essential for any other operation inside the filesystem.

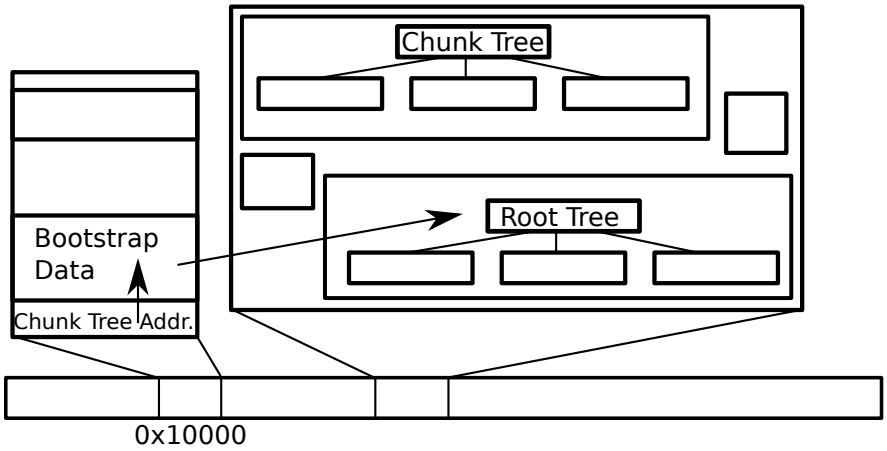

Figure 2.7: Function of the superblock during mount.

Superblock The general starting point for reading and analyzing filesystems is the superblock. It is usually the first area which is read while mounting the filesystem. It contains information about the filesystem layout, its size, used features and generally provides information about where to go from here. Therefore filesystems usually place the superblock in a fixed location to be able to find this essential information easily.

In Btrfs the superblock is located at 0x10000 and is the only data structure with a fixed address. It contains important information like the addresses of the Root of Roots Tree (see Figure 2.7), the address of the Chunk Tree as well as the bootstrap data to resolve the addresses of those essential trees (see Table A.3). This information is sufficient to find all the trees stored in a Btrfs volume. Apart from that, the superblock stores mostly parameters and general information about the Btrfs volume such as bytes used, sector size, and many more. It contains a checksum which covers the rest of the superblock (excluding the checksum itself). The Superblock is copied several times for data safety.

In comparison, the ExtX superblock is 1024 bytes long and located 1024 bytes after the start of the filesystem. It stores the information about the filesystem layout, mostly in terms of size. The ExtX layout is much more fixed than Btrfs' layout, so the superblock just contains some stat information (number of inodes, number of block groups, number of inodes per block group, etc.). [10, Chapter 14]

NTFS keeps this essential information in the boot sector which is located in the first 16 sectors of the filesystem and is referenced as file $Boot. Otherwise it is similar: It contains the addresses of important data structures as the Master File Table (MFT). [10, Chapter 12]

Chunk Tree Btrfs allocates space into chunks. A chunk is a continuous range of space which is used for a designated type of storage data (data, metadata or system data). Chunks usually have fixed sizes (1GiB for data chunks, 256MiB for metadata chunks [3]). The Chunk Tree holds DevItem (see Table A.9) and ChunkItem (see Table A.10, A.11) elements. It is used as

```
item 0 key (DEV_ITEMS DEV_ITEM 1)
    itemoff 3897 itemsize 98 dev item devid 1
    total_bytes 2048000000 bytes used 1266483200
item 1 key (FIRST_CHUNK_TREE CHUNK_ITEM 0)
    itemoff 3817 itemsize 80
    chunk length 4194304 owner 2 type 2 num_stripes 1
            stripe 0 devid 1 offset 0
item 2 key (FIRST_CHUNK_TREE CHUNK_ITEM 4194304)
    itemoff 3737 itemsize 80
    chunk length 8388608 owner 2 type 4 num_stripes 1
            stripe 0 devid 1 offset 4194304
item 3 key (FIRST_CHUNK_TREE CHUNK_ITEM 12582912)
    itemoff 3657 itemsize 80
    chunk length 8388608 owner 2 type 1 num_stripes 1
            stripe 0 devid 1 offset 12582912
```

Figure 2.8: Example of a chunk tree created with btrfs-debug-tree

an indirection layer for addressing storage which can be located on various physical disks and even be mirrored in different ways. The contents of the tree are also used for translating logical addresses into physical addresses (see 2.1). The separation of data and metadata allows the volume management to have different profiles for duplication defined for each type of data. So a user could use mirroring for metadata and striping for data. [3]

An example of chunk tree contents are shown in figure 2.8. The chunk tree in the example contains a single Dev Item describing a 2GiB device and several chunk items which translate the logical addresses in the key's offset field to the physical addresses in the chunk item stripes.

NTFS doesn't use indirection for addressing. It references files outside the MFT by their cluster number, which is a much simpler concept. Since the cluster size is fixed per filesystem, the physical address is just a multiplication of cluster size and cluster number. [10, Chapter 12]

ExtX uses block numbers inside the inodes to reference files contents. Each inode can store "the addresses of the first 12 blocks that a file has allocated. These are called direct pointers. If a file needs more than 12 blocks, a block is allocated to store the remaining addresses." [10, Chapter 14]. This is again a much simpler concept. The addressing is always done using a block number, possibly through a block pointer if a file allocates more space.

ZFS uses a quite similar concept: It addresses files with a DVA (Data Virtual Address) which consists of the virtual device id (32bit) and a sector offset (63[sic] bit). This concept is comparable to the concept of Btrfs: the vdev could be seen as chunk and the offset is the subtraction of the logical address in Btrfs minus the chunk's logical address. [5]

Dev Tree The Dev Tree is used to reverse the process of address translation. The Dev Tree holds DevExtent (see Table A.13) structures. DevExtent items are used to translate physical addresses into logical addresses. Their key in the Dev Tree is (device ID, 204, physical address).

```
item 0 key (0 UNKNOWN 1) itemoff 3955 itemsize 40
item 1 key (1 DEV_EXTENT 0) itemoff 3907 itemsize 48
    dev extent chunk_tree 3
    chunk objectid 256 chunk offset 0 length 4194304
item 2 key (1 DEV_EXTENT 4194304) itemoff 3859 itemsize 48
    dev extent chunk_tree 3
    chunk objectid 256 chunk offset 4194304 length 8388608
```

Figure 2.9: Example of a dev tree created with btrfs-debug-tree

```
item 0 key (EXTENT_TREE ROOT_ITEM 0) itemoff 3556 itemsize 439
    root data bytenr 132579328 level 1 dirid 0 refs 1 gen 23
item 1 key (DEV_TREE ROOT_ITEM 0) itemoff 3117 itemsize 439
    root data bytenr 132530176 level 0 dirid 0 refs 1 gen 21
item 2 key (FS_TREE INODE_REF 6) itemoff 3100 itemsize 17
    inode ref index 0 namelen 7 name: default
item 3 key (FS_TREE ROOT_ITEM 0) itemoff 2661 itemsize 439
    root data bytenr 132517888 level 0 dirid 256 refs 1 gen 21
```

Figure 2.10: Example of a root of roots tree created with btrfs-debug-tree

With that, it's possible to translate a known physical address into a logical address by searching for a key which has an offset greater than or equal to the physical address. Since the items in the tree are sorted ascending, this is very efficient. Transforming logical addresses into physical addresses is done through the chunk tree.

An example of a dev tree follows in figure 2.9. It shows a part of the contents of a dev tree with several Dev Extents which translate the logical addresses in the key's offset fields to physical addresses inside the Dev Extents.

As mentioned before, NTFS and ExtX don't share this concept of address translation, so there is also no concept to reverse this process.

Tree of Tree Roots The Tree of Tree Roots stores information on how the other trees can be found in Btrfs. Its logical address is stored in the superblock and can be translated to a physical address with help of the bootstrapping data. The tree of tree roots stores RootItem (see Table A.8) entries which describe how the Root node of every other tree in the filesystem can be reached (see Figure 2.1). [34] The logical address of the root node (block number of the root node) is used to locate the root node of the individual tree. The tree also contains RootRef and RootBackref items which are not of such great interest. A pointer to the default filesystem is also stored in the tree.

An example of a Tree of Tree Roots is shown in figure 2.10. It shows the Root Items of the Extent Tree, the Dev Tree and the Fs Tree along with the Inode Ref of the default Fs Tree.

```
item 20 key (260 EXTENT_DATA 0) itemoff 2678 itemsize 53
    extent data disk byte 336527360 nr 150798336
    extent data offset 0 nr 150798336 ram 150798336
    extent compression 0
item 21 key (260 EXTENT_DATA 150798336) itemoff 2625
    itemsize 53
    extent data disk byte 541327360 nr 150798336
    extent data offset 0 nr 150798336 ram 150798336
    extent compression 0
item 22 key (260 EXTENT_DATA 301596672) itemoff 2572
    itemsize 53
    extent data disk byte 746127360 nr 197439488
    extent data offset 0 nr 197439488 ram 197439488
    extent compression 0
```

Figure 2.11: Example of extents inside the FS tree created with btrfs-debug-tree

Neither NTFS nor ExtX share this concept, which is not surprising since they use more or less fixed tables for addressing. ZFS interestingly works completely different too: It relies mostly on dnodes which can have different types (just like the tree items in Btrfs), which are grouped in Meta Object Sets (MOS) and reference each others. [5]

Content Category

Brian Carrier defines this category as follows:

"The content category includes the storage locations that are allocated to files and directories so that they can save data. The data in this category are typically organized into equal sized groups[...]" [10, Chapter 8]

This is not totally applicable to Btrfs since there is no such equal sized group (blocks, clusters) which Btrfs uses to allocate files. Btrfs stores files in extents are continuous areas of storage without any additional format. In Btrfs there are two types of extents: (normal) extents and inline extents.

Normal Extents are used for larger files. The contents of the file are split up in extents. The extents are referenced from the FS tree (see fig. 2.11). The extent data items in there show three extent parts of the same file, which can bee seen from their identical object ID, the offset field of the key is equal to the offset inside the file.

Inline Extents are used for small files. The contents of the file are put directly inside the FS tree inside the extent data items.

The distinction between inline extents and normal extents is also used in similar form in NTFS: attributes can either be resident or non-resident. Resident attributes are stored directly in the MFT entry, while non-resident attributes are referenced by their clusters. [10, Chapter 12]

Ext2,3 on the other side always stores data in blocks and references them from inodes. Blocks very similar to NTFS clusters and are fixed in size at filesystem creation time. [10, Chapter 14] Ext4 introduced extents which are stored in the inodes. They also work as in Btrfs: they define the start of the data and the offset inside the file. [16]

Metadata Category

> The metadata category is where the descriptive data reside. Here we can find, for example, the last accessed time and the addresses of the data units that a file has allocated. Few tools explicitly identify metadata analysis; instead, it is typically merged with file name category analysis. We separate them, though, in this book to show where the data are coming from and why some deleted files cannot be recovered. [10, Chapter 8]

In Terms of Btrfs this data is stored in inode entries inside the FS tree. So the filesystem tree is introduced in this category but will reappear in the file name category again.

FS Tree The filesystem tree represents a subvolume in the Btrfs volume. This means there can be multiple FS trees inside a single volume. The FS Tree stores multiple items: InodeItem, InodeRef, DirItem, DirIndex, XattrItem and ExtentData. These structures are sufficient to represent the file system structure in the filesystem tree.

InodeItem (see Table A.14) structs are used to store the inode specific stat data of a file or directory. The stat struct in the kernel can be seen in figure 2.12. The data can be obtained by userspace programs with the stat command. The Btrfs InodeItem shares most of the data (except st_blksize, st_ino and st_dev) in the InodeItem. The InodeItem is very similar to the ExtX inode in terms of the user-visible data items. In the NTFS terminology the InodeItem would be part of the MFT entry.

ExtentData (see Table A.18) structs reference the actual file content. The data can either be stored directly in the tree as inline extent. In this case, the type field is set to 0 and the data follows the mandatory data parts. The data following the ExtentData structure is exactly n bytes long. Inline extents are used for small files. Larger files are usually stored outside the filesystem tree in extents. In that case the structure has the optional fields filled. The file content is found at the logical address specified in the ExtentData struct. Storing the smaller files near their metadata has the advantages of being very space efficient because there is no block-size which leaves unused slack-space behind if the file doesn't match the block-size and it has the advantage of being physically close to the metadata which is an advantage when reading the file. Usually reading the metadata already causes a small file's data to be in the reading cache. [3]

InodeRef (see Table A.15) elements translate inode numbers to file names. Their key is (inode id, 12, directory id). The directory ID is the inode number of the directory containing the inode.

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Figure 2.12: stat structure [2]

DirItem (see Table A.16) structs are used to find files and directories by their name. The key of a DirItem consists of (parent object id, 84, crc32c hash of the file name). Finding a file by name involves hashing the name and looking it up in the FS tree. The resulting DirItem can have repeated entries because of possible hash collisions.

DirIndex (see Table A.17) are used to look up files and directories by their index in a directory. This is usually done during a directory listing.

InodeRef, DirItem and DirIndex would relate to directory entries in the ExtX terminology: "An ExtX directory is just like a regular file except that it has a special type value in its inode. Directories allocate blocks that will contain a list of directory entry data structures. A directory entry is a simple data structure that contains the file name and the inode address where the file's metadata can be found." [10, Chapter 14] The directory entry is actually a linked list which contains references to all inodes and the file names. In Ext4 the usage of HTrees for this purpose became mandatory while Ext3 allowed this. [16]. NTFS uses Indexes to store directory entries in a b-tree. The directory itself is referenced from the MFT. [10, Chapters 11,12]. This is somehow similar to Btrfs where the index can be seen as part of the FS tree.

Checksum Tree The checksum tree stores checksums for extents. Since checksums are metadata, they are mentioned here, but for the forensic analysis they don't provide new information. The only possible thing would be the detection of corrupt extents.

ExtX and Ext4 don't provide checksums for data, but Ext4 provides checksums for journaling and block groups. [16].

Extent Tree The extent tree is the reverse of all FS trees. It constitutes a single tree which does the bookkeeping of allocations for the whole filesystem. The extent tree stores Extent Item objects which constitute the reference counted allocation map of the Btrfs volume. Extents are contiguous on-disk areas that hold user-data without additional headers or formatting. [34] Items

are referenced by the Extent Data items of the filesystem/subvolume tree. They are reference counted to support cloning filesystem trees. The extent items store an array of references to the actual data.

File Name Category

The file name category is already covered by the FS Tree in the Metadata Category (DirIndex and DirItem in the FS Tree) and thus not covered here again.

2.3 Digital Forensics

The term Digital Forensics describes Forensic Science applied to digital data. Often it is also called Computer Forensics [10, Foreword]. Digital Forensics is a wide field that not exclusively covers network forensics [14] [6], mobile forensics [36], memory forensics [35] and forensic data analysis, which is the topic of this thesis.

The NIST defines Computer Forensics, another often used term for Digital Forensics, as:

> The application of science to the identification, collection, examination, and analysis of data while preserving the integrity of the information and maintaining a strict chain of custody for the data. [23]

Brian Carrier defines Digital Forensics as:

> A digital forensic investigation is a process that uses science and technology to analyze digital objects and that develops and tests theories, which can be entered into a court of law, to answer questions about events that occurred. [10, Chapter 1]

Digital Forensics originated in the in the 1970s in the form of data recovery procedures. In the 1980s specialized tools were developed and widely used (e.g. “undelete”). This time was characterized by a huge diversity of hardware and software, widespread use of proprietary and undocumented file formats, centralized computing facilities compared to today’s omnipresence of computing equipment and the lack of formal processes, tools and education. Relatively small storace capacities often didn’t require digital forensics methods as printouts of storage media and manual analysis was possible. The years from 1999 to 2007 may be characterized as a “Golden Age” for digital forensics as the previous diversity in both hardware and software consolidated into a dominance of Microsoft Windows on the software side and a thorough standardization on the hardware side. The amount of relevant file formats of forensic interest (Microsoft Office formats, JPEG, AVI, WMV) decreased and forensic tools became better. [19]

The digital forensics process shares commonalities with the physical forensic process which could consist of the following phases [12]:

Preservation Phase Usually securing the exits of the crime scene, helping the wounded, detaining the suspects and identifying the witnesses.

Survey Phase Walking through the scene and identifying physical evidence, developing a first theory, documenting the fragile evidence.

Documentation Phase Involves taking photos, drawing sketches of the scene; the goal is to capture as much information as possible about the crime scene.

Search and Collection Phase The intensive search and collection of additional physical evidence in the crime scene, often with a concrete missing evidence (e.g. a weapon). After this phase the process usually continues outside the actual crime scene.

Reconstruction Phase The collected evidence is organized and a theory for the happened incident is developed.

Presentation Phase The evidence is presented to a court along with the theory developed during the investigation.

This process is of course not directly appicable for digital evidence as the physical device containing the digital data (e.g. a personal computer) should be treated as a crime scene on its own, so the above stated phases should be adopted to each physical device containing digital data separately.

The NIST defines the forensic phases as follows [23]:

Collection Data related to a specific event is identified, labeled, recorded, and collected, and its integrity is preserved

Examination Forensic tools and techniques appropriate to the types of data that were collected are executed to identify and extract the relevant information from the collected data while protecting its integrity. Examination may use a combination of automated tools and manual processes.

Analysis Involves analyzing the results of the examination to derive useful information that addresses the questions that were the impetus for performing the collection and examination.

Reporting May include describing the actions performed, determining what other actions need to be performed, and recommending improvements to policies, guidelines, procedures, tools, and other aspects of the forensic process.

The Preservation or Collection Phase is especially important in this process since computers store a lot of important data in volatile memory. Thus evidence collection should be done in the order of volatility, which has been defined in RFC3227 [8] as follows:

1. registers, cache
2. routing table, arp cache, process table, kernel statistics, memory
3. temporary file systems
4. disk
5. remote logging and monitoring data that is relevant to the system in question
6. physical configuration, network topology

7. archival media

The focus of this work concentrates on the disk, but the data with higher volatility are also becoming increasingly important. For example full disk encryption can prevent the analysis of data unless the encryption passphrase is known. [13] The acquisition of the passphrase could be part of the physical evidence collection (e.g. passphrase written on paper), acquired by interrogation or collected from the memory of the running system. The acquisition of a memory dump of a running system could be obtained with specialized hardware such as the prototypical "Tribble" PCI device [11]. Another possibility is the access of the system memory over DMA over Firewire [4] if the target system has an active Firewire port. Another possibility is the "warm-reboot" memory acquisition which dumps the memory to an USB drive after a reboot [38]. The "cold-boot" attack is also an interesting method to obtain a memory dump. It uses multipurpose duster spray to cool the memory chips to -50 degrees celsius, which causes the volatile data to persist longer before removing them from the device and reading out their contents on another machine [22].

The Preservation Phase should, apart from the above mentioned preservation of the memory, also cover the separation of the device from networks, the recording of processes, logged in users, and so on. These things are already covered by a memory dump, which has the additional advantage of not modifying the memory during the collection process. RFC3227 [8] suggests the usage of statically linked utilities to collect information about the running system, but since then the techniques for dumping system memory have been improved and should be preferred because they enable the investigator to see the complete state of the target device. Modern memory analysis frameworks such as volatility [39] enable the investigator to extract information directly from the memory dump instead of relying on a small subset of information that was captured from a running system.

The acquisition of a hard drive of a non-running system or a system that was shut down after acquiring a memory dump should start by using a write blocker to prevent accidential modification of data. Nelson et al. [30] suggest using a flag in the Windows registry to turn off writes to USB devices and using a USB interface to connect the hard disk to acquire the image from. A hardware write blocking device is a safer choice especially since the NIST didn't verify Windows[1]. The NIST suggests using a three step process to aquire hard disk images [23]:

- Develop a plan to acquire data
- Acquire data
- Verify integrity of acquired data

As long as it can be assured that the data is not modified and when the data can be verified to be the same as the captured data, and the chain of custody is strictly documented, the forensic process can continue with the Examination phase. The result of the Collection phase is the input to the methods described later in this thesis.

The natural antagonist of Digital Forensics shall also be mentioned here: Anti-Forensics is the summarization of techniques to prevent forensic methods to be applied successfully. This

[1] `http://www.cftt.nist.gov/software_write_block.htm`, Accessed 16.03.2014

includes secure deletion of data in hard drives [21] or SSDs [41], steganography [40] and cryptography [13]. Cryptography and its possible countermeasures on running systems have been discussed before, secure data deletion would certainly leave nothing left to analyze for any forensic scientist, but in practice people exhibit a more careless attitude towards data safety [17].

CHAPTER 3

Design

This chapter starts with a presentation of The Sleuthkit and continues with a justification why it has been chosen as the platform for the implementation of the forensic methods. It also gives an overview about the most important command line tools it provides and finishes with a description of it's filesystem specific API. The chapter continues with a definition of the forensic methods to retrieve artifacts from the Btrfs filesystem data. The methods apply the theoretical knowledge from the analysis part of the previous chapter and provide the necessary dynamic aspects to turn the static filesystem data into forensic artifacts. The methods defined here are structured by the artifacts they retrieve. The third section describes the implementation details of the methods inside The Sleuthkit.

3.1 The Sleuthkit

Finding digital evidence during an investigation is not something that is usually done by manually investigating filesystem images with the help of a hex editor. It usually is done with the help of software. Several commercial and free systems exist for the purpose of analyzing filesystems to find evidence.

Implementing forensic methods for a new filesystem, as this thesis tries to do, could be done as a completely independent tool with its own user interface, own parameters and file formats. However, the taken approach is another one. The implementation will be in form of a extension of a very widespread open-source forensics toolkit called The Sleuth Kit by Brian Carrier, which is the de-facto standard package in the open source world:

> The Sleuth Kit (TSK) and the Autopsy Forensic Browser are open source Unix-based tools that [Brian Carrier] first released (in some form) in early 2001. TSK is a collection of over 20 command line tools that can analyze disk and file system images for evidence. To make the analysis easier, the Autopsy Forensic Browser can be used. Autopsy is a front end to the TSK tools and provides a point-and-click type of interface. [10, Appendix A]

This has some advantages and disadvantages which will be discussed in this section.

Reasons for the Implementation in The Sleuth Kit

Efficiency The Sleuth Kit already provides forensic analysis for a large number of filesystems. The code is separated into general code and filesystem specific code. This is a quite obvious design decision because it is desireable to have a generic codebase instead of reimplementing the same functionality for every filesystem. At the same time this means that an API needs to be specified which the filesystem specific code is implemented against. That is a certain risk since Btrfs offers quite new features which didn't exist in previous Linux filesystems and thus raises the question whether The Sleuth Kit API is suitable for extending it for Btrfs.

Existing User Base A very clear advantage for extending TSK is its existing user base. Every new program requires learning it's syntax, usage and reading documentation to understand the way it works. This can be a cumbersome process, especially in more complex fields. Extending TSK in essence eliminates the learning phase for everybody knowing how to use TSK for other filesystems as the usage for analysing Btrfs is exactly the same.

Related Tools Related tools which use TSK for the forensic analysis of filesystems can analyze Btrfs filesystems as well because their side of the API respectively the command line (depending on how TSK is integrated into another tool) doesn't change. This is a clear advantage for extending TSK instead of the own implementation.

Existing Design This is somehow related to efficiency. Designing a new piece of software is a complex process which constitutes a risk for the later implementation if done wrong. Relying on an existing design clearly reduces the risk of coming to wrong design decisions as there already exists an implementation which is working. But again this point is not a clear win for extending TSK since the API wasn't designed for the advanced features of Btrfs. This places a risk on implementing Btrfs in TSK, especially since a similar approach for ZFS failed, although no concrete details were mentioned. [26]

Proven Code Base Due to being actively used by forensic investigators and having a community based around TSK, a higher level of quality can be supposed than the implementation by a single person. The large and tested common codebase is clearly an advantage for extending TSK. The fact that TSK is open source software origniating from an academic background may be an advantage too. [9]

Freedom of Choice Chosing the programming language, libraries and the environment (build system, scm, etc.) is an advantage mostly in the standalone implementation choice. Extending TSK fixes the choice of most of the previously listed parameters. This includes the programming language, the libraries to a high extent (since the goal of the extendsion is the integration of the Btrfs code into the upstream TSK code the final patch should be compact and not have a huge list of dependencies). Using a modern build system (such as CMake) is also impossible. The

TSK project's choice of Git as SCM is very suitable for working independently of the upstream project on the new features.

Image Format Support Raw filesystem images are not the only type of image that is created during the collection phase of a forensic investigation. TSK supports six other formats besides Raw, including the open AFF format [18]. A standalone tool would have to implement support for these formats itself.

In sum the arguments for extending The Sleuth Kit had more weight than implementing a standalone tool which has the high risk of not being used in practice. Thus the implementation of a standalone application or tool is discarded.

Command Line Tools

TSK consists of many command line tools which follow the Unix principle of serving a single purpose. It's tools are designed to work on different layers of a filesystem [10, Appendix A] of which the most relevant are:

- File System Category: fsstat
- Content Category: dcat, dls, dstat, dcalc[1]
- Metadata Category: icat, ifind, ils, istat
- File Name Category: ffind, fls

The filesystem layer is the highest layer and referes to the filesystem as a whole. Thus only very general information is relevant here. The file name layer is the second highest layer, it works with file names which assign meta data elements their names. That is usually the level that the user of an operating systems means when he refers to the term filesystem. The metadata layer is below the file name layer and contains the basic information about allocations. This includes various timestamps, the sizes of allocations and similar. Also the file contents fall into this category as their locations are defined by metadata. The content category operates on the allocation units.

fsstat Displays some general information about a filesystem. It depends on the filesystem analyzed which information is displayed here [10, Appendix A]. Usually it covers basic parameters of the filesystem such as size of the filesystem, size of allocation (block size) and similar.

fls The fls command (see Figure 3.4) lists the file names in a given directory either in human readable form (see Figure 3.2) or in machine readable form (see Figure 3.3). [10, Appendix A] It is comparable to the ls command in a normal Unix filesystem. It's arguments include various filtering and output format options besides filters for deleted files. In the example in Figure 3.2 a small filesystem is shown which contains a directory and three files. One file is contained in the directory while the others are located in the filesystem root.

[1] These tools are called blkcat, blkls, blkstat and blkcalc in newer TSK versions.

```
$ fsstat ext3.img
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name:
Volume ID: 58c992e6b7aebcafb94554a37ebdc693

Last Written at: Sat Feb  1 16:56:17 2014
Last Checked at: Sat Feb  1 16:56:17 2014

Last Mounted at: emptyUnmounted properly
Last mounted on:

Source OS: Linux
Dynamic Structure
Compat Features: Journal, Ext Attributes, Resize Inode,
    Dir Index
InCompat Features: Filetype,
Read Only Compat Features: Sparse Super, Has Large Files,
[...]
```

Figure 3.1: Example of an fsstat invocation for a Ext3 filesystem image.

```
$ fls -r ~/Diplomarbeit/testdata/fs1
r/r 1: testfile
d/d 2: testdir
+ r/r 3: file2
r/r 4: linux-3.7.tar
```

Figure 3.2: Example listing recursively all files in normal format.

```
./fls -f btrfs -a -r -m / ~/Diplomarbeit/testdata/fs1
0|/testfile|1|r/r---------|0|0|18|1353402818|
    1347964140|0|1347964140
0|/testdir|2|d/d---------|0|0|10|1353404607|
    1353404619|0|1353404619
0|/testdir/file2|3|r/r---------|0|0|13|1353404619|
    1353404619|0|1353404619
0|/linux-3.7.tar|4|r/r---------|0|0|499036160|
    1366612191|1366612199|0|1366612199
```

Figure 3.3: Example listing recursively allocated files in a machine-readable format.

```
usage: fls [-adDFlpruvV] [-f fstype] [-i imgtype]
[-b dev_sector_size] [-m dir/] [-o imgoffset] [-z ZONE]
[-s seconds] image [images] [inode]
If [inode] is not given, the root directory is used
-a: Display "." and ".." entries
-d: Display deleted entries only
-D: Display only directories
-F: Display only files
-l: Display long version (like ls -l)
-i imgtype: Format of image file (use '-i list' for
     supported types)
-b dev_sector_size: The size (in bytes) of the device
     sectors
-f fstype: File system type (use '-f list' for
     supported types)
-m: Display output in mactime input format with
      dir/ as the actual mount point of the image
-o imgoffset: Offset into image file (in sectors)
-p: Display full path for each file
-r: Recurse on directory entries
-u: Display undeleted entries only
-v: verbose output to stderr
-V: Print version
-z: Time zone of original machine (i.e. EST5EDT or GMT)
        (only useful with -l)
-s seconds: Time skew of original machine (in seconds)
       (only useful with -l & -m)
```

Figure 3.4: Command line options of fls.

icat The icat command outputs the contents of a file referenced by a metadata address. The metadata address is the internal address a file is given by a particular filesystem. [10, Appendix A]

ils The ils command lists the known metadata allocations of the filesystem. An example of the output of this command is given in Figure 3.5.

istat The istat command outputs the allocation status of a specific data unit (see Figure 3.6). [10, Appendix A]

blkcat The blkcat command outputs the contents of a data unit (e.g. block). [10, Appendix A] An example of the blkcat output of a part of a Btrfs superblock can be seen in Figure 3.7.

```
0|a|0|0|1366612191|1366612149|1366612191|0|40555|1|56
1|a|0|0|1347964140|1353402818|1347964140|0|100644|1|18
2|a|0|0|1353404619|1353404607|1353404619|0|40755|1|10
3|a|0|0|1353404619|1353404619|1353404619|0|100644|1|13
4|a|0|0|1366612199|1366612191|1366612199|0|100644|1|499036160
```

Figure 3.5: ils output listing the allocated inodes on a filesystem.

```
=== stat info ===
Size:  10
Access time:  1353404607
Modified time:  1353404619
Create time:  1353404619
```

Figure 3.6: istat information for a metadata address.

```
$ blkcat -a ~/Diplomarbeit/testdata/fs1 0x10000 | fold -w 60
.. ..............................G..tM.......i........
........._BHRfS_M..................@.....................
z.....p.....................................................
.....................................z......}K..............
................................F:2..C...,(g..|..G..tM.....
..i.mytestlabel.............................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................@....
.....................................................................F:
2..C...,(g..|...........@.............................".....
.............................@......F:2..C...,(g..|..........
.......F:
```

Figure 3.7: blkcat output of the Btrfs superblock of a test image.

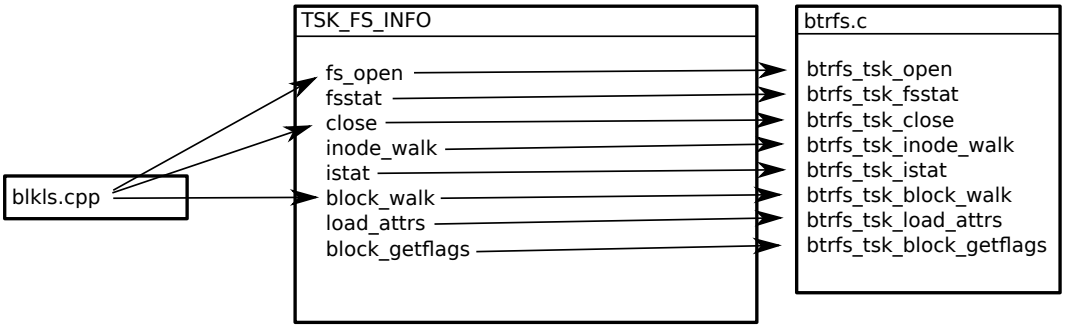

Figure 3.8: Usage of generic function pointers for blkls.

blkls This tool outputs the contents of all unallocated data units by default. Options to output the allocation status or outputting allocated data units are avaliable too [10, Appendix A].

blkstat This command outputs the allocation status of a specific data unit. [10, Appendix A]

TSK Filesystem API

The Sleuth Kit provides an API for implementing filesystem analysis code. The central entity that stores the state of an opened filesystem is the `TSK_FS_INFO` (see Figure 3.9) structure (tsk_fs.h). It keeps track of some general information such as start and size of the filesystem currently opened as well as the function pointers of functions to be implemented by filesystem specific code. The first handover of control from the TSK library code to filesystem specific code happens on opening an image (method `tsk_fs_open_img` in fs_open.c). This is the only place where the filesystem specific code is called directly. After opening the image, the filesystem specific code is called indirectly through function pointers defined in `TSK_FS_INFO`. This can be seen in an example for the blkls command in Figure 3.8.

The function pointers defined in `TSK_FS_INFO` are the main interface between the general TSK code and the filesystem specific code. What follows is a high level description of these methods.

fs_open This method is called after TSK has determined the basic information about the filesystem image: Its image format, contained partitions and other details hidden by the volume abstraction layer. The open method can rely on having access to the data of the filesystem. It is the first method from the filesystem specific code that is called by TSK. This function is a place for initializing things and creating and populating the `TSK_FS_INFO` struct. Typically it is also checked whether an image really has the right format: Usually filesystems define a specific signature which is checked by the open method.

fsstat This method prints general information about the filesystem to a file handle. Every filesystem implementation can print relevant information to the file handle without any con-

```
struct TSK_FS_INFO {
  int tag;
  TSK_IMG_INFO *img_info;
  TSK_OFF_T offset;

  /* meta data */
  TSK_INUM_T inum_count;
  TSK_INUM_T root_inum;
  TSK_INUM_T first_inum;
  TSK_INUM_T last_inum;

  /* content */
  TSK_DADDR_T block_count;
  TSK_DADDR_T first_block;
  TSK_DADDR_T last_block;
  TSK_DADDR_T last_block_act;
  unsigned int block_size;
  unsigned int dev_bsize;

  /* [...] */

  uint8_t (*block_walk)(TSK_FS_INFO * fs, TSK_DADDR_T start,
     TSK_DADDR_T end, TSK_FS_BLOCK_WALK_FLAG_ENUM flags,
     TSK_FS_BLOCK_WALK_CB cb, void *ptr);
  TSK_FS_BLOCK_FLAG_ENUM (*block_getflags)(TSK_FS_INFO * a_fs,
     TSK_DADDR_T a_addr);
  uint8_t (*inode_walk)(TSK_FS_INFO * fs, TSK_INUM_T start,
     TSK_INUM_T end, TSK_FS_META_FLAG_ENUM flags, TSK_FS_META_WALK_CB
      cb, void *ptr);
  uint8_t (*file_add_meta)(TSK_FS_INFO * fs, TSK_FS_FILE * fs_file,
     TSK_INUM_T addr);
  TSK_FS_ATTR_TYPE_ENUM (*get_default_attr_type)(const TSK_FS_FILE *)
     ;
  uint8_t (*load_attrs)(TSK_FS_FILE *);
  /* [...] */
  uint8_t (*istat)(TSK_FS_INFO * fs, FILE * hFile, TSK_INUM_T inum,
     TSK_DADDR_T numblock, int32_t sec_skew);
  TSK_RETVAL_ENUM (*dir_open_meta)(TSK_FS_INFO * fs, TSK_FS_DIR **
     a_fs_dir, TSK_INUM_T inode);
    /* [...] */
  uint8_t (*fsstat)(TSK_FS_INFO * fs, FILE * hFile);
  int (*name_cmp)(TSK_FS_INFO *, const char *, const char *);
  uint8_t (*fscheck)(TSK_FS_INFO *, FILE *);
  void (*close)(TSK_FS_INFO * fs);
  uint8_t (*fread_owner_sid)(TSK_FS_FILE *, char **);
};
```

Figure 3.9: `TSK_FS_INFO` from tsk_fs.h

straints. The output is used to get a broad idea about the filesystem under investigation. The information obtained here is usually not related to concrete data of the filesystem but rather the metadata of the filesystem. These things could include the version of the filesystem, used features, volume name, block size etc. The information is not passed internally to TSK, it is just printed out for the user. The printed data is usually read by fs_open and could be stored in the superblock or similar information blocks.

close This method closes the filesystem. This is a place to empty caches, free lists and similar.

inode_walk This method allows to iterate over a range of metadata addresses. Each metadata address is read from disk using the `file_add_meta` method. A specified callback decides whether to continue the iteration or to stop.

istat This method retrieves information about an inode and prints it out in textual form. It is similar to the `fsstat` method. Although its name contains *inode*, it is a general function which displays information on metadata addresses. As the `fsstat` method, it just prints information out for the user without internally storing the aggregation. Of course the read metadata could be stored in a structure, but the output of this method is just printed out.

block_walk This function allows TSK to iterate over the filesystem blocks. The argument is a disk block range which is specified in times of the block size. The block size is the size which is used by filesystems to internally address space in the filesystem. The `block_walk` method also has a callback parameter which is called for every block. The callback method recieves the block details in form of a `TSK_FS_BLOCK` struct and returns whether to continue reading the rest of the block range or if the process should be interrupted.

file_add_meta The method allows TSK to fetch the details of a metadata address (inode). It loads the required data from disk and populates the metadata attributes of a `TSK_FS_FILE` struct. Thus it is a mechanism to lazy-load file metadata information.

load_attrs This method is important for communicating file positions to the TSK API. An attribute is TSK's termininology for file content. The file content is represented through a list of data runs, each run describing a continuous area of space which is part of the file's content. The unit in which those file contents are declared is blocks. This means that each declared data run position needs to be a multiple of the block size. Smaller files can be represented as pointer to a memory area which carries the file's content.

get_default_attr_type This is mostly for NTFS compatibility as a file can have multiple data streams there. This method returns the default data stream for a filesystem. Usually this methods just returns a default value as most filesystems just have a single file content for each metadata address.

block_getflags This method is used to get information about a specific filesystem block. The information includes the allocation status (free, allocated) and the type of block (metadata, data). It is called by block_walk to gain information about a single block.

Limitations of the TSK API

Metadata Addressing As mentioned before the TSK API was not designed to support subvolumes. This simple fact becomes manifest in the way files are addressed in the API. TSK uses a so called *Metadata Address* to refer to metadata units (inodes in Ext and Btrfs terminology). This is a single value of type `TSK_INUM_T` (defined as unsigned 64bit number). This is not suitable for Btrfs because with TSK's implementation it is possible to address 2^{64} metadata units while Btrfs can address 2^{64} inodes in each subvolume, which is apparently a problem. A possible solution to bypass this limitation is the creation of an indirection layer to map virtual metadata addresses to real metadata addresses, thus simulating a single domain of addresses for TSK while in reality there are multiple independent domains of inodes. This is of course only possible under the assumption that the individual subvolumes in sum use less than 2^{64} inodes, which is such a high number that this workaround will work long enough to adapt the TSK API until such huge filesystems become relevant in future.

Subvolumes Related to the metadata addressing issue but in fact a different problem. Since subvolumes are not supported in TSK, there is no way to specify subvolumes for individual commands. In practice this means that it is not possible to list the files of a single subvolume or limit TSK commands to individual subvolumes. This problem can be worked around by using a virtual directory structure which puts files in a virtual prefix such as `/subvolume256/etc/fstab`. This virtual directory trick is already used in TSK for displaying deleted files. Altough specifying subvolumes on the command line is desireable for file layer operations (eg. listing files), it is not suitable for operations on other Btrfs trees which operate independently of the subvolume hierarchy such as listing the free space of an image. The free space is tracked for the whole volume and though limiting it to a subvolume is possible to implement, the results are dubious because files belonging to other subvolumes would appear as free space and thus make the wrong impression of being deleted/unreferenced. So the subvolumes seem to be more relevant for the file naming layer and the workaround of using virtual directories to represent the subvolumes seems feasible.

3.2 Forensic Methods

After the forensic description of the Btrfs filesystem in the previous chapter, it is now necessary to define the forensic methods to analyze Btrfs filesystem images in order to retrieve the desired artifacts. In a this step the the methods itself are the desired outcome. This section describes the methods itself in a high level way without any implementation details. To simplify the process of explaining the methods, a simple test filesystem has been created which will be used in this section to simplify the usage of examples in this section. It is a 1GiB Btrfs filesystem with a single subvolume with the id 5. Its contents are:

- the root directory with the inode number 256.
- a file named “testfile” having the inode number 257. The content is `Testfile contents`. Its size is 18 bytes.
- a directory named “testdir” having the inode number 258.
- a file named “file2” having the inode number 259. It’s content is `file2content`. Its size is 13 bytes.
- a file named “linux-3.7.tar” having the inode 260. It contains a tar of the linux kernel sources of version 3.7. Its size is 499036160 bytes.

The Btrfs driver in the Linux kernel is accompanied by several userspace tools to create the filesystem, mount it and among others the btrfs-debug-tree utility which prints the contents of the Btrfs trees of a volume. It’s output is used here to describe the forensic methods. It is noted that these methods are based upon the generic methods for resolving addresses and for reading the generic trees of Btrfs. It is not described how to iterate over a tree but rather what criteria is used for querying the tree. A matching element is a structure referenced by a key that fulfills the conditions stated. This makes the description of forensic methods better readable by not repeating technical details all over again.

Retriving the Filesystem Metadata (A1)

The relevant filesystem metadata for Btrfs includes the chunks of the filesystem, which is necessary to verify the address translation of the filesystem, the basic parameters of the filesystem such as the free size, allocated size and the addresses of the most relevant data structures of the filesystem. This includes the most important trees. Further it is relevant to retrieve the subvolume structure.

To get this information, the prime source is the superblock. This is analog to ExtX and involves reading the superblock. Since the location of the superblock is clearly defined, the relevant metadata can be extracted by parsing the superblock’s data. After the superblock has been interpreted, it is necessary to read the chunk tree to get a list of the chunks of the filesystem. This is done by reading the bootstrap data in the superblock first to get a working address translation and then starting to read the chunk tree at the translated physical address.

The subvolumes are stored in the root of roots tree. There is a Root Item entry for each subvolume. The entries for subvolumes differ from other tree entries in their object id. Every tree has a fixed object id, only subvolumes have ids that lie in the range $5 = x \,|256 \leq x <$ UINT64MAX.

To list all the subvolumes a simple query of the root of roots tree with $[?, \text{ROOT\_ITEM}, ?]$ is sufficient, the result must be filtered with above criteria.

For our example filesystem the only result is the first subvolume with the fixed id of five:

```
item 3 key (FS_TREE ROOT_ITEM 0) itemoff 2661 itemsize 439
    root data bytenr 132517888 level 0 dirid 256 refs 1 gen 21
```

```
 fs tree key (FS_TREE ROOT_ITEM 0)
leaf 132517888 items 23 free space 1997 generation 21 owner 5
fs uuid e5d14793-a874-4ddf-a3ba-0cbe06e5691b
chunk uuid 33bb6a1b-3492-4287-a9a4-c77fea8ca55a
item 0 key (256 INODE_ITEM 0) itemoff 3835 itemsize 160
    inode generation 3 size 56 block group 0 mode 40555 links 1
item 1 key (256 INODE_REF 256) itemoff 3823 itemsize 12
    inode ref index 0 namelen 2 name: ..
item 2 key (256 DIR_ITEM 982728850) itemoff 3785 itemsize 38
    location key (257 INODE_ITEM 0) type 1
    namelen 8 datalen 0 name: testfile
item 3 key (256 DIR_ITEM 2818265978) itemoff 3748 itemsize 37
    location key (258 INODE_ITEM 0) type 2
    namelen 7 datalen 0 name: testdir
item 4 key (256 DIR_ITEM 3645954615) itemoff 3705 itemsize 43
    location key (260 INODE_ITEM 0) type 1
    namelen 13 datalen 0 name: linux-3.7.tar
item 5 key (256 DIR_INDEX 2) itemoff 3667 itemsize 38
    location key (257 INODE_ITEM 0) type 1
    namelen 8 datalen 0 name: testfile
item 6 key (256 DIR_INDEX 3) itemoff 3630 itemsize 37
    location key (258 INODE_ITEM 0) type 2
    namelen 7 datalen 0 name: testdir
item 7 key (256 DIR_INDEX 4) itemoff 3587 itemsize 43
    location key (260 INODE_ITEM 0) type 1
    namelen 13 datalen 0 name: linux-3.7.tar
item 8 key (257 INODE_ITEM 0) itemoff 3427 itemsize 160
    inode generation 6 size 18 block group 0 mode 100644
    links 1
item 9 key (257 INODE_REF 256) itemoff 3409 itemsize 18
    inode ref index 2 namelen 8 name: testfile
item 10 key (257 EXTENT_DATA 0) itemoff 3370 itemsize 39
    inline extent data size 18 ram 18 compress 0
```

Figure 3.10: Filesystem Tree output of btrfs-debug-tree with the example filesystem

This entry states that the filesystem tree starts at the defined logical address and has a root inode with the number 256.

The artifact must then be presented in human readable form.

Retrieving the Filesystem Structure (A2)

The file and directory structure of Btrfs is stored in the filesystem tree (see page 16). This is the most important tree for operating on the file name layer. Every file has a unique number in the scope of a filesystem tree: the inode number. This number is used as the Object ID part of a key. The item type field of the key describes the actual structure that is referenced by the key. The offset field's meaning dependes on the type of the referenced structure. Thus the Filesystem tree contains keys of the scheme $[k, x, y]$ for an inode k. This means that every structure related to a certain inode is located next to each other because of the ordering keys.

The first step of listing the filesystem structure includes the finding of the root inode. This is stored in the root of roots tree which contains references to every tree. It contains an Root Item entry with a dirid of 256 which denotes that the inode 256 is the root inode of this filesystem tree. The filesystem tree is referenced by the Root Item entry by the block number field. This number is a logical address which is translated to a physical address. At this address the tree starts with it's header structure.

```
item 3 key (FS_TREE ROOT_ITEM 0) itemoff 2661 itemsize 439
    root data bytenr 132517888 level 0 dirid 256 refs 1 gen 21
```

Having found the inode number of the root inode, it is possible to query for it's contents. Therefore a listing based on the inode number is possible by querying the filesystem tree for inodes with the number 256. This results in the following list of items:

```
item 0 key (256 INODE_ITEM 0) itemoff 3835 itemsize 160
    inode generation 3 size 56 block group 0 mode 40555 links 1
item 1 key (256 INODE_REF 256) itemoff 3823 itemsize 12
    inode ref index 0 namelen 2 name: ..
item 2 key (256 DIR_ITEM 982728850) itemoff 3785 itemsize 38
    location key (257 INODE_ITEM 0) type 1
    namelen 8 datalen 0 name: testfile
item 3 key (256 DIR_ITEM 2818265978) itemoff 3748 itemsize 37
    location key (258 INODE_ITEM 0) type 2
    namelen 7 datalen 0 name: testdir
item 4 key (256 DIR_ITEM 3645954615) itemoff 3705 itemsize 43
    location key (260 INODE_ITEM 0) type 1
    namelen 13 datalen 0 name: linux-3.7.tar
item 5 key (256 DIR_INDEX 2) itemoff 3667 itemsize 38
    location key (257 INODE_ITEM 0) type 1
    namelen 8 datalen 0 name: testfile
item 6 key (256 DIR_INDEX 3) itemoff 3630 itemsize 37
```

```
    location key (258 INODE_ITEM 0) type 2
    namelen 7 datalen 0 name: testdir
item 7 key (256 DIR_INDEX 4) itemoff 3587 itemsize 43
    location key (260 INODE_ITEM 0) type 1
    namelen 13 datalen 0 name: linux-3.7.tar
```

The resulting inodes contain the Inode Item of the root inode with its corresponding inode ref. The Inode Ref is the name entry for an inode. In the case of the root directory is "..". The contents of the directory are represented by pairs of Dir Item and Dir Index entries. Dir Index items always contain one entry and their key is the index of the file in the current directory. So the key $[256, \text{DIR\_INDEX}, 3]$ means that the directory inode is 256, the item is a Dir Index and the element is the third element in this directory. So the Dir Index elements allow access of files by their index. This is useful for directory listings and similar operations. The Dir Item entries work different. They allow the lookup of files by by their names. Therefore the key $[256, \text{DIR\_ITEM}, 982728850]$ means that there is a file in the directory with the inode number 256 which is a Dir Item and it's filename matches the hash number 982728850. Of course with hash numbers (especially short ones) there can be collisions. Therefore Dir Items can reference multiple files if their hashes are the same. The struct is the same as a Dir Index, but can be repeated. The Dir Index and Dir Item elements have a type field which specifies the type of the file (directory, regular file, devices, etc.).

At this point everything needed for a file listing is already present. The list of files in a directory among their types and their inode numbers. It is possible to recursively list all the files in a filesystem by iterating over every found directory and to query it after the same scheme as used above. If we were interested to find the file "file2", we would have to query the filesystem tree for elements with key $[258, ?, ?]$ ("testdir"). This would result in similar results as above.

The data we queried above does not contain very detailed information about files though. To get more details of a file, it is necessary to fetch their individual Inode Item elements by querying for $[x, \text{INODE\_ITEM}, 0]$, x being the inode number of the element.

Retrieving the File Metadata (A3)

File metadata is stored in Inode Items in Btrfs, they don't share much of the concepts that inodes have in other Unix filesystems besides the name. In ExtX an inode can be described as:

> The metadata for each file and directory are stored in a data structure called an inode, which has a fixed size and is located in an inode table. [10, Chapter 14]

The difference is the following: In ExtX there is a single domain of inodes which are stored in an inode table. Accessing an inode is very simple due to the fixed size of the entry and the well-defined start of the inode table. It is simply a multiplication of the inode number with the inode size that results in the offset where the inode is located in the inode table. In Btrfs there is no such thing as a single domain of inodes as every subvolume contains subvolumes which have the same numbers but refer to different files. Also since there is no table which stores inodes in a fixed location, it is not possible to find deleted inodes by a simple lookup in the inode table. It

requires tree forensics (or brute force methods with good heuristics) to find unreferenced Inode Items in a filesystem tree. So it is necessary to limit the list of inodes to referenced inodes.

The filesystem table contains Inode Items for every allocated inode in the subvolume. Having found the richt filesystem tree makes it possible to query the tree for elements matching $[?, \text{INODE\_ITEM}, 0]$. The result list contains every item that is allocated in the current filesystem tree. In our example filesystem this would find the following elements:

```
item 0 key (256 INODE_ITEM 0) itemoff 3835 itemsize 160
    inode generation 3 size 56 block group 0 mode 40555
    links 1
item 8 key (257 INODE_ITEM 0) itemoff 3427 itemsize 160
    inode generation 6 size 18 block group 0 mode 100644
    links 1
item 11 key (258 INODE_ITEM 0) itemoff 3210 itemsize 160
    inode generation 17 size 10 block group 0 mode 40755
    links 1
item 15 key (259 INODE_ITEM 0) itemoff 2963 itemsize 160
    inode generation 17 size 13 block group 0 mode 100644
    links 1
item 18 key (260 INODE_ITEM 0) itemoff 2754 itemsize 160
    inode generation 21 size 499036160 block group 0 mode
    100644 links 1
```

If it is necessary to know the type of the inode (regular file, directory, etc.), more queries must be made for every inode. The type is only stated in the corresponding Dir Index or Dir Item elements. The problem is that those elements are not referenced by a key containing the actual file's inode number but rather the inode number of the directory containing the file. Therefore it is necessary to fetch the Inode Ref element with the query $[x, \text{INODE\_REF}, y]$, x being the inode number of the file, y matches everything. The found element contains the inode number of the parent directory in y. So it is possible to get the Dir Index by querying for $[y, \text{INODE\_INDEX}, ?]$.

Retrieving the Allocation Status (A4)

When iterating over the blocks of the image, one actually iterates over physical addresses. These physical addresses can have a logical address defined or not. The first step of validating if a block is allocated is the resolving of the logical address. If it is not defined, the block is unused because it cannot be addressed by the filesystem. If the address exists, it could be allocated or unallocated.

The whole space of a Btrfs volume is managed in the subvolume independent extent tree. It contains Extent Item elements whose keys have the meaning [Logical Address, EXTENT_ITEM, Bytes]. So we query the extent tree for keys matching [Logical Address $\leq a$, EXTENT_ITEM, (Logical Address + Bytes) $\leq a$], with a being the logical address of interest.

If the query has results, that means that the start address of the block is used by the filesystem. The type field of the Extent Item states whether the extent is used by a tree or by data. The structure continues with back references to the actual data, which is not needed here.

The Extent Item entries for the large tar archive in the test filesystem are listed below. One can see that each part of the file is referenced by it's logical address and size.

```
item 45 key (336527360 EXTENT_ITEM 150798336) itemoff 1647
    itemsize 53
    extent refs 1 gen 21 flags 1
    extent data backref root 5 objectid 260 offset 0 count 1
item 47 key (541327360 EXTENT_ITEM 150798336) itemoff 1570
    itemsize 53
    extent refs 1 gen 21 flags 1
    extent data backref root 5 objectid 260 offset 150798336
    count 1
item 49 key (746127360 EXTENT_ITEM 197439488) itemoff 1493
    itemsize 53
    extent refs 1 gen 21 flags 1
    extent data backref root 5 objectid 260 offset 301596672
    count 1
```

Retrieving the Contents of Allocated Files (A5)

To get the content of a file, it is necessary to find the filesystem tree that contains the file. The file is then referenced by it's inode number. Since the filesystem tree contains Extent Data items for every file, we start by querying them with the filter $[x, \text{EXTENT\_DATA}, ?]$. This results in the following entries for the example filesystem:

```
item 10 key (257 EXTENT_DATA 0) itemoff 3370 itemsize 39
    inline extent data size 18 ram 18 compress 0
item 17 key (259 EXTENT_DATA 0) itemoff 2914 itemsize 34
    inline extent data size 13 ram 13 compress 0
item 20 key (260 EXTENT_DATA 0) itemoff 2678 itemsize 53
    extent data disk byte 336527360 nr 150798336
    extent data offset 0 nr 150798336 ram 150798336
    extent compression 0
item 21 key (260 EXTENT_DATA 150798336) itemoff 2625
    itemsize 53
    extent data disk byte 541327360 nr 150798336
    extent data offset 0 nr 150798336 ram 150798336
    extent compression 0
item 22 key (260 EXTENT_DATA 301596672) itemoff 2572
    itemsize 53
    extent data disk byte 746127360 nr 197439488
    extent data offset 0 nr 197439488 ram 197439488
    extent compression 0
```

The result contains five items for the three files we stored in the example filesystem. The first two Extent Data items are inline extents. This is marked in the type field of the structure. This means that the contents of the file are so small that the file can be placed directly inside the filesystem tree.

The inline extent's file contents simply follow the Extent Data structure. The length of the data is specified by the size field of the structure and is simply read.

For the larger file with the inode number 260 there are three Extent Data elements found. This means that the contents of the file are divided into three continuous parts (extents) on the disk. The key marks the file offset with the offset field. The first Extent Data item starts at offset zero, the beginning of the file. The structure specifies the location of the data along with the length. The other Extent Data's offset fields in the key are larger. To read the data of the file, the Extent Data elements are iterated over in the order of their appearance in the filesystem tree. For each element the physical address is resolved using the logical address stored inside the file. The data is then read at this location until the length specified in the element is reached. Then the process is repeated.

Retrieving the Contents of Unallocated Files (A6)

To get the contents of unallocated files, the position of them inside the filesystem must be known. Since the position is unavailable because the file is deleted and thus the relevant trees don't reference any items that describe the unallocated file, there is no ad-hoc way to implement this retrieval except the application of tree forensics to get back to an elder state of the relevant trees in which the file existed.

Since the allocation status of data units (A4) has been described, there is another way to get the contents of unallocated files: since the allocation status of the data units is known, allocated data units can simply be excluded from the set of all data units. With the remaining unallocated data, it is possible to apply a heuristic approach to get files out of the stream of unallocated data. Using file signatures makes it possible to find unallocated data if it is not overwritten yet. Since this is also a limitation of the tree forensic approach (even knowing the exact location of a unallocated file doesn't help if that location has been overwritten), this is no big disadvantage.

3.3 Implementation of the Btrfs Code

As described previously, there is a discrepancy between the metadata addressing in TSK and its implementation in Btrfs: The SleuthKit is was not designed to handle filesystems with multiple subvolumes. It relies on a single domain of inode numbers to work. In Btrfs there is a domain of inode numbers for each subvolume. So the open method maps all present inodes of every subvolume to a virtual inode number that is visible for all TSK methods. This way the uniqueness of inode numbers is guaranteed at the cost of the additional overhead of iterating over all inodes of every subvolume. This has some disadvantages, but is a workaround which is feasible for average case filesystems.

The Btrfs implementation covers three files

btrfs.c The algorithms and implementations of the filesystem specific methods as well as helper methods.

btrfs_io.c Methods to read structs in from disk and to print out information.

tsk_btrfs.h Header file.

Most of the interesting code is found in btrfs.c while btrfs_io.c only holds quite repetitive input and output code which is used in the algorithms. The header file defines the method signatures and structs used in the Btrfs implementation.

The implementation of a new filesystem in TSK consists of the implementation of the before listed methods. They are called by TSK and the implemented features of TSK keep working independently of the filesystem implementation. Below follows a list of the key methods which were implemented along with a short description about their functioning.

Method `btrfs_tsk_open`

The open method (see Figure B.1) starts by allocating and initializing the `BTRFS_INFO` struct which stores the state and is passed to nearly every method. In an object oriented programming language the members of the struct would be member variables in a Btrfs class. The `BTRFS_INFO` struct internally stores the `TSK_FS_INFO` struct at the beginning and is thus usable as `TSK_FS_INFO`. This is a simple replacement for inheritance in object oriented languages.

After initializing the struct the method continues by verifying that the opened image really is a Btrfs image which is done by reading the superblock first (`btrfs_io_read_superblock_pa`) and then checking for the Btrfs magic string in the superblock. The superblock stores most of the things needed to initialize the `TSK_FS_INFO` struct and is also stored inside the struct.

The superblock data is then used to continue initializing the `TSK_FS_INFO` struct. After this has been done, the bootstrapping data from the superblock is read into the chunk list (`btrfs_chunk_read_superblock_bootstrap_data`). With that the address lookup mechanism is able to read the entries of the chunk tree, which is done in the next step: The entire chunk tree is read into the chunk list (`btrfs_chunk_read_tree`) which is used by the address resolving logic afterwards. After the chunk tree is read, the address lookup works for every address in the filesystem.

The next step is the initialization of the root-, extent- and dev tree. Those trees are independent from subvolumes and are needed by other methods. The initialization only stores the parameters to read the tree later on. The dev tree is read into memory (`btrfs_read_dev_tree`) since it is used for the translation of physical addresses into logical addresses.

After reading those trees, the subvolumes are read and their contents are counted to create the virtual inode mapping which is later on used to identify every inode in every subvolume. This process can be time consuming depending on the size of the subvolumes.

The last step is the setup of the generic function pointers which are used by TSK to call the filesystem specific methods.

(see Figure B.2).

Method `btrfs_tsk_fsstat`

The fsstat method only relies upon data read in by the `btrfs_tsk_open` method. It simply prints out the information for the user and then returns (see Figure B.3). There is no special format or convention to follow. This method is free to print whatever seems important about the opened filesystem.

In the implementation the printed details are:

- Total bytes of the filesystem
- Used bytes
- Logical addresses of chunk tree root and root tree root
- Tree parameters as nodesize, sectorsize
- Subvolume information as their count and the id of every subvolume
- A listing of the cunk tree

The method fsstat is used by the command line tool fsstat which allows users to print meta information about a filesystem.

```
$ fsstat ~/Diplomarbeit/testdata/fs1
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Btrfs
Total Bytes: 2048000000
Bytes Used: 500068352
Logical address of chunk tree root: 20971520
Logical address of root tree root: 132575232
nodesize: 4096
sectorsize: 4096
number of subvolumes: 1
subvolume id: 5
number of inodes: 5
Chunk Tree:
*** chunk entries ***
KEY<object_id: 256, item_type: 228 (CHUNK_ITEM), offset: 0>
CHUNK_ITEM<size_of_chunk: 4194304, owner: 2, stripe_len:
65536, type: 2 (SYSTEM), io_align: 4096, io_width: 4096,
sector_size: 4096, num_stripes: 1, sub_stripes: 0>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 0>
KEY<object_id: 256, item_type: 228 (CHUNK_ITEM), offset:
20971520>
CHUNK_ITEM<size_of_chunk: 8388608, owner: 2, stripe_len:
```

```
65536, type: 34 (SYSTEM,MIRRORED), io_align: 65536,
io_width: 65536, sector_size: 4096, num_stripes: 2,
sub_stripes: 0>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 20971520>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 29360128>
KEY<object_id: 256, item_type: 228 (CHUNK_ITEM), offset: 0>
CHUNK_ITEM<size_of_chunk: 4194304, owner: 2, stripe_len:
65536, type: 2 (SYSTEM), io_align: 4096, io_width: 4096,
sector_size: 4096, num_stripes: 1, sub_stripes: 0>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 0>
```

Method `btrfs_chunk_read_superblock_bootstrap_data`

This method reads the bootstrap data of the superblock and inserts it into the BTRFS_INFO struct (see Figure B.4). The storage is in form of a generic List (TAILQ, from BSD). This helps to reduce code duplication and strives to a better readablity of the code. This method calls `btrfs_chunk_add` as a helper method.

Method `btrfs_chunk_read_tree`

This method reads the entire chunk tree into the same list as the bootstrap data before (see Figure B.5). Reading the tree in background requires the presence of the bootstrap data. The helper method `btrfs_chunk_add` is used as before. The method `btrfs_tree_list` is a shorthand for `btrfs_tree_list_filter` which finds every node.

Method `btrfs_chunk_add`

This method contains the logic to read a chunk structure from disk and to to insert it into a list (see Figure B.6). Reading a chunk structure includes reading the chunk item along with the variable number of chunk item stripes. The data is stored in a TAILQ list of `chunk_entry_s`.

Method `btrfs_chunk_print_entries`

This method is used to print the contents of the chunk tree. The resulting output can be used to manually translate virtual addresses to physical addresses and for information purposes as it is done for the fsstat method.

Method `btrfs_read_dev_tree`

Reads the dev tree into a list of dev extent entries (see Figure B.7). This tree is required for the translation of physical into logical addresses. The data is stored in a TAILQ list of `dev_extent_entry_s` elements. The actual reading is done by the method `btrfs_tree_list` which reads a tree into a result list which is then iterated over and stored in the list.

Method `btrfs_resolve_logical_address`

This method uses the previously read chunk data and allows to translate a logical address into a physical address (see Figure B.8). This is simply done by iterating over the list of chunk entries (`chunk_entry_s`), finding the right entry and then calculating the physical address. Every logical address needs to be translated to a physical address to read from that location.

A possible improvement would be the in-memory storage of this list in form of a tree to find the results faster. This has been left for later improvement as it is a sole performance optimization.

Method `btrfs_resolve_physical_address`

This method is the reverse of the previous one (see Figure B.9). It translates a physical address to a logical one. This time the dev tree is used, previously read into a list for performance reasons (`btrfs_read_dev_tree`). The process is the same: finding the right entry and then calculating the logical address and returning it.

As stated in the method description of `btrfs_resolve_logical_address`, the use of a tree could improve performance.

Method `btrfs_tsk_istat`

The istat method prints information about an inode to stdout (see Figure B.10). Similar to the `btrfs_tsk_fsstat` method, there is no special convention to follow in the process. Anything relevant can be printed here. The implementation prints the metadata of the inode as well as additional information:

1. virtual inode number
2. real inode number and subvolume
3. file size
4. access-, modified- and create time

This function is used mainly in the command line tool istat. In this example below, the metadata information for inode 4 is printed to the user by invoking the istat command:

```
$ istat filesystem.img 4
Inode number 4 (virtual)
Btrfs inode number is 260 on subvolume 5
=== stat info ===
Size:  499036160
Access time:  1366612191
Modified time:  1366612199
Create time:  0
```

Method `btrfs_tsk_inode_walk`

Iterating over a range of inodes is implemented in this method (see Figure B.11). It uses the `tsk_inode_lookup` method to load inodes from disk and to call the callback method with the result.

This method is used by the ils command line tool to generate a listing of inodes in a filesystem. An invocation of the ils command generates a list of inodes that were found on the image along with specific metadata. An example follows:

```
$ ils filesystem.img
class|host|device|start_time
ils|andreas-desktop||1391182492
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|
st_crtime|st_mode|st_nlink|st_size
0|a|0|0|1366612191|1366612149|0|1366612191|0|0|56
1|a|0|0|1347964140|1353402818|0|1347964140|0|0|18
2|a|0|0|1353404619|1353404607|0|1353404619|0|0|10
3|a|0|0|1353404619|1353404619|0|1353404619|0|0|13
4|a|0|0|1366612199|1366612191|0|1366612199|0|0|499036160
```

Method `btrfs_tsk_block_walk`

Iterating over a range of blocks is similar to the previous method (see Figure B.12). It iterates over the range and reports to a callback method which decides whether to continue iterating. Additionally there is a filter built in which decides whether to call the callback. The actual data is read by `btrfs_tsk_block_getflags` which returns the flags (free, used, etc.).

This method is used by the command blkls which is used to output the data of filesystem blocks and provides options to filter the data whether the block is used or free and other criteria.

Method `btrfs_tree_search`

The trees in Btrfs are generic and completely independent of what data they store. This means that the code for traversing trees and searching data needs to implemented only once and can be used for every tree in the filesystem (see Figure B.13). A tree usually is traversed from its top level and then searched until the leaf level is reached (level 0). The inner (non-leaf) nodes are stored as block pointers which point to another position in the tree. In every inner level the key which is greater than the target key is searched and iteration continues at the block pointer before this node. The method `btrfs_tree_search` is called recursively for each level. The leaf level (level 0) is special since it stores the actual data. The result of the method is always a single entry. The method takes a compare function as parameter which is used to determine whether the key if found or not.

Method `btrfs_tree_list_filter`

This method is comparable to `btrfs_tree_search` but searches for multiple results (see Figure B.14). It uses a compare function to determine whether a node is to be included in the results or not. So it filters the contents of a tree compared to the search in the previous method. This is a practical feature for reading in trees like the chunk tree. Here it is easier to fetch the whole tree instead of doing a search every time a logical address needs to be translated. This is also a matter of performance.

Method `btrfs_cmp_func_exact`

This function is one implementation of the cmp_func signature which is used as comparator for keys (see Figure B.15). Thus it defines an ordering for the keys which is expressed through the return value: an integer value which is zero when both keys are equal and greater than or smaller than zero when one of both is bigger. Among the implementations is a method which ignores the offset (`btrfs_cmp_func_exact_ignore_offset`), a method which matches every key (`btrfs_cmp_func_match_all`) which can be used to get the whole tree with the `btrfs_tree_list_filter` method among some other implementations.

Method `btrfs_subvolume_init`

Since a volume can have more than one subvolume, this searches the root tree for references of subvolumes. Each found subvolume is then added into several arrays for easy access by id, storage of the tree information and an in-memory list of the subvolume tree contents. A convenience method `btrfs_subvolume_get_by_id` is used to return the right subvolume tree by id.

Method `btrfs_inode_create_mapping`

Due to the beforehand mentioned addressing problem of metadata entries, the used workaround is the creation of a virtual inode mapping which assigns every real inode a virtual inode number. This virtual inode number is then used in communication with TSK. The mapping data is stored in an array with the size of the total number of inodes. The mapping itself is a struct that contains the subvolume id and the inode number. So the mapping is from the array position which is used as virtual inode number to the real inode number and the subvolume it is to be found.

Method `btrfs_inode_resolve`

This method fetches the inode mapping data from the storage array created by `btrfs_inode_create_mapping`. It does some additional error checking and returns the data from the array.

Method `btrfs_inode_resolve_reverse`

This method is is the counterpart of `btrfs_inode_resolve`. It translates a real inode mapping to a virtual address. Since there is no index or similar, the whole list needs to be searched

in memory. This can be optimized in future.

Method `btrfs_tsk_close`

This is the last method being called by TSK. This method is for closing and freeing everything that was opened and allocated before. This includes the list of chunk entries, the list of subvolume trees, the dev tree and others.

Method `btrfs_dir_open_meta`

This method opens a directory and reads its contents into a result struct. Since filesystems are hierarchically storing information, each directory can contain directories itself. With the presence of this method, a recursive listing of a filesystem can be achieved. This methods only returns the directory contents of a single directory, but iterating over the directories in its result leads to a complete listing of the hierarchy of the filesystem. This method is used by the fls tool for the sake of displaying the filesystem contents that are allocated and unallocated. In the case of this implementation only allocated entries are returned since tree forensics is not used in this implementation.

Method `btrfs_tsk_block_getflags`

This method determines the metadata of a filesystem block. Filesystem blocks can be either allocated or unallocated. The method starts by calculating the physical address of a block which is simple multiplication of the block address (number) with the block size. This physical address is then translated into a logical address which is used to query the extent tree for entries with that address. Since the extent tree contains the allocation status of every allocated address, a positive match (an extent item) means that the block is allocated. Furthermore it can be determined which type the block has since the extent item has a flag attribute which describes the usage of the allocation (data, metadata). The method can thus return if a block is allocated with data or metadata or if it's unallocated.

Method `btrfs_tsk_inode_lookup`

This method finds an inode based on its virtual inode number. The virtual inode is translated to a physical inode number and subvolume which is used to get the right subvolume tree. The tree is then searched for matching inode item and inode ref and dir index elements. Those elements are then used to populate the `TSK_FS_FILE` struct which contains the meta information about the inode. This process includes the translation of Btrfs specific file types into TSK specific file types for special files such as devices, pipes and others.

Method `btrfs_tsk_load_attrs`

This method is used to construct a `TSK_FS_ATTR` struct which contains information about the location of the attributes of a file. An attribute can roughly be said to be the content of a file. The attribute consists of multiple runs which represent the parts of the file.

Since the method argument is a virtual inode number, the first step is resolving it. Afterwards the matching extent data item is searched in the filesystem tree for the particular subvolume of the inode. If the file is rather small, it could be stored in an inline extent which is determined by the type field of the extent data. Inline extents are stored as a pointer in the `TSK_FS_ATTR` struct. Normal extents are converted to data runs: each of the found extent data items corresponds to a part of the file which is then translated into a data run. The actual conversion is done by the method `btrfs_convert_extent_data_to_data_run`.

Method `btrfs_convert_extent_data_to_data_run`

The actual conversion from an extent data item to a data run is done in this method. It takes a extent data item as an argument and returns a `TSK_FS_ATTR_RUN` struct. The first step is resolving the logical address in the extent data to a physical address. The extent data also has an offset stored in it's key to mark the position inside the file which is stored inside the actual extent. The values of the extent data item are simply recalculated to fit the block-based values of the data run. This is a workaround since blocks don't exist in btrfs, but since extents are usually page-aligned, it is possible to describe the location of individual files in terms of blocks:

The start block of an extent item is calculated as the physical address of the extent divided by the block size. The block length is the division of the size in bytes by the block size, one is added if the rest is not zero. The offset of the data run is copied from the extent data, since it is not defined in blocks but bytes.

Parse Methods

Btrfs uses a lot of data structures on disk. Reading these structure form disk into memory requires consideration of endian-ness, differences in the representation of structures in-memory, compiler optimizations and so on. A concrete example why reading data into a char array and casting it to the struct doesn't work (assuming the same endianness) is the alignment optimization of the compiler. The compiler aligns data structures for performant access (depending on compiler flags or possible optimization hints in the source code) which means that casting the data directly to the struct will (depending on the fields of the struct) probably fail. It has to be said that that compiler behavior can be disabled, but it is not recommended (Btrfs source code does exactly that anyways). The other way is to read data in with methods that read the char data and write it to the fields of the struct. In btrfs_io.c many such methods exist and they do the work of translating on-disk data streams into structs. An example can be seen in fig. B.16. The helper method `btrfs_io_read_field` reads data from the source char array to the struct's field, considering length and calculating the current position in the source array. This way the code is quite compact.

Read Methods

Data stored on disk must be read into in memory data structures at some point. This is done by helper methods in btrfs_io.c (see Figure B.17). Usually two such methods exist: one for reading at a logical address and a second one for reading data from a physical address. The first method

includes the translation step from logical to physical addresses and calls the second method. The implementation otherwise is quite simple and just includes reading the data into an array of the correct size and using translating it to the target struct using the parse method.

Print Methods

The print methods allow the printing of structs to human readable characters. It some helper methods to format the data and outputs a string in the convention

```
STRUCT_NAME<field1: value1, field2: value2>
```

Nesting of data structures is possible too. The print methods were used exhaustively during development and still serve the purpose to debug the program.

CHAPTER 4

Evaluation

This chapter evaluates the implementation described in the previous chapter according to the defined artifacts in the introduction. Each artifact is evaluated either manually with the performed steps defined in the evaluation or automatically (either using a JUnit based test or some sort of script). The goal of each evaluation is to see to what extent the artifact can be retrieved from a Btrfs filesystem. The chapter finishes with possible future work that can be made in the context of this thesis.

4.1 Test Environment

Hardware As hardware is usually not a factor that influences the filesystem's on-disk data, as that is abstracted by the operating system, it doesn't need to be specified in detail.

Software The test data needs of course to be created by a Linux based operating system which doesn't have incompatible changes to the TSK implementation. The choice was made arbitrarly by personal preference of the author: Because the linux distributions usually don't change filesystem implementation in their distribution kernels, the question which distribution to use for testing is irrelevant. Debian Wheezy 7.4 amd64 is used.

Filesystem Images For testing purposes small (1-2GiB) sparse files are used. They have the advantage of being created very fast because the underlying filesystem (Ext4 in the test environment) keeps track of the ranges which consist only of zeroes. These images are then formatted to Btrfs and the individual tests are carried out upon these images. Due to the lack of existing reference Btrfs images suitable for forensic analysis, the images need to prepared by the author for the individual tests. This is an additional effort and also has some issues (validation is harder, the dataset could be non-representative, etc.) [20]. In absence of a better alternative, self-made test images need to be used for the evaluation though.

Automated Test Suite To be able to reproduce the tests, a Java-based JUnit test environment was created for some evaluations. This test environment automatically creates test images, formats it to Btrfs, copies test data in it and carries out the functionality tests. This makes it possible to verify the functionality of the Sleuthkit implementation by invoking `mvn clean install`.

4.2 Filesystem Metadata Retrieval (A1)

The goal of this evaluation is to confirm the suitability of the filesystem metadata retrieval implemented in TSK.

The retrieval of the filesystem metadata is related to the Sleuthkit tool fsstat. This tool is used to display general metadata of filesystems. The evaluation here is done manually by calling the command. Executing fsstat leads to the following result (see Figure 4.1). Most chunk items are skipped here for brevity. This output is comparable to the fsstat output of an Ext3 filesystem (see Figure 4.2).

This output includes the most relevant parameters of the filesystem metadata: The filesystem type, the size of the filesystem, the used bytes, the addresses of the most important trees, the tree size parameters, the subvolume structure and the chunk entries.

4.3 Filesystem Structure Retrieval (A2)

The evaluation's goal is to verify that the forensic methods and implemented procedures are able to reproduce the structure of the filesystem as it exists in a disk image.

This evaluation operates only on metadata and compares the contents of the test image to the reference data. This test operates on known files since no tree forensics are used in this implementation. The used Btrfs data structures include the chunk tree (required for address translation) and the filesystem tree.

Automated Testing

This functionality is related to Sleuthkit's fls tool. The test procedure is as follows:

1. Creation of an empty sparse file
2. Formatting the image to Btrfs
3. Mounting the image
4. Copying a directory of test data to the mount point
5. Umounting the image
6. Run `fls -f btrfs -r -m / -a` on the test image
7. Comparing the pathes for equality

```
$ fsstat ~/Diplomarbeit/testdata/fs1
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Btrfs
Total Bytes: 2048000000
Bytes Used: 500068352
Logical address of chunk tree root: 20971520
Logical address of root tree root: 132575232
nodesize: 4096
sectorsize: 4096
number of subvolumes: 1
subvolume id: 5
number of inodes: 5
Chunk Tree:
*** chunk entries ***
KEY<object_id: 256, item_type: 228 (CHUNK_ITEM), offset: 0>
CHUNK_ITEM<size_of_chunk: 4194304, owner: 2, stripe_len:
65536, type: 2 (SYSTEM), io_align: 4096, io_width: 4096,
sector_size: 4096, num_stripes: 1, sub_stripes: 0>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 0>
KEY<object_id: 256, item_type: 228 (CHUNK_ITEM), offset:
20971520>
CHUNK_ITEM<size_of_chunk: 8388608, owner: 2, stripe_len:
65536, type: 34 (SYSTEM,MIRRORED), io_align: 65536,
io_width: 65536, sector_size: 4096, num_stripes: 2,
sub_stripes: 0>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 20971520>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 29360128>
KEY<object_id: 256, item_type: 228 (CHUNK_ITEM), offset: 0>
CHUNK_ITEM<size_of_chunk: 4194304, owner: 2, stripe_len:
65536, type: 2 (SYSTEM), io_align: 4096, io_width: 4096,
sector_size: 4096, num_stripes: 1, sub_stripes: 0>
CHUNK_ITEM_STRIPE<device_id: 1, offset: 0>
```

Figure 4.1: An execution of fsstat with the example Btrfs filesystem.

```
FILE SYSTEM INFORMATION
--------------------------------------------
File System Type: Ext3
Volume Name:
Volume ID: 3730fc488dc71f9e294f36336e5c0e17

Last Written at: Fri Mar  7 19:55:01 2014
Last Checked at: Wed Apr  3 19:17:11 2013

Last Mounted at: Sun Feb 16 11:04:08 2014
Unmounted properly
Last mounted on: /mnt/ext

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

Journal ID: 00
Journal Inode: 8

METADATA INFORMATION
--------------------------------------------
Inode Range: 1 - 61038593
Root Directory: 2
Free Inodes: 60937007

CONTENT INFORMATION
--------------------------------------------
Block Range: 0 - 244140287
Block Size: 4096
Free Blocks: 35639906

BLOCK GROUP INFORMATION
--------------------------------------------
Number of Block Groups: 7451
Inodes per group: 8192
Blocks per group: 32768
```

Figure 4.2: An execution of fsstat on an Ext3 filesystem.

The algorithm for comparing the pathes is very simple: It iterates through the first list of pathes and tries to find a matching element in the second list of pathes. When no element is found, false is returned. On a match the element is removed from the second list. The lists are equal when a match for every element in the first list was found and the second list is empty.

Result Executing the automated test confirms that the structure read by the fls tool is equal to the test data structure. Thus this evaluation's result is clearly positive.

4.4 File Metadata Retrieval (A3)

File Metadata retrieval is handled by the TSK tools istat and ils. Istat outputs the metadata in text form for a single inode while ils prints all found inodes in a list. The data used by those tools are read from the filesystem tree (Inode Items).

The artifact file metadata is related to stat information, so the easiest way to verify the implemented procedures is to start with the known stat information of a set of testfiles:

1. Saving the stat information of a known set of test files.
2. Creation of an empty Btrfs test image.
3. Mounting the image.
4. Copying the test files to the mounted image (`cp -a`, the flag preserves all necessary stat information).
5. Retrieving the metadata of the files with the TSK tools (fls, istat, ils).
6. Comparing the metadata.

With the help of the ils tool, the metadata of all files can be displayed. The stat information can be formatted like the output of ils by specifying the right output format (`stat -c '%i|a|%u|%g|%Y|%Z|%X|%W|%a|%h|%s' *`). With the output it is easy to compare the metadata.

Output of the stat command for the test files (inode numbers omitted):

```
$ stat -c '%i|a|%u|%g|%Y|%Z|%X|%W|%a|%h|%s' *
|a|1000|1000|1390151451|1390151451|0|1393685775|644|1|7859002
|a|1000|1000|1390151451|1390151451|0|1393685775|644|1|5412610
|a|1000|1000|1390151451|1390151451|0|1393685775|644|1|4046367
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|18063125
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|4084421
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|893341
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|8551922
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|4922217
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|6913192
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|51300889
|a|1000|1000|1390151451|1390151451|0|1393685776|644|1|102312577
```

Output of the ils command for the test image:

```
$ ils ~/Diplomarbeit/testdata/istat-test
class|host|device|start_time
ils|andreas-desktop||1393686048
st_ino|st_alloc|st_uid|st_gid|st_mtime|st_atime|st_ctime|
st_crtime|st_mode|st_nlink|st_size
0|a|0|0|1393685776|1347963804|1393685776|0|40555|0|918
1|a|1000|1000|1390151451|1392540704|1393685775|0|100644|0|7859002
2|a|1000|1000|1390151451|1392540704|1393685775|0|100644|0|5412610
3|a|1000|1000|1390151451|1392540704|1393685775|0|100644|0|4046367
4|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|18063125
5|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|4084421
6|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|893341
7|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|8551922
8|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|4922217
9|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|6913192
10|a|1000|1000|1390151451|1392540704|1393685776|0|100644|0|51300889
11|a|1000|1000|1390151451|1392540704|1393685777|0|100644|0|102312577
```

Result The relevant metadata (except inode number which cannot match across filesystems), is the same. The access time seems to be overwritten, though. In the ils output there is an entry for the root directory itself, which is not there because it is not covered by the wildcard in the stat command.

4.5 Allocation Status Retrieval (A4)

The allocation status is related to the TSK tools blkstat and blkls. Every block is either allocated or unallocated.

To start with this evaluation a test image is prepared with some testfiles (pictures from Wikimedia Commons) copied to it. In the following table the testfiles with their relevant properties such as file name, file size and the position inside the test image as well as the calculated block boundaries are listed. With the help of this table it is possible to verify the output of the blkls command. The test procedure is as follows:

1. Creation of an empty image
2. Formatting the image with btrfs
3. Mounting the image
4. Copying test files to the mounted image
5. Finding the positions of the test files inside the image[1]

[1]This process could be done manually or with the help of a simple tool: `http://unix.stackexchange.com/a/39739`

6. Comparing the blkls output with the positions of the test files

Filename	Offset	Length	Block Start	Block End
4_Cilindros,_Múnich, _Alemania,_2013-02-11, _DD_07.JPG	12582912	7859002	12288	19963
Ajaccio_phare_citadelle.jpg	242483200	5412610	236800	242086
Berliner_Fernsehturm _November_2013.jpg	247898112	4046367	242088	246040
Calle_en_centro_de _Maracaibo.jpg	251944960	18063125	246040	263680
Death_Valley_exit_SR190- _view_Panamint_Butt_flash- _flood_2013.jpg	270008320	4084421	263680	267669
Dunvegan_Castle_in_the- _mist01editcrop_2007-08- 22.jpg	274096128	893341	267672	268545
Gypful.jpg	274993152	8551922	268548	276900
Joshua_Tree_Park- _approaching_thunderstorm- _02_2013.jpg	283545600	4922217	276900	281707
Spb_06-2012_Palace- _Embankment_various_01.jpg	288468992	6913192	281708	288460
Walking_caterpillar.ogv	295383040	51300889	288460	338559
Wikimania_2011_-_Opening- _ceremony_greetings_by_Meir- _Sheetrit.ogv	346685440	102312577	338560	677120

Since the blkls command outputs a single line for every block, the output is not very readable (1GiB image means 1024^2, about a million lines). To summarize the block allocation status to block ranges, a simple python script was written (see Figure 4.3). The output then becomes readable and interpretable by humans.

The blkls execution for the image is executed and shows the following results:

```
$ blkls -l -a image.img | tail -n+4 | collapse_blocks.py
12288-20028|a
20484-20488|a
28676-28680|a
36884-36892|a
36912-36916|a
36924-36928|a
36932-37048|a
37056-37204|a
37216-37220|a
37240-37256|a
```

```
#!/usr/bin/env python
import fileinput

rng = None
rng_alloc = None
for line in fileinput.input():
    b, a = line.split('|')
    block = int(b)
    alloc = a.strip()

    if rng == None:
        rng = (block, block)
        rng_alloc = alloc
    else:
        rng_start, rng_end = rng
        if rng_end == (block - 1) and alloc == rng_alloc:
            rng = (rng_start, block)
        else:
            print "%d-%d|%s" % (rng_start, rng_end, rng_alloc)
            rng = (block, block)
            rng_alloc = alloc
rng_start, rng_end = rng
print "%d-%d|%s" % (rng_start, rng_end, rng_alloc)
```

Figure 4.3: collapse_blocks.py

```
136832-136832|a
136852-136860|a
136880-136884|a
136892-136896|a
136900-137016|a
137024-137172|a
137184-137188|a
137208-137224|a
236800-438476|a
```

The block ranges 12288-20028 corresponds to the first file, the last line refers to the rest of the files. The other allocations are for trees and metadata. The comparison here is easy because Btrfs writes all files in a continuous range in the image and the sizes are easy to check.

To show that the blkls implentation is correct, we copy the image, mount it and delete the file “Death_Valley...jpg”. The blkls command must now show this area as free space.

```
$ blkls -l -a image2.img | tail -n+4 | collapse_blocks.py
```

```
12288-19964|a
20484-20488|a
28672-28672|a
28676-28680|a
36864-36876|a
36884-36892|a
36896-36900|a
36904-36912|a
36920-36932|a
36936-36964|a
36976-37048|a
37068-37204|a
37212-37216|a
37220-37232|a
136832-136844|a
136852-136860|a
136864-136868|a
136872-136880|a
136888-136900|a
136904-136932|a
136944-137016|a
137036-137172|a
137180-137184|a
137188-137200|a
236800-263680|a
267672-438540|a
```

Surprisingly Btrfs compacted the extent allocation for the first file to its real block size (the allocation was bigger before). But the last two lines show that the blocks of the deleted file are now reported as free. This confirms that the implementation is correctly reporting the allocation information of the filesystem.

4.6 Allocated Files Retrieval (A5)

The contents of the files are stored either in extents or directly in the tree. The Btrfs Sleuthkit implementation transparently enables the user to output the data of an inode to a file. This is done with the icat command. This involves reading the chunk and filesystem tree.

The test is structured as follows:

1. Create a test image (first 5 steps of the previous test)
2. Use fls to get the inodes of all files
3. Use `icat -f btrfs [image] [inode]` to output each inode to a new file

4. Compare this file to the contents of the corresponding test file

The comparison of the files is a binary comparison between the testfile and the extracted file.

Result The test runs successful which means that every file known inside the Btrfs filesystem can be read by icat and it's contents match the test file.

4.7 Content of Unallocated Files Retrieval (A6)

This evaluation is the most intensive evaluation because it is not possible to find unallocated files inside the filesystem tree, so the evaluation asks the question how long deleted files physically exist first. The next part is about evaluating if allocated and unallocated data blocks can be separated correctly. The next part is evaluating whether deleted files can be restored from the unallocated filesystem areas. The last part evaluates whether secure file deletion works due to the copy on write nature of Btrfs.

Time to Overwrite

One of the core questions for digital forensics is how long deleted files can be recovered until they are overwritten with new content. This behavior limits the effectiveness of forensic methods as they can generally just recover deleted content which is physically present on disk. Effort has been made to automatically evaluate the behavior of the Btrfs filesystem implementation to answer that question.

The test starts with the creation of a test image. After the creation of the test image, a single file is deleted. The test process then iterates over the following steps:

1. Check if the contents of the deleted file are still available. This is done by comparing the known location of the file in the image to the actual test file content in the test data directory. If it is not available, stop the loop.

2. Create a new 1MiB sized file with random content in the mounted image.

3. Increase a counter variable storing the number of iterations.

This process is expected to overwrite the data of the deleted file at some point. Since the free space of the image is a limited resource and after the creation of the image just new files are created and thus steadily decreasing the free space, the freed space of the deleted file must be reclaimed at some point.

Result The results of this test were mixed. Although with the normal process of using loop-back mounted 1GiB sparse files as storage for the image, the expected result appears: The deleted file is overwritten after about 200 new files created, the current number is varying, but not of big interest for this evaluation. This is expected behavior and proves the point that files are overwritten at some point in Btrfs. Since the exact number of data needed to be written

until data is overwritten depends on too many factors (current state of the filesystem, size of the created files, time when the files are created, Btrfs version, in harddisks: the position of the head, etc.), it does not make sense to evaluate the exact behavior. For the forensic process the important question is whether the data can be recovered or not. The answer is: yes, if it has not been overwritten in the meantime.

Repeating the same test with a real device instead of a file-based image changes the result in an unexpected way: The deleted file is never overwritten and the test aborts with a "No space left on device" exception. After joining the codepaths of the evaluation procedures for both image-types and triple-checking the implementation, the said behavior stays unexplained. The obvious difference between the two scenarios is the usage of a real block device instead of a loopback device whose implementation differs from a real block device's implementation. At some point in the Btrfs implementation the reclaiming of free space seems not to work as expected. Since the Btrfs documentation mentions that disk full conditions are a general weakness, the author expects that this effect is related to those documented issues.

Nevertheless, in real-world images, the typical writing pattern will in most cases be unequal to the test behavior and thus the trivial finding that freed data blocks are reused at some point holds true for Btrfs as for every filesystem which reclaims freed space.

Divide and Conquer

Since the contents of deleted files are defined as artifact that should be retrieved and due to the decision of not using tree forensics in this thesis, the contents of deleted files must be retrieved in some heuristical manner.

The core concept to increase the effectiveness of heuristic methods is the separation of allocated and unallocated blocks. Disk images are not generally full of data, they are allocated to some individual percentage. Applying heuristic methods to retrieve data from the whole image increases the time of the heuristic search, as more data needs to be analyzed. The Sleuthkit provides the blkls tool which can output the contents of allocated and unallocated blocks of an image. In the first step, this process needs to be verified for correct results. The image with the single deleted file from the last section is reused here because it suits the case very well.

This step can be divided into two separate evaluation steps carried out by two scripts. In the first step the output of the allocated blocks is verified by ensuring the presence of the files that are allocated and ensuring the non-presence of unallocated files (see B.18). The second step is the verification that the unallocated blocks only contain the unallocated blocks, which is being done by ensuring that the deleted file is present in the output, but none of the allocated areas.

Result The results were as expected: The two partitions of blocks had no intersection and the known contents were correctly in one of the two partitions.

Finding Unallocated File Content

Based on the previous findings from this chapters, the image can be divided into allocated and unallocated content. To find file content in unallocated areas, we start with the blkls output which only contains the unallocated data blocks. This part of the image will be searched heuristically.

This process can be carried out by a class of forensic software called file carvers. They usually use signatures to find classes of files in raw data streams.

The test image has been investigated with foremost[2], scalpel[3] and recoverjpeg[4]. Scalpel and foremost failed to find the deleted image inside the unallocated blocks because their signatures were too strict. JPEG images usually start with 0xff 0xd8 0xff 0xe0 and end with 0xff 0xd9. The deleted image starts with 0xff 0xd8 0xff 0xe1 and ends with the same binary string. But the image contained the end string inside the content, thus scalpel and foremost prematurely ended the outputted file which leads to incomplete content. Recoverjpeg is using better algorithms and instantly found the deleted image.

Result The evaluation shows that deleted file contents can be recovered based on signatures and individual algorithms. In the case of JPEG image files, the signature based carvers are oversimplifying the problem and thus lead to incomplete output. Generally these tools are out of scope of this thesis, but the evaluation shows that the deleted file artifacts can be recovered under the following premises: The files to find need to have some start and end signatures or a specialized carving algorithm for the specific file format is needed. The file needs to be written in a continuous extent, otherwise the heuristics will fail, the file format to find needs to be known. The carving process has been specified and evaluated in [31] and it was stated that the process is able to restore 87% in average of the images stored in media of different size. The problem with the premature JPEG end sequence was not mentioned.

Zeroing File Content Before Deleting

Another interesting question is the effectiveness of intentionally clearing file contents before deleting. This is an over-simplification of data wiping programs which usually overwrite files in multiple passes with random data. To evaluate this aspect, a Btrfs filesystem is mounted, a single file is overwritten with zeros using a script (see 4.4) and the contents of the file are then searched after the image was unmounted.

Result The file contents are still readable in the image. This is the case due to the copy on write nature of Btrfs. Since files are never updated in-place, the old contents are still reachable. Unless wiping tools for Btrfs are developed to work around this, wiping files in Btrfs will be difficult. This is a known limitation: “Overwriting tools rely on the following file system property: each file block is stored at known locations and when the file block is updated, then all old versions are replaced with the new version. If this assumption is not satisfied, userlevel overwriting tools silently fail.” [32] Btrfs’ COW nature contradicts this assumption.

[2] `http://foremost.sourceforge.net/`
[3] `https://github.com/sleuthkit/scalpel`
[4] `http://www.rfc1149.net/devel/recoverjpeg`

```
#!/usr/bin/env python
import os, sys

filename = sys.argv[1]
size = os.path.getsize(filename)
pos = 0
with open(filename, "rb+") as f:
    while pos < size:
        f.seek(pos)
        f.write("\0")
        pos = pos + 1
```

Figure 4.4: Script to overwrite a file with zeroes

4.8 Future Work

As stated before, this thesis didn't consider tree forensics. Since Btrfs filesystem data is stored nearly exclusively in trees, this could be a very promising way to further improve forensic methods. Tree forensics could tackle the problem that elder states of the trees could contain explicit information about deleted files and metadata. Explicitely reading this information from an elder version of a tree would eliminate the need of heuristic methods to retrieve deleted contents. Koruga and Bača analyzed NTFS Catalog B-Trees to find metadata of deleted files [25]. Also due to the nature of trees, there will most certainly be only a limited amount of unused tree information before it gets overwritten with current data. The presence of deleted tree entries could possibly be detected by analyzing the fill rate of the tree nodes [24]. Depending on the write behavior of Btrfs in a concrete usage scenario, there could be more or less of this data to retrieve. The big advantage would of course be to have explicit information about files and metadata. A first step could be the monitoring of the update behavior of tree data. The locations of old tree data could be recorded with the goal of finding patterns of where to find unallocated nodes. Also the locations of old tree roots could be monitored and thus enabling a program to reconstruct old tree versions. Depending of the result of those processes, the best strategy to gain information about elder tree states can be chosen. Possibly separation of the tree data into used and unused data with a heuristic search could lead to a list of old items which can then be searched for valid references to data extents.

Also in the area of the prototype there are a lot of possible improvements. Since the implementation of the prototype didn't focus on runtime performance, a lot of performance optimizations can be performed. The performance could be improved by using better data structures to speed up the search inside cached tree data or to optimize search routines by implementing better possibilities to express search criteria. Also it would be possible to use a visitor-like pattern for the search process if the implementation was rewritten in an object oriented programming languate. The API of The Sleuthkit could also be extended to be usable with the new Btrfs features such as subvolumes and snapshots. This would obsolete the workarounds used in the prototype

implementation; although this would be only practical if those changes would be merged into the TSK source code since maintainence would be unneccessarily hard otherwise.

Since the Btrfs format is also a moving target in the sense that the on-disk format is improved and new features are added, future work could be done in the area of multi-format capabilities. This could mean that the implementation is extended to be able to read multiple versions of Btrfs. This could include older versions as well as new versions yet to come.

The snapshots of Btrfs filesystems could also be of interest for the aggregation of historic data to detect manipulations [29]. This would be suitable for forensic analysis with the goal of detecting tampering of data. Since The Sleuthkit doesn't contain a tool to compare snapshots yet, such a tool must be written at first. Ideally such a new tool would be able to compare snapshots of ZFS filesystems too.

CHAPTER 5

Conclusion

This thesis analyzed the forensic aspects of Btrfs. In the problem definition in chapter one the general motivation and the need for forensic methods for Btrfs was justified and the artifacts to retrieve from Btrfs filesystems were defined. The second chapter presented Btrfs in a high-level way, describing the general ideas behind its design. After that, the filesystem was analyzed in the same way as Brian Carrier's reference work. The Sleuth Kit was presented for the later implementation of a protoype for Btrfs forensics. Chapter three presented the the forensic methods that were developed to retrieve the needed artifacts from Btrfs filesystems and outlined their implementation in TSK. Chapter four evaluated the implementation of the forensic methods.

The evaluation shows that most of the defined artifacts can be retrieved by the implemented prototype. Artifact A1, the filesystem metadata, is easily retrievable by reading the filesystem superblock and the root of roots tree. The output of the fsstat tool is easily extendable for more details and usage in practice will lead to improvements of the information presented by it. The filesystem structure was also verified to be exactly the same as reported by the Linux Btrfs driver. This artifact is also completely retrievable by the prototype implementation. The difference between elder generations of filesystems is that the structure information of deleted files is not retrievable without applying tree forensics. The evaluation of later artifacts shows that this is not a big restriction though. The file metadata was also found to be retrievable for allocated files in the same manner as it is for other filesystems supported by TSK. The relevant information such as timestamps could be restored and validated against the metadata reported by the stat command. The allocation status was evaluated by dividing a test image with known contents into allocated and unallocated parts. Due to the knowledge of what content to expect in each of the parts, the evaluation was simplified. The process of analyzing the output of blkls was supported by a useful script which converted the allocations from a block-based listing to a range based listing which makes it possible for humans to compare the huge listings produced by blkls. Evaluation showed that the separation into allocated and unallocated partial images works as expected. The next artifact of the allocated files was easy to check with an automated test suite as it only needs to prepare test images with known content and compare the result of the prototype output with the contents of the known test files. The evaluation of the unallocated file

contents gave interesting insights about the implication of the copy on write nature of Btrfs. One question was when deleted files are overwritten in the filesystem. The automated test showed that after a certain amount of new written content, the unallocated content was overwritten. This was expected and is similar in other filesystems. The fact that overwriting files with new content doesn't lead to the overwriting of their physical content in the storage media is also not surprising knowing that Btrfs uses the copy on write paradigm, but is surprising in the context of the majority of existing filesystems editing files in-place. The core point of this evaluation was the question whether unallocated file contents can be retrieved. Since the actual method relies on heuristics or specialized algorithms to find certain file formats, the answer is yes – but with restrictions.

The developed forensic methods are in general enabling a forensic investigator to gain information about a Btrfs filesystem and to retrieve artifacts from it. For the allocated artifacts the output is deterministic and the evaluation showed that it is equal to the output of the Linux Btrfs driver. The unallocated artifacts are relying on heuristics, but the search space can be easily minimized with the help of the reference implementation such that the allocated artifacts can be deterministically retrieved and processed and the heuristic search of the unallocated content can be speeded up by only analyzing the unallocated blocks of the filesystem.

The choice of not using tree forensics to get elder versions of the individual Btrfs trees was a simplification to limit the scope of the thesis and to keep the effort of the prototype implementation in reasonable bounds. Evaluation showed though that in practice the analysis of Btrfs is also possible without applying tree forensics, the divide and conquer strategy to look only at unallocated areas makes it quite usable.

Looking back at the state prior this thesis, forensics for Btrfs filesystems is now in a better shape than before, the prototypical implementation in The Sleuthkit makes it possible to investigate Btrfs filesystem images and to perform forensic investigations. Without this implementation the investigation would be only possible by using the Linux implementation which is not very well documented and doesn't consider forensic aspects at all. The recovery of deleted files would be limited to the application of heuristic methods which would return both allocated and uallocated files.

Bibliography

[1] Btrfs wiki - on-disk format (https://btrfs.wiki.kernel.org/index.php/on-disk_format). Accessed: 2013-03-02.

[2] Linux programmer's manual - stat(2). Version 3.44 of the man-pages project.

[3] Josef Bacik. Btrfs swiss army knife of storage, 2012.

[4] Michael Becher, Maximillian Dornseif, and Christian N Klein. Firewire all your memory are belong to us. *Proceedings of CanSecWest*, 2005.

[5] Nicole Lang Beebe, Sonia D Stacy, and Dane Stuckey. Digital forensic implications of zfs. *digital investigation*, 6:S99–S107, 2009.

[6] Robert Beverly, Simson Garfinkel, and Greg Cardwell. Forensic carving of network packets and associated data structures. *digital investigation*, 8:S78–S89, 2011.

[7] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The zettabyte file system. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, 2003.

[8] D Brezinski and Tom Killalea. Guidelines for evidence collection and archiving. *Request For Comments*, 3227, 2002.

[9] Brian Carrier. Open source digital forensics tools: The legal argument. Technical report, stake Research Report, 2002.

[10] Brian Carrier. *File system forensic analysis*. Addison-Wesley, 2005.

[11] Brian Carrier and Joe Grand. A hardware-based memory acquisition procedure for digital investigations. *Digital Investigation*, 1(1):50–60, 2004.

[12] Brian Carrier, Eugene H Spafford, et al. Getting physical with the digital investigation process. *International Journal of digital evidence*, 2(2):1–20, 2003.

[13] Eoghan Casey, Geoff Fellows, Matthew Geiger, and Gerasimos Stellatos. The growing impact of full disk encryption on digital forensics. *Digital Investigation*, 8(2):129–134, 2011.

[14] Vicka Corey, Charles Peterman, Sybil Shearin, Michael S Greenberg, and James Van Bokkelen. Network forensics analysis. *Internet Computing, IEEE*, 6(6):60–66, 2002.

[15] Knut Eckstein. Forensics for advanced unix file systems. In *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*, pages 377–385. IEEE, 2004.

[16] Kevin D Fairbanks. An analysis of ext4 for digital forensics. *Digital Investigation*, 9:S118–S130, 2012.

[17] Simson Garfinkel. Remembrance of data passed: A study of disk sanitization practices, 2003.

[18] Simson Garfinkel. Aff: A new format for storing hard drive images. *Communications of the ACM*, 49(2):85, 2006.

[19] Simson Garfinkel. Digital forensics research: The next 10 years. *Digital Investigation*, 7:S64–S73, 2010.

[20] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *digital investigation*, 6:S2–S11, 2009.

[21] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, 1996.

[22] J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[23] Karen Kent, Suzanne Chevalier, Tim Grance, and Hung Dang. Guide to integrating forensic techniques into incident response. *NIST Special Publication*, pages 800–86, 2006.

[24] Peter Kieseberg, Sebastian Schrittwieser, Martin Mulazzani, Markus Huber, and Edgar Weippl. Trees cannot lie: Using data structures for forensics purposes. In *Intelligence and Security Informatics Conference (EISIC), 2011 European*, pages 282–285. IEEE, 2011.

[25] Petra Koruga and Miroslav Bača. Analysis of b-tree data structure and its usage in computer forensics. In *Central European Conference on Information and Intelligent Systems*, 2010.

[26] Andrew Li. Zettabyte file system autopsy: Digital crime scene investigation for zettabyte file system, 2009.

[27] Lanyue Lu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST'13), San Jose, California*, 2013.

[28] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[29] Martin Mulazzani and Edgar Weippl. Aktuelle herausforderungen in der datenbankforensik. In *7th Information Security Konferenz in Krems*, 2009.

[30] Bill Nelson, Amelia Phillips, and Christopher Steuart. *Guide to computer forensics and investigations*. Cengage Learning, 2010.

[31] Digambar Povar and V.K. Bhadran. Forensic data carving. In Ibrahim Baggili, editor, *Digital Forensics and Cyber Crime*, volume 53 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 137–148. Springer Berlin Heidelberg, 2011.

[32] Joel Reardon, David Basin, and Srdjan Capkun. Sok: Secure data deletion. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 301–315. IEEE, 2013.

[33] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)*, 3(4):2, 2008.

[34] Ohad Rodeh, Josef Bacik, and Chris Mason. Brtfs: The linux b-tree filesystem. IBM Research Report RJ10501 (ALM1207-004), 2012.

[35] Nicolas Ruff. Windows memory forensics. *Journal in Computer Virology*, 4(2):83–100, 2008.

[36] Vrizlynn LL Thing, Kian-Yong Ng, and Ee-Chien Chang. Live memory forensics of mobile phones. *digital investigation*, 7:S74–S82, 2010.

[37] Theodore Y. Ts'o and Stephen Tweedie. Planned extensions to the linux ext2/ext3 filesystem. In *Proceedings of the Freenix Track: 2002 USENIX Annual Technical Conference*, pages 235–244, 2002.

[38] Timothy Vidas. Volatile memory acquisition via warm boot memory survivability. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pages 1–6. IEEE, 2010.

[39] Aaron Walters, Michael Ligh, Andrew Case, and Jamie Levy. The volatility framework: Volatile memory artifact extraction utility framework, 2013.

[40] Yinglei Wang, Wing-kei Yu, Sarah Q Xu, Edwin Kan, and G Edward Suh. Hiding information in flash memory. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 271–285. IEEE, 2013.

[41] Michael Yung Chung Wei, Laura M Grupp, Frederick E Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, volume 11, pages 8–8, 2011.

APPENDIX A

On-Disk-Format

A.1 Conventions

The following conventions structure definitions in this section contain fixed size structs (identifiable by a fixed size in the table header) and dynamic structs. Dynamic structs can be repeated (see Table A.1) and optional (see Table A.2). Repeated means that the marked part can occur multiple times in the actual structure, optional means that the structure can be present once or not at all.

`struct RepeatingStruct` ($\geq$ 8B)	
Type	Description
uint64_t	A single value.
↻▶uint64_t	A repeated value.
time◀	A repeated timestamp.

Table A.1: A repeating struct.

`struct OptionalStruct` ($\geq$ 8B)	
Type	Description
uint64_t	A single value.
↬▶uint64_t	Only there if the previous value is 42!
time◀	An optional timestamp.

Table A.2: A struct with optional data.

A.2 Data Structures

`struct Superblock` (4096B)	
Type	Description
char[32]	Checksum of the superblock.
char[16]	UUID
uint64_t	The physical address of the superblock.
uint64_t	Flags
char[8]	The Btrfs magic `_BHRfS_M`.
uint64_t	generation;
uint64_t	The logical address of the root of the root tree.
uint64_t	The logical address of the root of the chunk tree.
uint64_t	The logical address of the root of the log tree root.
uint64_t	log_root_transid;
uint64_t	The total bytes of the volume.
uint64_t	The used bytes of the volume.
uint64_t	The Object ID of the root directory (always 6).
uint64_t	The number of devices in the volume.
uint32_t	The sector size.
uint32_t	The node size.
uint32_t	The leaf size.
uint32_t	The stripe size.
uint32_t	The number of used bytes of bootstrap data (n).
uint64_t	chunk_root_generation;
uint64_t	compat_flags;
uint64_t	compat_ro_flags;
uint64_t	incompat_flags;
uint16_t	csum_type;
uint8_t	root_level;
uint8_t	chunk_root_level;
uint8_t	log_root_level;
struct `dev_item`	The dev_item of the disk.
char[256]	The volume label.
char[256]	Reserved.
char[2048]	The bootstrap data (only the first n bytes are used).
char[1237]	Unused.

Table A.3: The contents of the Btrfs superblock. [1]

`struct Key` (17B)	
Type	Description
uint64_t	Object ID.
uint8_t	Item type.
uint64_t	Offset.

Table A.4: The structure of a Btrfs key. [1] [34]

`struct Header` (101B)	
Type	Description
checksum 0x20	Checksum
uuid 0x10	Filesystem UUID
uint64_t	Logical address of this node
char[7]	Flags
uint8_t	Backref Rev.
uuid 0x10	Chunk tree UUID
uint64_t	Generation
uint64_t	tree id
uint32_t	Number of items
uint8_t	Level

Table A.5: The structure of a Btrfs key. [1]

`struct BlockPointer` (33B)	
Type	Description
struct `Key`	Key
uint64_t	The logical address of the referenced header.
uint64_t	Generation.

Table A.6: The structure of a Btrfs BlockPointer. [1]

`struct Item` (25B)	
Type	Description
struct `Key`	Key
uint32_t	The offset of the data relative to the end of the header.
uint32_t	The size of the data.

Table A.7: The structure of a Btrfs Item. [1] [34]

`struct RootItem` (239B)	
Type	Description
struct `InodeItem`	Inode Item
uint64_t	Expected Generation
uint64_t	always 100
uint64_t	Block Number of the Root Node
uint64_t	Byte limit (always 0)
uint64_t	Bytes used
uint64_t	The last generation of a snapshot
uint64_t	flags
uint32_t	nr references
struct `Key`	Drop progress (always 0:0:0)
uint8_t	drop level (always 0)
uint8_t	Level of the root of the tree.

Table A.8: The structure of a Btrfs RootItem. [1]

`struct DevItem` (98B)	
Type	Description
uint64_t	The device ID.
uint64_t	The number of bytes.
uint64_t	The number of bytes used.
uint32_t	The optimal IO alignment.
uint32_t	The optimal IO width.
uint32_t	The minimal IO size (sector size).
uint64_t	The type.
uint64_t	The generation.
uint64_t	The start offset.
uint32_t	The dev group.
uint8_t	The seek speed.
uint8_t	The bandwidth.
uuid 0x10	The devce UUID.
uuid 0x10	The filesystem UUID.

Table A.9: The structure of a Btrfs DevItem. [1]

`struct ChunkItem` (48B)	
Type	Description
uint64_t	The size of the chunk.
uint64_t	The root referencing this chunk.
uint64_t	The stripe length.
uint64_t	The type.
uint32_t	The optimal IO alignment.
uint32_t	The optimal IO width.
uint32_t	The minimal IO size (sector size).
uint16_t	The number of ChunkItemStripes.
uint16_t	The number of sub-stripes.

Table A.10: The structure of a Btrfs ChunkItem. [1]

`struct ChunkItemStripe` (32B)	
Type	Description
uint64_t	The device ID.
uint64_t	The offset.
uuid 0x10	The device UUID.

Table A.11: The structure of a Btrfs ChunkItemStripe. [1]

`struct ExtentItem` ($\geq$ 42B)	
Type	Description
uint64_t	The reference count.
uint64_t	The generation.
uint64_t	The flags 1: Data, 2: Tree Block.
struct `Key`	The Key (flags=2 only).
uint8_t	Level of the node (flags=2 only).
↻►uint8_t	Inline references: Type of the following structure.
struct `REF`◄	One of TreeBlockRef, ExtentDataRef, SharedBlockRef or SharedDataRef.

Table A.12: The structure of a Btrfs ExtentItem. [1]

`struct DevExtent` (48B)	
Type	Description
uint64_t	chunk tree id (always 3)
uint64_t	chunk oid (always 100)
uint64_t	logical address
uint64_t	size in bytes
char[16]	UUID

Table A.13: The structure of a Btrfs DevExtent. [1]

`struct InodeItem` (160B)	
Type	Description
uint64_t	The Generation
uint64_t	transid
uint64_t	st_size
uint64_t	st_blocks
uint64_t	Block Group
uint32_t	st_nlink
uint32_t	st_uid
uint32_t	st_gid
uint32_t	st_mode
uint64_t	st_rdev
uint64_t	flags
uint64_t	sequence
char[32]	reserved
time	st_atime
time	st_ctime
time	st_mtime
time	otime (reserved)

Table A.14: The structure of a Btrfs InodeItem. [1]

`struct InodeRef` ($\geq$ 10B)	
Type	Description
↻►uint64_t	Index in the directory.
uint16_t	Length of name n.
char[n]◄	ASCII name.

Table A.15: The structure of a Btrfs InodeRef. [1]

`struct DirItem` ($\geq$ 30B)	
Type	Description
↻▶struct `Key`	The key of the child.
uint64_t	Transid.
uint16_t	m
uint16_t	n
uint8_t	Type of child
char[n]	Name of child
char[m]◀	Data (empty for normal directory items)

Table A.16: The structure of a Btrfs DirItem. [1]

`struct DirIndex` ($\geq$ 30B)	
Type	Description
struct `Key`	The key of the child.
uint64_t	Transid.
uint16_t	m
uint16_t	n
uint8_t	Type of child
char[n]	Name of child
char[m]	Data (empty for normal directory items)

Table A.17: The structure of a Btrfs DirIndex. [1]

`struct ExtentData` ($\geq$ 21B)	
Type	Description
uint64_t	generation
uint64_t	n size of decoded extent
uint8_t	Type of compression.
uint8_t	Type of encryption.
uint16_t	Other encoding.
uint8_t	Type (inline/regular/prealloc).
↯▶uint64_t	The logical address of the extent.
uint64_t	The size of the extent.
uint64_t	The offset within the extent.
uint64_t◀	Logical number of bytes in the file.

Table A.18: The structure of a Btrfs ExtentData. [1]

APPENDIX B

Sourcecode

```
TSK_FS_INFO *
btrfs_tsk_open(TSK_IMG_INFO * img_info, TSK_OFF_T offset,
    TSK_FS_TYPE_ENUM ftype, uint8_t test)
{

    BTRFS_INFO *btrfs_info;
    TSK_FS_INFO *fs;

    // clean up any error messages that are lying around
    tsk_error_reset();

    // [...]

    fs = &(btrfs_info->fs_info);

    fs->ftype = ftype;
    fs->flags = 0;
    fs->img_info = img_info;
    fs->offset = offset;
    fs->tag = TSK_FS_INFO_TAG;
    fs->endian = TSK_LIT_ENDIAN;

    /*
     * Read the superblock struct.
     */
    btrfs_superblock *superblock = btrfs_io_read_superblock_pa(fs,
        BTRFS_SUPERBLOCK_LOCATION);
    btrfs_info->superblock = superblock;

    /*
     * Verify we are looking at an Btrfs image
     */
    if (strncmp(btrfs_info->superblock->magic, BTRFS_FS_MAGIC, 8) !=
        0) {
        fs->tag = 0;
        free(btrfs_info->superblock);
        free(btrfs_info);
        tsk_error_reset();
        tsk_error_set_errno(TSK_ERR_FS_MAGIC);
        tsk_error_set_errstr("not an Btrfs file system (magic)");
        if (tsk_verbose)
            fprintf(stderr, "btrfs_open: invalid magic\n");
        return NULL;
    }
```

Figure B.1: `tsk_btrfs_open` from btrfs.c

```
    // [...]

    /*
     * Find and set all the subvolumes.
     */
    btrfs_subvolume_init(btrfs_info);

    /*
     * Count the number of inodes.
     */
    fs->inum_count = btrfs_inode_count(btrfs_info);
    fs->last_inum = fs->inum_count - 1;

    /*
     * Create a mapping for inodes.
     */
    btrfs_inode_create_mapping(btrfs_info);

    // set the top level subvolume fs tree
    btrfs_info->top_level_subvolume_fs_tree =
        btrfs_subvolume_get_by_id(btrfs_info,
            BTRFS_FIRST_SUBVOLUME_ID);

    /* Set the generic function pointers */
    fs->fsstat = btrfs_tsk_fsstat;
    fs->close = btrfs_tsk_close;
    fs->inode_walk = btrfs_tsk_inode_walk;
    fs->istat = btrfs_tsk_istat;
    fs->block_walk = btrfs_tsk_block_walk;
    fs->file_add_meta = btrfs_tsk_inode_lookup;
    fs->dir_open_meta = btrfs_tsk_dir_open_meta;
    fs->load_attrs = btrfs_tsk_load_attrs;
    fs->get_default_attr_type = btrfs_tsk_get_default_attr_type;
    fs->block_getflags = btrfs_tsk_block_getflags;

    return (fs);
}
```

Figure B.2: `tsk_btrfs_open` from btrfs.c

```
btrfs_tsk_fsstat(TSK_FS_INFO * fs, FILE * hFile)
{
    BTRFS_INFO *btrfs_info = (BTRFS_INFO *) fs;
    btrfs_superblock *sb = btrfs_info->superblock;

    // clean up any error messages that are lying around
    tsk_error_reset();

    tsk_fprintf(hFile, "FILE SYSTEM INFORMATION\n");
    tsk_fprintf(hFile, "--------------------------------------------\
        n");

    tsk_fprintf(hFile, "File System Type: Btrfs\n");

    char buf[32] = "";
    btrfs_io_print_uint64_t(buf, sb->total_bytes);
    tsk_fprintf(hFile, "Total Bytes: %s\n", buf);
    btrfs_io_print_uint64_t(buf, sb->bytes_used);
    tsk_fprintf(hFile, "Bytes Used: %s\n", buf);

    // [...]

    return 0;
}
```

Figure B.3: `tsk_btrfs_fsstat` from btrfs.c

```
void
btrfs_chunk_read_superblock_bootstrap_data(BTRFS_INFO * btrfs_info,
    btrfs_superblock * super)
{
    // Initialize the TAILQ
    TAILQ_INIT(&(btrfs_info->chunks_head));

    int total_bytes = super->n;
    char n_bytes_valid[2048];
    memcpy(n_bytes_valid, super->bootstrap_chunks, 2048);
    int read = 0;

    while (read < total_bytes) {
        btrfs_key k;
        btrfs_io_parse_key(n_bytes_valid + read, &k);
        read += STRUCT_KEY_SIZE;
        read += btrfs_chunk_add(btrfs_info, k, n_bytes_valid + read);
    }
}
```

Figure B.4: `btrfs_chunk_read_superblock_bootstrap_data` from btrfs.c

```
void
btrfs_chunk_read_tree(BTRFS_INFO * btrfs_info, btrfs_tree * ct)
{
    struct btrfs_tree_list_result_head list;
    TAILQ_INIT(&list);

    btrfs_tree_list(btrfs_info, ct, &list);

    struct btrfs_tree_list_result_s *iter;
    TAILQ_FOREACH(iter, &list, pointers) {
        if (iter->key.item_type == ITEM_TYPE_CHUNK_ITEM) {
            char *d = tsk_malloc(iter->data_size);
            tsk_fs_read(&(btrfs_info->fs_info), iter->
                physical_address, d,
                iter->data_size);
            btrfs_chunk_add(btrfs_info, iter->key, d);
            free(d);
        }
    }

    btrfs_tree_list_result_free(&list);
}
```

Figure B.5: `btrfs_chunk_read_tree` from btrfs.c

```
ssize_t
btrfs_chunk_add(BTRFS_INFO * btrfs_info, btrfs_key k, char *data
{
    btrfs_chunk_item ci;
    btrfs_io_parse_chunk_item(data, &ci

    const int nr_stripes = ci.num_stripes

    btrfs_chunk_item_stripe *stripes =
        tsk_malloc(sizeof(btrfs_chunk_item_stripe) * nr_stripes);
    int i;
    for (i = 0; i < nr_stripes; i++) {
        btrfs_chunk_item_stripe cis;
        btrfs_io_parse_chunk_item_stripe(data +
            + (i * STRUCT_CHUNK_ITEM_STRIPE_SIZE), &cis);
        memcpy(&stripes[i], &cis, sizeof(btrfs_chunk_item_stripe));


    struct chunk_entry_s *ce = tsk_malloc(sizeof(struct chunk_entry_s
        ));
    memcpy(&ce->chunk_item, &ci, sizeof(btrfs_chunk_item));
    ce->chunk_item_stripes = stripes;
    memcpy(&ce->key, &k, sizeof(btrfs_key

    TAILQ_INSERT_TAIL(&(btrfs_info->chunks_head), ce, pointers);

    return STRUCT_CHUNK_ITEM_SIZE +
        (nr_stripes * STRUCT_CHUNK_ITEM_STRIPE_SIZE);
}
```

Figure B.6: `btrfs_chunk_add` from btrfs.c

```
void
btrfs_read_dev_tree(BTRFS_INFO * btrfs_info)
{
    struct btrfs_tree_list_result_head tlr;
    TAILQ_INIT(&tlr);
    btrfs_tree_list(btrfs_info, &(btrfs_info->dev_tree), &tlr);

    struct btrfs_tree_list_result_s *iter;

    struct dev_extent_entry_s *new_entry;

    // Initialize the TAILQ
    TAILQ_INIT(&(btrfs_info->dev_extents_head));

    TAILQ_FOREACH(iter, &tlr, pointers) {
        btrfs_dev_extent e = btrfs_io_read_dev_extent_pa(btrfs_info,
            iter->physical_address);

        // create the new entry
        new_entry = tsk_malloc(sizeof(struct dev_extent_entry_s));
        memcpy(&(new_entry->dev_extent), &e, sizeof(btrfs_dev_extent)
            );
        memcpy(&(new_entry->key), &(iter->key), sizeof(btrfs_key));

        // insert it into the list
        TAILQ_INSERT_TAIL(&(btrfs_info->dev_extents_head), new_entry,
            pointers);
    }

    // free the result
    btrfs_tree_list_result_free(&tlr);
}
```

Figure B.7: `btrfs_read_dev_tree` from btrfs.c

```
uint64_t
btrfs_resolve_logical_address(BTRFS_INFO * btrfs_info,
    uint64_t logical_address)
{
    const uint64_t our_dev_id = btrfs_info->superblock->dev_item.
       dev_id;

    struct chunk_entry_s *e;
    TAILQ_FOREACH(e, &(btrfs_info->chunks_head), pointers) {
        btrfs_key k = e->key;
        uint64_t key_offset = k.offset;
        uint64_t size_of_chunk = e->chunk_item.size_of_chunk;
        if ((key_offset <= logical_address)
            && (logical_address <= (key_offset + size_of_chunk))) {
            // found the address, return the address for this device
            int i;
            for (i = 0; i < e->chunk_item.num_stripes; i++) {
                btrfs_chunk_item_stripe *s =
                    (btrfs_chunk_item_stripe
                    *) (&e->chunk_item_stripes[i]);
                if (s->device_id == our_dev_id) {
                    uint64_t stripe_offset = s->offset;
                    return stripe_offset + (logical_address -
                       key_offset);
                }
            }
            return -1;
        }
    }
    return -1;
}
```

Figure B.8: `btrfs_resolve_logical_address` from btrfs.c

```
uint64_t
btrfs_resolve_physical_address(BTRFS_INFO * btrfs_info,
    uint64_t physical_address)
{
    struct dev_extent_entry_s *e;
    TAILQ_FOREACH(e, &(btrfs_info->dev_extents_head), pointers) {
        uint64_t key_offset = e->key.offset;
        uint64_t extent_size = e->dev_extent.size;
        if (physical_address >= key_offset
            && physical_address <= (key_offset + extent_size)) {
            // found the right extent
            uint64_t result = e->dev_extent.logical_address
                + (physical_address - key_offset);
            return result;
        }
    }
    return -1;
}
```

Figure B.9: `btrfs_resolve_physical_address` from btrfs.c

```
static uint8_t
btrfs_tsk_istat(TSK_FS_INFO * fs, FILE * hFile, TSK_INUM_T inum,
    TSK_DADDR_T numblock, int32_t sec_skew)
{
    BTRFS_INFO *btrfs_info = (BTRFS_INFO *) fs;

    btrfs_inode_mapping *m = btrfs_inode_resolve(btrfs_info, inum);
    btrfs_tree fstree =
        btrfs_subvolume_get_by_id(btrfs_info, m->subvolume_id);
    btrfs_inode_item_result res =
        btrfs_find_inode_item(btrfs_info, &fstree,
        m->inode_nr);

    if (res.found) {
        TSK_FS_FILE *f = tsk_malloc(sizeof(TSK_FS_FILE));
        f->fs_info = fs;
        f->meta = tsk_fs_meta_alloc(0);
        btrfs_read_metadata(btrfs_info, &(res.inode_item), NULL, inum
            ,
            f->meta);
        tsk_fprintf(hFile, "Inode␣number␣%" PRIuINUM "␣(virtual)\n",
            inum);
        tsk_fprintf(hFile,
            "Btrfs␣inode␣number␣is␣%" PRIu64 "␣on␣subvolume␣%" PRIu64
            "\n", m->inode_nr, m->subvolume_id);
        tsk_fprintf(hFile, "===␣stat␣info␣===\n");
        tsk_fprintf(hFile, "Size:\t\t\t␣%" PRIu64 "\n", f->meta->size
            );
        tsk_fprintf(hFile, "Access␣time:\t\t␣%" PRIu64 "\n",
            f->meta->atime);
        tsk_fprintf(hFile, "Modified␣time:\t\t␣%" PRIu64 "\n",
            f->meta->mtime);
        tsk_fprintf(hFile, "Create␣time:\t\t␣%" PRIu64 "\n",
            f->meta->ctime);
        return 0;
    }
    else {
        return 1;
    }
}
```

Figure B.10: `btrfs_tsk_istat` from btrfs.c

```
uint8_t
btrfs_tsk_inode_walk(TSK_FS_INFO * fs, TSK_INUM_T start_inum,
    TSK_INUM_T end_inum, TSK_FS_META_FLAG_ENUM flags,
    TSK_FS_META_WALK_CB a_action, void *a_ptr)
{
    TSK_FS_FILE *file;

    // Allocate TSK_FS_FILE and META
    if ((file = tsk_fs_file_alloc(fs)) == NULL) {
        return 1;
    }
    if ((file->meta = tsk_fs_meta_alloc(BTRFS_FILE_CONTENT_LEN)) ==
       NULL) {
        return 1;
    }

    int i;
    for (i = start_inum; i <= end_inum; i++) {
        if (btrfs_tsk_inode_lookup(fs, file, i)) {
            // ERROR
            return 1;
        }
        else {
            TSK_WALK_RET_ENUM retval = a_action(file, a_ptr);
            if (retval == TSK_WALK_STOP || retval == TSK_WALK_ERROR)
                {
                return 0;
            }
        }
    }
    return 0;
}
```

Figure B.11: `btrfs_tsk_inode_walk` from btrfs.c

```
uint8_t
btrfs_tsk_block_walk(TSK_FS_INFO * a_fs, TSK_DADDR_T a_start_blk,
    TSK_DADDR_T a_end_blk, TSK_FS_BLOCK_WALK_FLAG_ENUM a_flags,
    TSK_FS_BLOCK_WALK_CB a_action, void *a_ptr)
{
    // [...]
    if (a_start_blk < a_fs->first_block || a_start_blk > a_fs->
       last_block) {
    // [...]
    if (a_end_blk < a_fs->first_block || a_end_blk > a_fs->last_block
        || a_end_blk < a_start_blk) {
    // [...]

    // Iterate over the blocks.
    uint8_t callback;
    for (addr = a_start_blk; addr <= a_end_blk; addr++) {
        callback = 0;

        TSK_FS_BLOCK_FLAG_ENUM flags =
            btrfs_tsk_block_getflags(&(btrfs_info->fs_info), addr);
        // [...]
        if (tsk_fs_block_get_flag(a_fs, fs_block, addr, flags) ==
           NULL) {
            tsk_error_set_errstr2("btrfs_block_walk: block %"
               PRIuDADDR,
                addr);
            tsk_fs_block_free(fs_block);
            return 1;
        }

        if (callback == 1) {
            int retval = a_action(fs_block, a_ptr);
            if (retval == TSK_WALK_STOP) {
                break;
            }
            else if (retval == TSK_WALK_ERROR) {
                tsk_fs_block_free(fs_block);
                return 1;
            }
        }

    }
    return 0;
}
```

Figure B.12: `btrfs_tsk_block_walk` from btrfs.c

```
btrfs_tree_search_result
btrfs_tree_search(BTRFS_INFO * btrfs_info,
    btrfs_key * key, btrfs_tree * tree, int verbose, compare_func cmp
        )
{
    // [...]
    if (h->level == 0) {
        // leaf node
        // [...]
        int i;
        for (i = 0; i < h->number_items; i++) {
            // [...]
            btrfs_key current_key = btrfs_io_read_key_pa(btrfs_info,
                physical_end_of_header + (i * STRUCT_ITEM_SIZE));
            int ret = cmp(&current_key, key);
            if (ret == 0) {
                btrfs_item it = btrfs_io_read_item_pa(btrfs_info,
                    physical_end_of_header + (i * STRUCT_ITEM_SIZE));
                memcpy((&r.key), (&it.key), sizeof(btrfs_key));
                char *d = tsk_malloc(it.data_size);
                tsk_fs_read(fs, physical_end_of_header + it.
                    data_offset, d,
                    it.data_size);
                r.data = d;
                r.data_size = it.data_size;
                r.physical_address =
                    physical_end_of_header + it.data_offset;
                r.found = 1;
                return r;
            }
        }
    }
    else {
        // inner node
        // [...]
        }
    }
    return r;
}
```

Figure B.13: `btrfs_tree_search` from btrfs.c

```
int
btrfs_tree_list_filter(BTRFS_INFO * btrfs_info, btrfs_tree * tree,
    struct btrfs_tree_list_result_head *list_head, compare_func cmp,
    btrfs_key * k)
{
    // [...]
    if (h->level == 0) {
        // leaf node
        int i;
        for (i = 0; i < h->number_items; i++) {
            btrfs_item it = btrfs_io_read_item_pa(btrfs_info,
                physical_end_of_header + (i * STRUCT_ITEM_SIZE));
            memcpy((&r.key), (&it.key), sizeof(btrfs_key));
            char *d = tsk_malloc(it.data_size);
            tsk_fs_read(fs, physical_end_of_header + it.data_offset,
                d,
                it.data_size);
            r.data = d;
            r.data_size = it.data_size;
            r.physical_address = physical_end_of_header + it.
                data_offset;
            r.found = 1;
            num_results++;
            if (cmp(k, &(r.key)) == 0) {
                btrfs_tree_result_func_list(&r, list_head);
            }
            else {
                free(d);
            }
        }
    }
    else {
        // inner node
        // [...]
    }
}
```

Figure B.14: `btrfs_tree_list_filter` from btrfs.c

```
int
btrfs_cmp_func_exact(btrfs_key * k1, btrfs_key * k2)
{
    if (k1->object_id > k2->object_id) {
        return 1;
    }
    else if (k1->object_id < k2->object_id) {
        return -1;
    }

    if (k1->item_type > k2->item_type) {
        return 1;
    }
    else if (k1->item_type < k2->item_type) {
        return -1;
    }

    if (k1->offset > k2->offset) {
        return 1;
    }
    else if (k1->offset < k2->offset) {
        return -1;
    }

    // they are equal
    return 0;
}
```

Figure B.15: `btrfs_cmp_func_exact` form btrfs.c

```
ssize_t
btrfs_io_parse_header(char *data, btrfs_header * header)
{
    ssize_t off = 0;
    off = btrfs_io_read_field(data, (&header->checksum), off, 0x20);
    off = btrfs_io_read_field(data, (&header->uuid), off, 0x10);
    off = btrfs_io_read_field(data, (&header->logical_address), off,
       0x8);
    off = btrfs_io_read_field(data, (&header->flags), off, 0x7);
    off = btrfs_io_read_field(data, (&header->backref), off, 0x1);
    off = btrfs_io_read_field(data, (&header->chunk_tree_uuid), off,
       0x10);
    off = btrfs_io_read_field(data, (&header->generation), off, 0x8);
    off = btrfs_io_read_field(data, (&header->tree_id), off, 0x8);
    off = btrfs_io_read_field(data, (&header->number_items), off, 0x4
       );
    off = btrfs_io_read_field(data, (&header->level), off, 0x1);
    assert(off == STRUCT_HEADER_SIZE);
    return off;
}

ssize_t
btrfs_io_read_field(char *data, void *out, ssize_t offset,
    ssize_t field_size)
{
    memcpy(out, (data + offset), field_size);
    return offset + field_size;
}
```

Figure B.16: `btrfs_io_parse_header` and `btrfs_io_read_field` from btrfs_io.c

```
btrfs_block_ptr
btrfs_io_read_block_ptr_la(BTRFS_INFO * btrfs_info,
    uint64_t logical_address)
{
    uint64_t physical_address = btrfs_resolve_logical_address(
        btrfs_info,
        logical_address);

    return btrfs_io_read_block_ptr_pa(btrfs_info, physical_address);
}

btrfs_block_ptr
btrfs_io_read_block_ptr_pa(BTRFS_INFO * btrfs_info,
    uint64_t physical_address)
{
    // Read the data from disk
    const ssize_t s = STRUCT_BLOCK_PTR_SIZE;
    char data[s];
    tsk_fs_read(&(btrfs_info->fs_info), physical_address, data, s);

    // Parse it
    btrfs_block_ptr kp;
    btrfs_io_parse_block_ptr(data, &kp);

    return kp;
}
```

Figure B.17: `btrfs_io_read_block_ptr_la` and `btrfs_io_read_block_ptr_pa` from btrfs_io.c

```
#!/bin/bash
BLKLS=../public/sleuthkit/tools/fstools/blkls
IMG=istat-test-deleted1file
ALLOC=istat-test-deleted1file.allocated
IMGDIR=../public/sleuthkit-btrfs-testsuite/src/test/
    resources/testfiles
PRESENT=(
    "4_Cilindros,_Múnich,_Alemania,_2013-02-11,_DD_07.JPG"
    "Ajaccio_phare_citadelle.jpg"
)
ABSENT=("Death_Valley_exit_SR190_view_Panamint_Butt_flash_
    flood_2013.jpg")

# Run blkls
echo "Running blkls..."
$BLKLS -a $IMG > $ALLOC

# Check that the present files are there
echo "Verifying present files..."
for i in "${PRESENT[@]}"
do
    IMGPATH="$IMGDIR/$i"
    ADDR=$(./find $IMGPATH $ALLOC)
    if [ $? -eq 0 ]; then
        echo "OK, $ADDR, $i"
    else
        echo "NOK, $i"
    fi
done

# Check that the absent files are not there
echo "Verifying absent files..."
for i in "${ABSENT[@]}"
do
    IMGPATH="$IMGDIR/$i"
    ADDR=$(./find $IMGPATH $ALLOC)
    if [ $? -eq 1 ]; then
        echo "OK, $i"
    else
        echo "NOK, $i"
    fi
done

echo "Finished."
```


Figure B.18: Script to verify the results of blkls -a