DFRWS
DIGITAL FORENSIC RESEARCH CONFERENCE

Forensic Analysis of Multiple Device BTRFS Configurations Using the Sleuth Kit

By

Jan-Niclas Hilgert, Martin Lambertz, Shujian Yang

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2018 USA

Providence, RI (July 15th - 18th)

Digital Investigation 26 (2018) S21–S29

Contents lists available at ScienceDirect

Digital Investigation

journal homepage: www.elsevier.com/locate/diin

DFRWS 2018 USA — Proceedings of the Eighteenth Annual DFRWS USA

Forensic analysis of multiple device BTRFS configurations using The Sleuth Kit

Jan-Niclas Hilgert [a, *], Martin Lambertz [a], Shujian Yang [b]

[a] Fraunhofer FKIE, Bonn, Germany
[b] Cap Barbell, Houston, TX, USA

Keywords:
File systems
Pooled storage
Forensic analysis
BTRFS
The Sleuth Kit

A B S T R A C T

The analysis of file systems is a fundamental step in every forensic investigation. Long-known file systems such as FAT, NTFS, or the ext family are well supported by commercial and open source forensics tools. When it comes to more recent file systems with technologically advanced features, however, most tools fall short of being able to provide an investigator with means to perform a proper forensic analysis.

BTRFS is such a file system which has not received the attention it should have. Although introduced in 2007, marked as stable in 2014, and being the default file system in certain Linux distributions, there is virtually no research available in the area of digital forensics when it comes to BTRFS; nor are there any software tools capable of analyzing a BTRFS file system in a way required for a forensic analysis.

In this paper we add support for BTRFS—including support for multiple device configurations—to The Sleuth Kit, a widely used toolkit when it comes to open source file system forensics. Moreover, we provide an analysis of forensically important features of BTRFS and show how our implementation can be used to utilize these during a forensic analysis.

1. Introduction

In 2005, Brian Carrier published his book “File System Forensic Analysis” Carrier (2005), in which he analyzed and explained storage devices and file systems in an unprecedented depth. Furthermore, he proposed a model how to analyze storage media from the physical media up to the analysis of extracted files. His work quickly became the foundation for any analysis conducted in this area. Moreover, he provided an implementation for his theoretical model, known as The Sleuth Kit (TSK) Carrier (2017). TSK is a forensic toolkit, providing multiple commands, which enables an investigator to perform a forensic analysis of file systems, independent of the actual file system at hand. Thus, no extensive background knowledge about the internal structures of a file system is required in order to create a file listing, recover deleted files, or search for unallocated sections. Along with the fact that it is open source and can be used or extended by anyone, TSK became a commonly used tool for many analysts and researchers next to commercial products.

TSK provides support for a variety of file systems including ext4 on Linux, Microsoft's NTFS and FAT, and Apple's HFS+. Although these file systems are still widely used on today's computers, other file systems have been introduced since the publication of Carrier's book and TSK. While FAT for instance is still often used on thumb drives or memory cards due to its simplicity, the demand for reliability, security, and maintainability has sparked progress in the world of file systems. The copy-on-write principle is used to keep file systems in a stable state, even after a crash has caused a write operation to fail. Encryption on a file system-level increases the protection of personal data in such a way that it is available out of the box and transparent to the user. Furthermore, modern file systems decrease the overhead for administrative tasks like volume management or partitioning. By implementing multiple device support like ZFS or BTRFS, volumes can be added or removed straightforwardly to existing file systems. Additionally, snapshots are used to effortlessly create complete backups of a file system.

In this paper, we implement one of these modern file systems into TSK in order to close the gap between them and the forensic world. For this purpose, we are taking an in-depth look at BTRFS as one of the most prominent examples in this area. BTRFS supports multiple of the aforementioned features, including copy-on-write, snapshots, and multiple device support. Despite the fact that it

* Corresponding author.
E-mail addresses: jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), martin.lambertz@fkie.fraunhofer.de (M. Lambertz), yang_shujian@hotmail.com (S. Yang).

https://doi.org/10.1016/j.diin.2018.04.020

was implemented into the Linux Kernel more than eight years ago, it has not received the adequate amount of attention in the academic or practical forensic area. Therefore, we also provide the first multiple device analysis of BTRFS form a forensic point of view.

2. Related work

In this section we present related work for two main aspects: forensic analyses of BTRFS and extensions of TSK with a focus on modern file systems with multiple device support.

2.1. BTRFS forensics

As already mentioned in the Introduction, there is virtually no academic work dealing with BTRFS in the context of digital forensics. While there are a few papers introducing BTRFS and some of its structures Bacik (2012); Rodeh et al. (2013), to the best of our knowledge there is no prior work investigating which structures are of particular relevance to perform a forensic analysis of BTRFS.

Looking at the non-academic world, the situation is similar. At the time of this writing the well known forensic suites like EnCase Forensic, FTK, or X-Ways Forensics do not list BTRFS in their lists of supported file systems. X-Ways only mentions the "*ability to identify BTRFS file systems*" in their changelog of X-Ways Forensics Fleischmann and Stefan, 2012. Although there is an open pull request for BTRFS support for TSK on GitHub Pöschel and Stefan, 2015, the code changes have not been merged since 2015. Moreover, the code is not able to handle multiple device configurations which mirror or stripe data to their devices making it applicable to a small fraction of BTRFS configurations only. What is more, during our experiments the implementation failed for large test pools ($\approx$ 1 TB of size).

2.2. Multiple device file systems in The Sleuth Kit

In their work "Extending The Sleuth Kit and its Underlying Model for Pooled Storage File System Forensic Analysis" Hilgert et al. (2017), Hilgert et al. use the term "pooled storage file systems" to refer to modern multiple device file systems like ZFS and BTRFS. These file systems are characterized by the fact that all available space is combined to a pool and then shared between the file systems created on this pool. Thus, none of the file systems needs to be assigned a fixed size as they can grow and shrink dynamically. In the same transparent way, storage can be added and removed to the storage pool. These advantages of pooled storage file systems are possible, since they are providing their own type of volume management functionality keeping track of the pool members and the mapping between the logical file system addresses and the actual physical offsets on the members.

In the same paper, Hilgert et al. assess the applicability of the model behind TSK for such modern pooled storage file systems. They found that the steps of the original model are still required, but that the class of pooled storage file systems needs an additional step to be performed between the volume analysis and the file system analysis. The authors call this step "pool analysis" and Fig. 1 depicts where it has been added to in the original model.

Furthermore, they define five key aspects this step has to implement. An obvious aspect is the capability to detect pooled storage file systems. Since pooled file systems play their strength when on multiple disks, support for such multiple device configurations is also an important requirement for this step. Hilgert et al. state that it should be possible to determine the pool membership of disks and afterwards analyze the resulting storage pools, which are potentially comprised of more than one disk. Furthermore, the authors highlight that a forensic analysis should not rely on an assembled pool, but that a forensic tool should be able to parse all of the important data structures on its own in order to allow for the adequate level of detail for a forensic analysis. Finally, the authors demand that the pool analysis should be able to deal with missing pool members where possible. That is, it should be possible to perform a forensic analysis of a RAID or mirror pool if there are still enough pool members present for example.

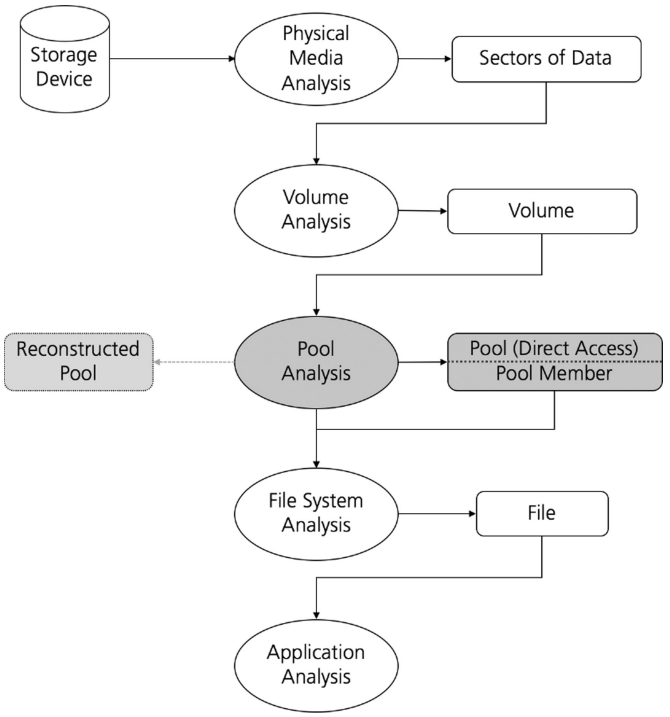


Fig. 1. Extended model for a file system forensic analysis of pooled storage file systems Hilgert et al. (2017).

As a proof of concept Hilgert et al. implemented support for ZFS into TSK to show that their extended model enables a forensic analysis of modern pooled storage file systems. However, even though the authors mention BTRFS as a pooled storage file system, they do not provide a detailed investigation of this particular file system. Neither do they prove that BTRFS is in fact covered by their model.

3. BTRFS fundamentals

BTRFS is a modern copy-on-write file system primarily for the Linux operating system. It supports advanced features like checksums, deduplication, and SSD awareness btrfs Wiki (2018a). Moreover, BTRFS allows the creation of subvolumes which can be considered as "*independently mountable POSIX filetree[s]*" btrfs Wiki (2017e). These subvolumes can be used to divide the complete file system into smaller units. Typically, such a unit contains areas of the file system which are cohesive in some way. The subvolumes of a BTRFS file system can be mounted independently of each other and with different mount options.

Furthermore, BTRFS supports snapshots, which utilize the copy-on-write principle to save and restore (parts of) a file system. Snapshots are created per subvolume and technically a snapshot is a subvolume itself. A snapshot of a subvolume represents the state of the original subvolume at the time the snapshot was created. Since snapshots are subvolumes, they can be mounted and modified. This concept gives users a comfortable option to create backups of their data without any additional soft- or hardware. This

renders snapshots a highly interesting feature when it comes to the forensic analysis of a BTRFS file system.

As already mentioned before, BTRFS is a file system with built in multiple device support. That is, it has its own volume manager implemented responsible for storing data on and reading it back from the underlying volumes of a file system. BTRFS supports different configurations in such a multiple device setup. At the time of this writing the BTRFS status page btrfs Wiki (2017d) lists RAID0, RAID1, and RAID10 as stable implementations and RAID5 and RAID6 as flawed implementations. In line with what Hilgert et al. did in their paper Hilgert et al. (2017) for ZFS, we will use the terms pool and BTRFS file system interchangeably to refer to the complete file system including subvolumes from now on; even though the term pool is not part of the BTRFS terminology.

3.1. General overview

Similar to the ext file systems, BTRFS starts with a superblock, which stores the most basic metadata about the file system. Apart from that, the rest of the data is stored in different B-trees. The addresses of the roots of these trees can be found in the root tree. The address of the root tree in turn is stored in the superblock.

A main characteristic of a B-tree is that all information is stored in its leaf nodes. The non-leaf nodes, known as internal nodes in BTRFS, are only used as references to leaf nodes. Due to this, the internal nodes of different tree types are very similar as they only contain pointers to other nodes. The leaf nodes on the other hand have different types of records called items. Their exact structure and content depends on the type of the tree at hand. Listed below is an overview of the most important types of trees in BTRFS:

- **Chunk tree:** The chunk tree is used to perform the mapping from logical to physical addresses in BTRFS. All addresses used in BTRFS are logical addresses, which translate to one or more physical addresses depending on the pool configuration. Since also the chunk tree is referenced by its logical address, the superblock contains a part of the chunk tree, the system chunk items, for the initial mapping. This is required to build the chunk tree in the first place. A detailed description of the mapping performed by the chunk tree in BTRFS is given in Section 3.2. Besides, the chunk tree also contains information about the devices used in the pool.
- **Root tree:** The root tree stores the addresses of the roots of the trees used by BTRFS. This includes the extent tree, checksum tree, and device tree as well as all available file system trees. The root address of the chunk tree on the other hand is not stored in the root tree, but in the superblock.
- **File system tree:** This type of tree stores information about the file and directory hierarchy in file systems, subvolumes, and snapshots. This includes the metadata of files and directories as well as extent data items referencing the actual data.
- **Extent tree:** Allocation records can be found in the extent tree. This includes block group items, defining regions in the logical address space of BTRFS as well as metadata and extent items allocating space within these regions. The number of references to these items as well as a back reference for each reference is also stored.
- **Checksum tree:** This tree simply contains checksums for the data stored in the BTRFS file system.
- **Device tree:** The device tree is used for the reversed address mapping, from physical to logical addresses. This becomes necessary, when physical devices are for instance removed from the pool.

The general approach to perform a BTRFS file walk from the superblock to the contents of a file is depicted in Fig. 2 and includes the following main steps:

1. Locate the superblock at the default physical address 0x10000.
2. Extract the system chunk items stored in the superblock for the initial logical to physical address mapping.
3. Find the logical address of the chunk tree in the superblock, translate it to its physical counterpart, and build the chunk tree. From now on, this tree will be used to perform the mapping from logical to physical addresses.
4. Find the logical address of the root tree in the superblock, translate it to its physical address and build the root tree.
5. The root tree stores the logical addresses of the roots of the other trees including the file system trees. Find the address of the corresponding root of the file system tree, translate it and build the tree.
6. Traverse the file system tree to find the file of interest. Its name is stored in a directory item.
7. Read the corresponding inode item of the file in the file system tree, referenced by the directory item, to retrieve its ID and metadata.
8. Use the ID as a key to find its extent data items in the file system tree.
9. Extract the data described by all extents corresponding to the file by mapping their logical to physical addresses.

In summary, the analysis of BTRFS starts with reading the superblock and extracting the roots of the trees. Once the tree roots are available, the rest of the analysis is all about expanding, referencing, and reading the child nodes of these tree roots. More detailed information about the on-disk format and data structures of BTRFS can be found in the official Wiki btrfs Wiki (2018b, 2017a).

3.2. Multiple device support

An integral feature of BTRFS is the support for multiple devices, whose available space is combined and shared by the subvolumes. In order to accomplish this, BTRFS adds another layer of abstraction between the logical addresses used by the file system and the corresponding physical addresses referring to the actual devices. This abstraction is implemented by a mapping, which translates a logical address to the correct combination of physical device and corresponding physical offset. Depending on the configuration, a logical address can also map to multiple physical offsets and devices in order to increase the redundancy of the data.

For keeping track of its devices and performing the logical-to-physical mapping, BTRFS uses special structures stored in the chunk tree. For each device, a device item is added to the chunk tree, containing information such as a unique identifier for the device, another device identifier used to index the available devices, and its total available space. In addition to device items, the chunk tree contains multiple chunk items defining logical chunks. In BTRFS, the complete logical address space is split into these non-overlapping logical chunks. Thus, one logical address can be uniquely associated with one logical chunk. These logical chunks also correspond to the regions defined by the block group items found in the extent tree. Each chunk item contains the logical start address of the chunk it describes as well as its length, the type of data it stores, and the RAID configuration used to store it. Different types of chunk items are used to map different types of data btrfs Wiki (2017b):

- **System:** System chunk items are used for the translation of logical addresses of the chunk tree itself. For this reason, all

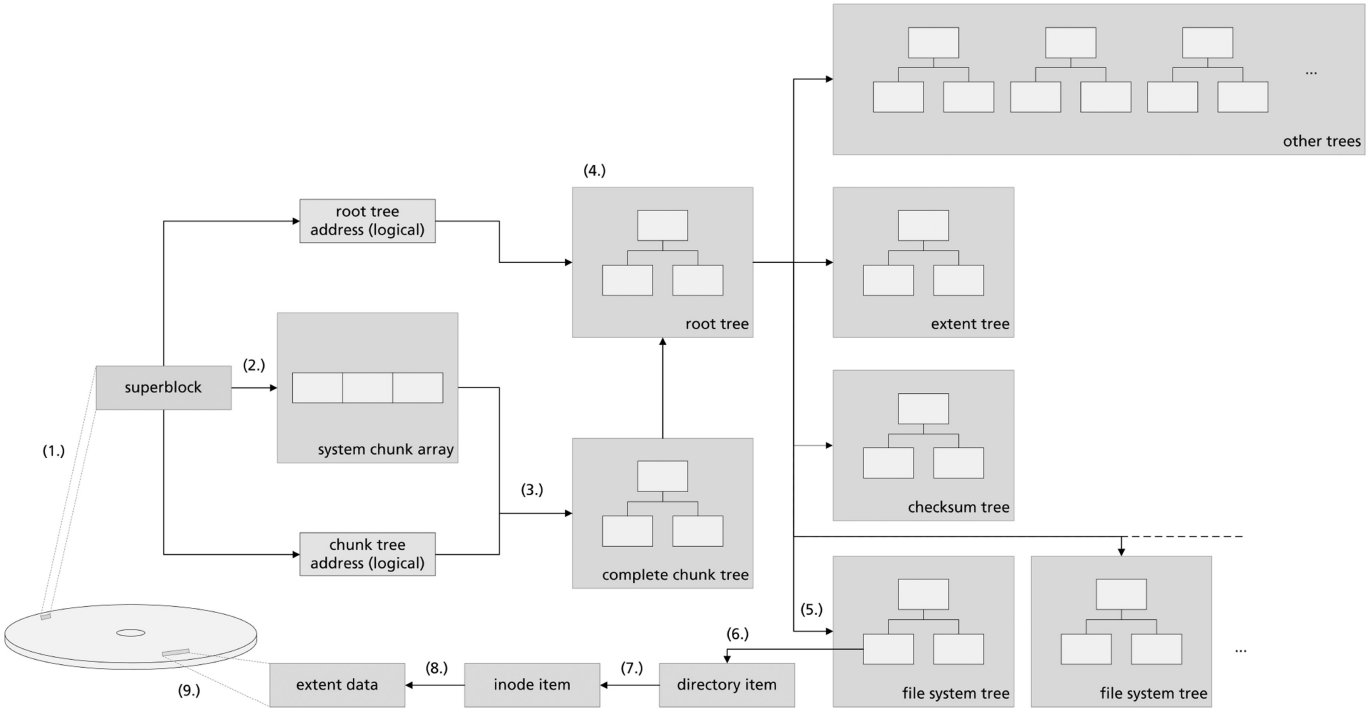


Fig. 2. Overview of the most important BTRFS structures used for a file walk.

available system chunk items are also already stored in the superblock as described previously.

- **Metadata:** Metadata chunk items are used for the translation of logical addresses of file system internal data structures like root items, inode items or directory items. Thus, tree structures like the root tree, extent tree, device tree, and file system trees are built using this type of chunk items. In BTRFS, small amounts of data can be stored inside of metadata structures, for example in extent data items. In this case, this chunk type is implicitly used to map the addresses of the embedded raw data.
- **Data:** These chunk items are only used for the translation of logical addresses of data blocks.

Each chunk is further divided into a number of stripes defined in the chunk. The device corresponding to a stripe can be identified by the given device identifier. The physical offset of each stripe indicates the beginning of the data on a device. Each stripe in a chunk item is in turn divided into equally sized units with a stripe length defined in the chunk item. In addition to the type of data stored within the chunk, its type also defines the RAID configuration used to store data.

In RAID0, all data is striped across the available stripes of the logical chunk. After a unit in a stripe is filled, the data is written to the next stripe. This configuration leads to data loss, if one of the stripes fails. RAID1 mirrors the data to all stripes in the chunk resulting in redundancy. That is, the units of each stripe are the same. As far as we know, RAID1 always uses a pair of all available devices as its stripes for each chunk item, while RAID0 always uses all of the available devices. The exact number of stripes used by each chunk item is always specified in the chunk item itself. RAID10 combines the aforementioned concepts in such a way, that all of the available stripes in a chunk are split into RAID1 configurations across which the data is then striped. Each of these RAID1 configurations in turn mirrors the data across all of their corresponding stripes. The exact number of stripes used per RAID1 configuration is defined in the chunk and referred to as sub stripes.

Listing 1. Chunk item example.

```
$ btrfs-debug-tree/dev/sda
[...]
item 7 key (FIRST_CHUNK_TREE CHUNK_ITEM 299892736)
itemoff 15265 Itemsize 176
chunk length 262144000 owner 2 stripe_len 65536
type DATA|RAID10 num_stripes 4 sub_stripes 2
stripe 0 devid 2 offset 9437184
dev uuid: 66aaeb1a-8cbb-4979-89cf-56fb0c6c958a
stripe 1 devid 1 offset 152043520
dev uuid: b3b74185-13b0-4d2a-8300-ca740c384f4b
stripe 2 devid 5 offset 140509184
dev uuid: c7099e88-5597-4776-9ee0-3d6b662e53b3
stripe 3 devid 4 offset 140509184
dev uuid: e84da2d2-d5fe-4226-a8aa-52d1ad8988b5
[...]
```

As an example, the chunk item depicted in Listing 1 defines the chunk starting at the logical address 299892736 spanning to address 562036736. It is used to store data using a RAID10 configuration with four stripes and two sub stripes. As described earlier, this means that these stripes are split into two RAID1 configurations, each consisting of two stripes. In this case, stripe 0 and 1 as well as stripe 2 and 3 are used as a RAID1 configuration and store the same data. For each stripe, the corresponding device identifier is given indicating the physical device on which the data of the stripe is stored. In this example, the stripes of the chunk are located on devices 1, 2, 4 and 5. The exact location of the data on each stripe (and therefore on the devices) can be determined using the given offset for each stripe.

4. Integrating BTRFS into TSK

In order to integrate BTRFS into TSK, it is indispensable to evaluate the applicability of its underlying model to multiple device file systems which BTRFS is an instance of. Hilgert et al. already

discussed this and presented an extended model for TSK, which enables a forensic analysis of multiple device file systems. For this reason, we will first assess the applicability of the revised model for BTRFS followed by a detailed overview of our implementation.

4.1. Theoretical model

Hilgert et al. adopted the first and last step, the physical media analysis and the application analysis, as they stood because they do not need to be changed in order to be applied to pooled storage file systems. The first step only processes the available data on the devices—the pool members in our case—as a sequence of bytes and does not interpret the data at all. The last step on the other hand, interprets the extracted data as files. This does not require any file system specific information, because at that time of the analysis, the files have already been extracted from the file system. Since these two steps are independent of the file system, they can also be applied unchanged to BTRFS.

The original model was extended by adding a pool analysis step. Hilgert et al. added this step to address the integrated volume management capability pooled storage file systems are equipped with. The potentially multiple devices spanned by a BTRFS pool, however, are not necessarily raw hard disks. Instead, they can also be partitions, RAIDs or other multiple disk volumes. Therefore, also for BTRFS it is still required to perform a volume analysis in order to detect the volumes involved.

Furthermore, mounting a BTRFS file system also results in access to the data (i.e. files and directories) stored on the most recent version of the file system. Apart from that, no access to file system internal data structures is possible. Accessing older versions of files as well as file system data and metadata directly requires direct access to the BTRFS pool, which is obtained during the pool analysis step. Taken all together, BTRFS fits into the model presented by Hilgert et al. without any needs for further modification.

As Hilgert et al. pointed out, the pool analysis is a highly file system dependent step, which needs to be implemented for each new file system. This is similar to the file system analysis functionality in TSK that differs from file system to file system. The next section describes in detail how the pool analysis for BTRFS is implemented.

4.2. Pool analysis

The tasks of the pool analysis can be divided into two major steps. First, the given volumes need to be searched for a pooled storage file system. Furthermore, the corresponding pool and its members need to be identified. Second, after the members and file system type are known, the mapping from logical to physical addresses needs to be performed. This results in direct access to the pool, which is required to perform a complete file system analysis of BTRFS.

4.2.1. Pool membership detection

As an input, the pool analysis receives the volumes found during the volume analysis and detects the underlying pooled storage file system, if there is any. Each device in BTRFS stores a superblock at the physical offset 0x10000, containing the most essential file system information. It does not only identify the volume as part of a BTRFS pool, but it also contains the file system UUID. This ID is global for the whole BTRFS pool and can be used to identify other members of the multiple device configuration. Unlike ZFS which requires a name for its pools, BTRFS does not demand a label to be set for a file system or a pool. The superblock also includes a device item for the current device containing its unique identifier enabling us to rule out duplicate volumes.

Another essential part of this step is the detection of missing devices. Although the superblock contains the total number of devices used in a BTRFS pool, it provides information only about the device it is stored on and not about any of the other devices of the pool. Some information can be obtained by looking at the system chunk items stored in each superblock. These chunks contain the IDs and the UUIDs of the devices used for its stripes. However, in configurations like RAID1 or RAID10, not all available devices may be used for the available system chunks. In that case, this method will not provide a complete listing of all devices. Another possibility to obtain more information about the available devices opens up, when all devices storing the chunk tree are available. In this case, the complete chunk tree can be built containing device items for all devices used in the BTRFS pool.

4.2.2. Mapping of logical to physical addresses

After the available volumes of a pool have been detected, we need to gain direct access to data at the correct offsets stored on the pool members. For this, we need to be able to perform the mapping from logical to physical addresses. In BTRFS, this mapping is done by utilizing the chunk tree as described in Section 3.2. Fig. 3 illustrates the following steps describing how to map a logical address to a physical address (i.e. the physical offset on the disk) for a RAID0 configuration:

1. Locate the **chunk item** containing the given logical target address (t_{log}) in the chunk tree. This gives us the logical start address of the chunk (c_{log}).
2. Calculate the **difference** (Δ) between the logical target address and the logical start address of the chunk.

$$\Delta = t_{log} - c_{log}$$

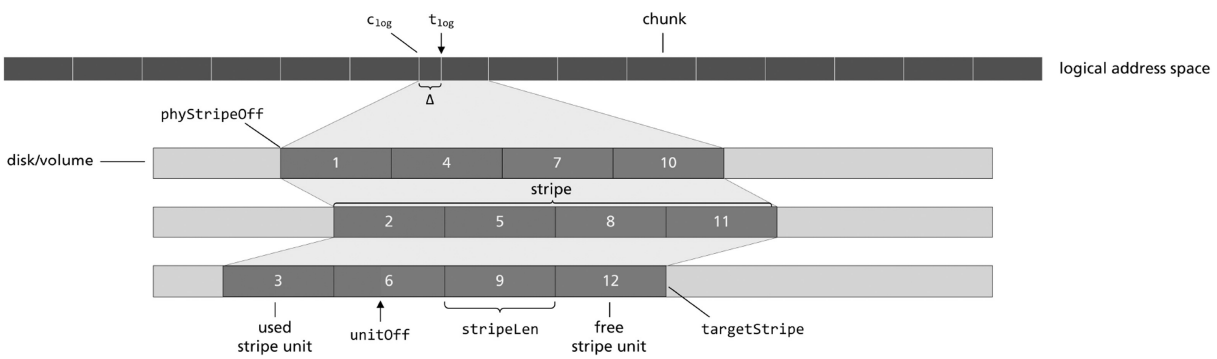


Fig. 3. Distribution of data in a RAID0 chunk item using three stripes.

This difference represents the offset of the target address within the chunk item.

3. Use Δ and the stripe length (`stripeLen`) to compute the total number of stripe units preceding our target address (`preStripeUnits`):

$$\texttt{preStripeUnits} = \left\lfloor \frac{\Delta}{\texttt{stripeLen}} \right\rfloor$$

4. Find out on **which stripe** (`targetStripe`) our logical address (and thus the start of the data) lies by calculating the total number of preceding units modulus the number of stripes (`nStripes`).

$$\texttt{targetStripe} = \texttt{preStripeUnits} \bmod \texttt{nStripes}$$

5. Knowing the corresponding stripe gives us the **physical start offset** (`phyStripeOff`) of the data on the device specified in the chunk item.
6. Calculate the **number of units** (`nStripeUnits`) that have already been allocated on our stripe by dividing the total number of units already filled by the number of available stripes.

$$\texttt{nStripeUnits} = \left\lfloor \frac{\texttt{preStripeUnits}}{\texttt{nStripes}} \right\rfloor$$

7. Calculate the offset within the unit (`unitOff`) on our stripe.

$$\texttt{unitOff} = \Delta \bmod \texttt{stripeLen}$$

8. Adding the calculated values results in the **final physical offset** (`phyOff`)

$$\begin{aligned}\texttt{phyOff} = {} & \texttt{phyStripeOff} \\ & + \texttt{nStripeUnits} \cdot \texttt{stripeLen} \\ & + \texttt{unitOff}\end{aligned}$$

For a single disk configuration, the logical address space described by the chunk starts at the physical offset of the one and only stripe and continues without any interruption. For this reason, the physical offset can simply be calculated by:

$$\texttt{phyOff} = \texttt{phyStripeOff} + \Delta$$

Since each stripe in a RAID1 configuration stores the same data, it is possible to choose any stripe of the chunk and calculate the physical offset in a similar way to a single disk configuration. For RAID10, it is necessary to choose one stripe out of each used RAID1 configuration. Afterwards, these stripes are nothing but a RAID0 configuration, whose mapping can be calculated following the aforementioned steps 1 to 8.

BTRFS also supports RAID5 and RAID6, however, due to bugs in the implementations and the consequent risk of data loss, it is officially recommended not to use these configurations btrfs Wiki (2017c). Therefore, we do not cover RAID5 and RAID6 in our implementation for now.

5. Forensic artifacts in BTRFS

The following sections are used to highlight features of BTRFS which are of particular interest for a forensic examiner when presented with a BTRFS file system. We extended the implementation by Hilgert et al., to enable a forensic analysis of BTRFS Hilgert et al. (2018). In the same way as their support for ZFS, our implementation does not alter the functionality of the original Sleuth Kit, so that it can still deal with any previously supported file systems.

5.1. Forensic analysis of a BTRFS pool

As already described in Section 4.2.1, a main aspect during a forensic investigation of a pooled storage file system is the detection of its members followed by the detection of the pool configuration. For this purpose, we extended the `pls` command introduced by Hilgert et al. to enable support for BTRFS. This command is used to perform and display the results of the pool analysis. As shown in Listing 2, the output gives an investigator insight into the most important information found in the superblock stored on a device. This information includes the file system as well as the device UUID. For further analysis, it also displays information about the pool including its label, if one was given, and its total number of devices.

Listing 2. Using `pls` for a pool membership detection of a single disk.

```
$ pls/BTRFS/raid10_5disks/disk1
Part of BTRFS pool:
Label: RAID10Pool
File system UUID:
D369B8F5-53EA-4DA9-A020-F6E585AA67D4
Root tree root address: 45711360
Chunk tree root address: 20987904
Generation: 42
Chunk root generation: 39
Total bytes: 5242880000
Number of devices: 5
Device UUID: B3B74185-13B0-4D2A-8300-CA740C384F4B
Device ID: 1
Device total bytes: 1048576000
Device total bytes used: 1004535808
[...]
```

After detecting the single members of a BTRFS pool, `pls` can be used to analyze the pool configuration. For this, it provides the `-P` parameter, indicating that the input volumes are now analyzed as a pool. Listing 3 shows that all of the five devices of the BTRFS pool have been successfully detected. It also gives information about the RAID levels used for each type of chunk items in the pool as well as the available and total number of these chunk items. In a case of missing pool members, this provides information about the availability of metadata and thus the chances of recovering data.

Listing 3. Initial analysis of acquired volumes using `pls`.

```
$ pls -P/BTRFS/raid10_5disks/Detected BTRFS Pool
Label: RAID10Pool
File system UUID:
D369B8F5-53EA-4DA9-A020-F6E585AA67D4
Number of devices: 5 (5 detected)
—
Device ID: 1 (B3B74185-13B0-4D2A-8300-CA740C384F4B)
Device ID: 2 (66AAEB1A-8CBB-4979-89CF-56FB0C6C958A)
Device ID: 3 (71D7CC24-BBE3-4E31-B532-EDF15C5AC527)
Device ID: 4 (E84DA2D2-D5FE-4226-A8AA-52D1AD8988B5)
Device ID: 5 (C7099E88-5597-4776-9EE0-3D6B662E53B3)
System chunks: RAID10 (1/1)
Metadata chunks: RAID10 (1/1)
Data chunks: RAID10 (6/6)
```

After a pool has successfully been detected, the other tools provided by our implementation can be used for a forensic analysis

including file listings, timeline generation, or data extraction. In line with the implementation of Hilgert et al., we have implemented support for BTRFS to the following tools of TSK:

- **fsstat:** Shows general information about the BTRFS file system including its snapshots and subvolumes.
- **fls:** Lists all files and directories of a BTRFS file system, snapshot, or subvolume.
- **istat:** Shows metadata information about an object, which is uniquely identified by its object ID shown in `fls` and its parent file system, subvolume, or snapshot.
- **icat:** Extracts the data associated with a metadata structure.

5.2. Snapshots

As mentioned earlier, BTRFS offers the possibility to create snapshots of existing file systems. Remember that a snapshot saves the current state of the file system and can afterwards be used to revert the file system to the point in time when the snapshot was taken. What is more, snapshots are part of the file system and thus always in a consistent state. Hence, they represent an outstanding source for the recovery of deleted files. Enabling the detection and analysis of snapshots is therefore an important analysis technique during the forensic examination of a BTRFS file system.

Listing 4. Listing all available snapshots and subvolumes using `fsstat`.

```
$ fsstat -P/BTRFS/raid10_5disks/
File system UUID:
D369B8F5-53EA-4DA9-A020-F6E585AA67D4
[...]
The following subvolumes or snapshots were found:
259 snapshot_2017-12-06
260 snapshot_2017-12-13
261 snapshot_2017-12-20
```

Since snapshots are subvolumes in BTRFS, the following description applies not only to snapshots but also to subvolumes in general. For each snapshot, a separate file system tree is created. These file system trees can be analyzed similar to the default "top-level" file system. Each of these file system trees is referenced by a `ROOT_REF` in the root tree containing for example the ID of the file system tree or the name of the snapshot. Furthermore, a root item is added to the root tree storing a reference to the root node of the tree and additional information like the number of the generation that created the snapshot. These generation numbers are always updated whenever a transaction is written to the BTRFS pool.

Using `fsstat`, we are able to list all subvolumes and snapshots for a particular BTRFS file system as shown in Listing 4. Afterwards, the corresponding name can be used to list, extract, or recover files from snapshots. This is done by passing the snapshot as an argument to the other file system analysis tools like `fls`. Listing 5 shows an example in which `snapshot_2017-12-06` contains multiple files, which have been deleted in the most recent version of the file system tree. These deleted files, still available in the snapshot, are located in the `/home/user/directory` and can be restored using `icat`.

Listing 5. Recovering files using snapshots.

```
$ fls -P/BTRFS/raid10_5disks/
r/r 265: 043349.ppt
d/d 266: data
+ r/r 267: 018367.docx
+ r/r 268: 018370.docx
+ r/r 269: 018371.docx
d/d 270: home
+ d/d 271: user
++ r/r 272: 516411.docx
$ fls -P/BTRFS/raid10_5disks/snapshot_2017-12-06
r/r 265: 043349.ppt
d/d 266: data
+ r/r 267: 018367.docx
+ r/r 268: 018370.docx
+ r/r 269: 018371.docx
d/d 270: home
+ d/d 271: user
++ r/r 272: 516411.docx
++ r/r 275: 043083.html
++ r/r 276: 043084.html
++ r/r 277: 043088.txt
```

5.3. Metadata-based file recovery

BTRFS only stores allocated metadata for files and directories in its trees. For this reason, searching for unallocated metadata structures for file recovery in the most recent tree is not an option. Nevertheless, it is possible to look at still existing metadata structures of older trees. Due to the copy-on-write principle used by BTRFS, each transaction creates a new root tree and results in a new generation number. Thus, accessing an old root tree makes it possible to jump back in time, analyze a previous version of the file system, and extract deleted files.

Unfortunately, there are two issues when trying to perform file recovery in this manner. First, we are dealing with possibly inconsistent metadata. The analysis is performed on artifacts of the file system and chances are high that parts of them have already been overwritten. If this happens to metadata, it will not be possible to continue the analysis.

Second, the location of an older root tree needs to be determined. Apart from scanning the complete set of volumes for these root structures, file systems sometimes keep track of these locations. ZFS for example stores the last 128 versions of its root structure (called überblock) in an array. In BTRFS, unfortunately only four versions of a structure referred to as `btrfs_root_backup` are stored in an array in each superblock.

Listing 6. Backup root addresses stored in the superblock shown by `pls`.

```
$ pls /BTRFS/raid10_5disks/disk1[...]
Backup Roots:
1. tree root at 45711360 (generation: 42)
chunk tree root at 20987904 (generation: 39)
2. tree root at 44646400 (generation: 39)
chunk tree root at 20987904 (generation: 39)
3. tree root at 45285376 (generation: 40)
chunk tree root at 20987904 (generation: 39)
4. tree root at 45629440 (generation: 41)
chunk tree root at 20987904 (generation: 39)
```

As shown in Listing 6, these backup structures can be listed using pls. Though the output only shows the logical addresses of the root and chunk trees from previous generation numbers, the backup structure also contains the logical addresses of the roots of other important trees, like the extent or device tree. Furthermore, it also stores the generation number corresponding to each tree and its logical address. These generation numbers are not necessarily the same for each tree in a backup structure, since not every transaction modifies, for example, the chunk or device tree. In Listing 6, the chunk tree at generation 39 is still used for the

mapping of the most recent root tree. In our example, it can be seen that the most recent generation of the pool found in the superblock is 42. The root tree address for that generation is 45711360 and can already be found in the backup roots. The corresponding chunk tree is stored at address 20987904.

Listing 7. File listings using the most recent version and an older version of the tree root of the BTRFS file system.

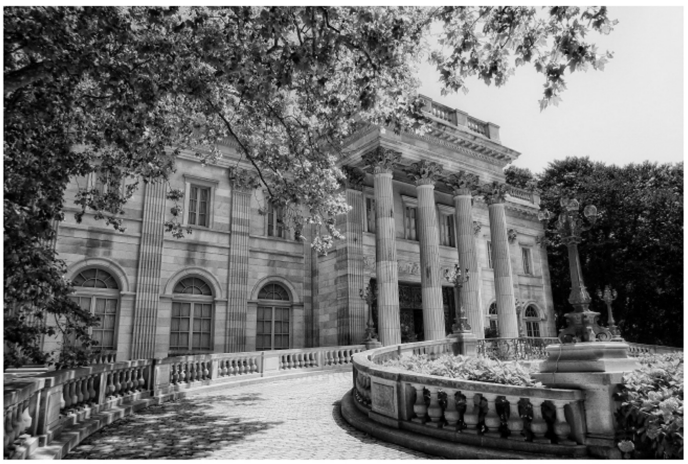
```
$ fls -P/BTRFS/raid10_5disks/
r/r 265: 043349.ppt
d/d 266: data
+ r/r 267: 018367.docx
+ r/r 268: 018370.docx
+ r/r 269: 018371.docx
$ fls -P/BTRFS/raid10_5disks/-T 41
using rootTree at logical address: 45629440 (generation 41)
r/r 265: 043349.ppt
d/d 266: data
+ r/r 267: 018367.docx
+ r/r 268: 018370.docx
+ r/r 269: 018371.docx
r/r 279: IMG00561.jpg
```

The ZFS extension of TSK by Hilgert et al. provides a parameter to specify an older transaction group number for the recovery of deleted files by using the corresponding überblock stored in the array. In a similar manner, our tool expects the generation number as a parameter, detects the corresponding backup structure and uses the provided addresses for the reconstruction of the BTRFS file system. Listing 7 shows two file listings, one performed with the tree root referenced by the superblock—the most recent version—followed by one performed by specifying the previous generation 41. By comparing the outputs, an image which is not available in the most recent version can be found. Similar to the usage of `fls`, we were able to successfully recover the file by using `icat` with the same parameter. However, as already mentioned, this type of file recovery does not always yield sufficient results.

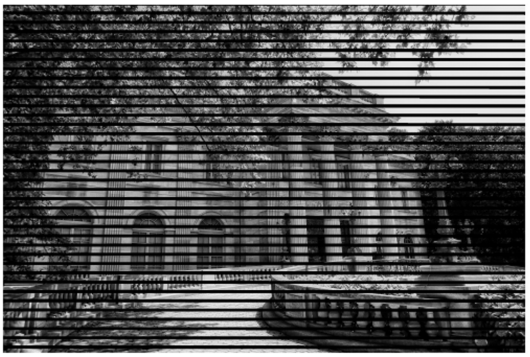
5.4. Missing disk

During a forensic examination, an investigator should be in a position to perform an analysis of a BTRFS file system even if there are disks missing. There are various reasons for missing disks: e.g. a disk might have been destroyed or formatted before it could be acquired. Similar to what Hilgert et al. observed for ZFS Hilgert et al. (2017), missing disks render the normal file system tools useless. That is, a BTRFS file system spanning its data over multiple devices cannot be successfully accessed if there is a single device missing. This even holds for scenarios in which at least some of the data is still recoverable.

As a test scenario, we created a BTRFS file system comprised of three disks with the metadata profile set to RAID1 and the data profile set to RAID0. This means that with a single missing disk all metadata should still be completely accessible whereas on average



(a) Original image stored on the pool.



(b) Extracted image after removing one of the disks of the pool disks.

(c) Extracted image after removing two of the disks of the pool disks.

Fig. 4. Extracting data from a degraded BTRFS pool missing disks.

one third of the actual file data is expected to be missing. As shown in Listing 8, `btrfs filesystem show` recognizes the missing disk, but cannot provide any additional information about it.

Listing 8. BTRFS pool with missing disk2.

```
$ btrfs filesystem show
warning, device 2 is missing
warning devid 2 not found already
Label: none uuid:
18f4475c-0b32-47c8-8827-739c6b8328d0
Total devices 3 FS bytes used 89.91MiB
devid 1 size 500.00MiB used 139.00MiB path/dev/sda
devid 3 size 500.00MiB used 147.00MiB path/dev/sdc
*** Some devices missing
```

Trying to mount the file system in a degraded state using the mount option `-o degraded`, fails with the message: `BTRFS: missing devices(1) exceeds the limit(0), writeable mount is not allowed`. After mounting the file system readonly , it is possible to browse the directories and files. This is because the metadata containing this information is still available since it was mirrored to two independent stripes. Nevertheless, trying to access any of the files—whose content is not stored inline in metadata—fails with `cp: error reading/mnt/missing_disk/IMG00158.bmp: Input/output error`.

In line with what Hilgert et al. did to deal with missing disks in ZFS Hilgert et al. (2017), we also implemented direct access to the file systems internal structures instead of relying on tools provided by the file system as described in Section 4.2. This fact enables us to replace any data of missing devices with zeros so that we are able to extract the data which is still available and store it at the right offsets in the file.

Listing 9. BTRFS pool with missing disk2.

```
$ pls/BTRFS/missing_disk/
[...]
Number of devices: 3 (2 detected)
—
Device ID: 2 (2A756EDA-87F4-44CB-9745-361026DC91C8)
Device ID: 3 (AA193982-4C41-4E44-A2A5-350730E35E9B)
System chunks: RAID1 (1/1)
Metadata chunks: RAID1 (1/1)
Data chunks: RAID0 (0/2)
```

Using `pls` on the test scenario gives us additional information about the chunk items of the detected pool as depicted in Listing 9. As we can see, all of the data chunks are incomplete. However, all metadata chunks are completely available due to the RAID1 configuration. This enables us to perform a recovery by filling the missing parts of the data.

An example for this recovery is depicted in Fig. 4. In this scenario, the common BTRFS tools—even though they provide support for degraded pools—would not provide any of the data, though roughly two thirds of it are still available. To take this even a step further, we have removed a second disk from our test scenario. Since the metadata is mirrored in such a way, that it is still available on the one and only remaining disk, our implementation is able to successfully detect, pad, and extract the remaining data of the image. As shown in Fig. 4c, this is sufficient to obtain an identifiable image, in a case, in which former tools and methods returned nothing at all.

6. Conclusion and future research

Just like Hilgert et al. we are convinced that pooled storage file systems will become common in forensic investigations any time soon. At the time of writing we hold the opinion that the forensic community is not well enough prepared for file systems of this class: there are virtually no research papers and the tools—both commercial ones as well as their open source counterparts—do not support them.

In this paper we tie in with the efforts of Hilgert et al. to close this serious gap. We confirmed that their proposed model is indeed applicable to BTRFS. Subsequently, we followed their model to implement BTRFS support to TSK. This implementation enables practitioners to perform forensic analyses of BTRFS file systems. Moreover, it can be used by the academic community for further research regarding BTRFS. While there have already been approaches to add BTRFS support before, to the best of our knowledge we provide the first implementation being able to handle multiple device configurations correctly and efficiently.

In addition to the implementation, which is publicly available and open source Hilgert et al. (2018), we also show how to perform a forensic analysis of a BTRFS file system using our extended TSK version. Furthermore, we highlight features of BTRFS of particular interest during a forensic investigation. These include snapshots and means to be able to deal with missing or corrupted disks. Again, we also show how our TSK extension can be used to utilize these features during an analysis.

References

Bacik, J., 2012. Btrfs: the Swiss army knife of storage. USENIX Login 37, 7–15.

btrfs Wiki, 2017a. Data Structures - Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Data_Structures.

btrfs Wiki, 2017b. Manpage/mkfs.btrfs - Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Manpage/mkfs.btrfs.

btrfs Wiki, 2017c. RAID56-btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/RAID56.

btrfs Wiki, 2017d. Status - Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Status.

btrfs Wiki, 2017e. SysadminGuide - Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/SysadminGuide.

btrfs Wiki, 2018a. Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/Main_Page.

btrfs Wiki, 2018b. On-disk Format - Btrfs Wiki. https://btrfs.wiki.kernel.org/index.php/On-disk_Format.

Carrier, B., 2005. File system forensic analysis. Addison-wesley professional.

Carrier, B., 2017. The Sleuth Kit. https://www.sleuthkit.org/sleuthkit/.

Fleischmann, Stefan, 2012. X-ways Forum: X64-ways Forensics 16.4. http://www.x-ways.net/winhex/forum/messages/1/3685.html?1359801502.

Hilgert, J.N., Lambertz, M., Plohmann, D., 2017. Extending the Sleuth Kit and its underlying model for pooled storage file system forensic analysis. Digit. Invest. 22, S76–S85.

Hilgert, J.N., Lambertz, M., Yang, S., 2018. The Sleuth Kit with Support for BTRFS. https://github.com/fkie-cad/sleuthkit.

Pöschel, Stefan, 2015. Btrfs Support by Basicmaster · Pull Request #413 · Sleuthkit/sleuthkit. https://github.com/sleuthkit/sleuthkit/pull/413.

Rodeh, O., Bacik, J., Mason, C., 2013. BTRFS: the Linux B-Tree filesystem. Transact. Storage (TOS) 9 (9), 1–9, 32.