Digital Investigation 22 (2017) S76–S85

Contents lists available at ScienceDirect

Digital Investigation

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

DFRWS 2017 USA — Proceedings of the Seventeenth Annual DFRWS USA

Extending The Sleuth Kit and its underlying model for pooled storage file system forensic analysis

Jan-Niclas Hilgert*, Martin Lambertz, Daniel Plohmann

Fraunhofer FKIE, Zanderstr. 5, 53177 Bonn, Germany

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

A B S T R A C T

Carrier's book *File System Forensic Analysis* is one of the most comprehensive sources when it comes to the forensic analysis of file systems. Published in 2005, it provides details about the most commonly used file systems of that time as well as a process model to analyze file systems in general. The Sleuth Kit is the implementation of Carrier's model and it is still widely used during forensic analyses today—standalone or as a basis for forensic suites such as Autopsy.

While The Sleuth Kit is still actively maintained, the model has not seen any updates since then. Moreover, there is no support for modern file systems implementing new paradigms such as pooled storage.

In this paper, we present an update to Carrier's model which enables the analysis of pooled storage file systems. To demonstrate that our model is suitable, we implemented it for ZFS—a file system for large scale storage, cloud, and virtualization environments—and show how to perform an analysis of this file system using our model and extended toolkit.

Introduction

File systems play a vital part during a digital forensic investigation. Their analysis enables the collection and recovery of data and files—the digital artifacts necessary for further analysis steps. Unfortunately, each file system differs in many aspects such as the data structures it uses, the layout it follows, or the features it implements. For this reason, Brian Carrier introduced a universal model for a file system forensic analysis in 2005 (Carrier, 2005). He also provided an implementation of his model: The Sleuth Kit (TSK) (Carrier, 2017b), a forensic toolkit providing means for the analysis, recovery, and collection of digital evidence. Carrier's model and The Sleuth Kit enable investigators to perform a file system forensic analysis without requiring an extensive background knowledge of the underlying file system and its peculiarities. Moreover, it serves as a basis for further analysis techniques and tools (Carrier, 2017a; Garfinkel, 2009; Buchholz and Falk, 2005).

Although the model works great on file systems that were in use at the time of its publication more than a decade ago, its limitations become obvious when trying to apply it to modern file systems. In the last years, relatively new file systems like ZFS, BTRFS, or ReFS have gained more importance for users. A common concept these file systems share is pooled storage. This concept violates the idea of "one file system is assigned to one volume", which Carrier's model is based on (a volume in this case can be any kind of logical or physical volume like a partition or a RAID). Instead, multiple volumes are combined to form a pool, which can be accessed by multiple file systems. As a result of this change, the model and thus also TSK cannot be applied to modern file systems implementing pooled storage without revision. This leaves investigators with a serious gap in forensic analysis capabilities because these file systems and especially their underlying concepts will most likely become the future in the area of file systems.

In this paper, we present a revision of Carrier's model, which makes it applicable to pooled storage file systems like ZFS and BTRFS. Furthermore—just like Carrier when he introduced his model—we provide an implementation of our extended model for ZFS (Hilgert et al., 2017) proving it to be applicable to pooled storage file systems. In addition to standard file system meta data such as timestamps, file ownership, and file listings, our implementation enables the recovery of deleted data by reconstructing old ZFS tree structures. This method makes it possible to recover the state of a ZFS file system from certain points in the past. Also,

* Corresponding author.
E-mail addresses: jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), martin.lambertz@fkie.fraunhofer.de (M. Lambertz), daniel.plohmann@fkie.fraunhofer.de (D. Plohmann).

http://dx.doi.org/10.1016/j.diin.2017.06.003

our implementation is able to deal with missing disks of a pool, so that a forensic analysis can be performed on incomplete pools, which are neither importable nor accessible by common tools.

File system forensic analysis

In this section, we give a brief recap of Carrier's theoretical model and its implementation in TSK as a background for describing our extension.

Theoretical model

Carrier's model (depicted in Fig. 1) divides a file system forensic analysis into four interdependent steps where the output of one step is used as input for the next. Therefore, it is necessary to follow these steps one by one in the correct order.

The first step is the *physical media analysis*, which deals with the acquisition of data from storage devices. At that point, the data is considered a sequence of bytes only and not interpreted at all.

In the *volume analysis*, the acquired data is scanned for volume structures. Possible types of volumes include partitions, RAIDs, and logical volumes of volume groups. A complete disk can also be a volume, e.g. when a file system is used directly on a raw device. Moreover, volumes can be combined in an arbitrary number of ways.

After the underlying volume structure has been identified, each volume can be analyzed in the *file system analysis* step. Here, the data stored on each volume is interpreted as a file system and directories, files, and their meta data are collected and recovered from the detected file system. In his model, Carrier divides file system data into five categories:

- **File System Category:** Contains file system specific data used to describe the layout of the file system.
- **Meta Data Category:** All data which is used to describe files and directories belongs to this category. This includes e.g. temporal information or file sizes.
- **Content Category:** Most of the data can be found in this category. It contains the actual content of files stored on the file system.

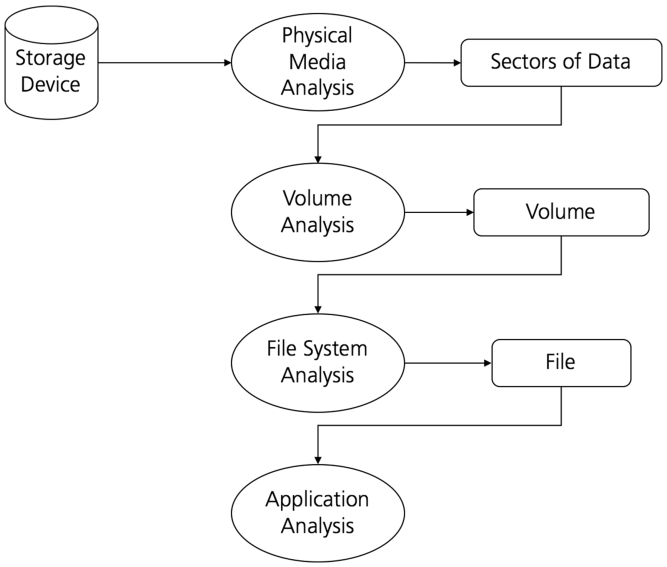

Fig. 1. Standard model for a file system forensic analysis by Brian Carrier (Carrier, 2005).

- **File Name Category:** This category is also referred to as the *human interface category* since its data only provides a name in order to identify files more easily.
- **Application Category:** Data in this category is not essential for the file system and only implemented to provide additional features, which would be less efficient if implemented on a higher software level.

After the digital evidence has been collected, it is processed during the *application analysis*, which is the fourth and last step of Carrier's model. Data is interpreted on content-level and analyzed using application-level techniques like searches in documents or detailed analyses of malicious software.

The Sleuth Kit

TSK is a forensic toolkit mainly developed by Carrier and the implementation of the model described in the previous section. It provides various tools for a file system forensic analysis, which operate on different categories of the model. These tools include:

- **`mmls`:** Analyzes a single volume providing information about its layout.
- **`fsstat`**: Detects the file system type stored on a single volume and presents statistics and meta data about it.
- **`fls`**: Lists all files and directories of a file system stored on a single volume.
- **`istat`**: Presents information of a given meta data structure.
- **`icat`**: Extracts data belonging to a meta data structure.

Pooled storage file systems

When Carrier published his model in 2005 there was a one-to-one association between a file system and a volume. That is, one volume was formatted with one file system and one file system spanned one volume (Bonwick et al., 2003). System administrators had to carefully plan the volume structure to meet the desired storage requirements. A mirror RAID for example provides reliability by storing multiple copies of the data, a striped RAID increases efficiency by employing multiple disks at the same time, and volume groups make it possible to divide the available space logically. The drawback of this concept becomes clear when storage has to be resized. Instead of simply adding or removing a disk, this process usually involves complicated file system resizing, RAID resilvering, or other convoluted tasks, making this seemingly easy job a rather painful and daring venture.

Pooled storage overcomes these issues by combining the available storage devices into a pool, which is shared between all file systems. No file system has a fixed size and thus never needs to be resized as it simply adapts to the available space of the pool. Of course, file system sizes can still be logically bounded by using reservations and quotas. Furthermore, reducing and increasing the available pooled storage becomes a simple task. Whenever a new device is added to the pool, it begins to provide the newly gained space. On the other hand, when a device is removed, the data is dynamically shifted to other available parts of the pool without the file system noticing it. This ease of use is one of the main reasons that such pooled storage file systems like ZFS, BTRFS, and ReFS enjoy great popularity.

For the implementation of pooled storage, modern file systems are no longer stored on top of single volumes. Instead, the file system including its data and meta data is stored across all available volumes in the pool. For this reason, modern file systems implement their own kind of volume management functionality, which

governs the distribution of the data across all devices. This volume management functionality is responsible for two major tasks. First, it has to keep track of all members belonging to a specific pool. This is done by storing additional information on these members such as a unique identifier of the pool they belong to and in some cases the layout of the whole pool configuration. This task is essential in order to be able to tell if a pool is complete and can be accessed without any errors resulting from missing members.

Second, it needs to provide means to define and keep track of the exact location where data is stored on the pool members. This can for example be realized by an additional structure, which identifies a unique pool member and specifies an offset. Another possibility is a mapping between the logical address space of the pool and the physical addresses of its available members.

Extending Carrier's model

In order to extend Carrier's model, we have to evaluate whether there are any steps that may need to be changed, removed, or added.

The physical media analysis obviously does not need any change since it only interprets data as a sequence of bytes and has no knowledge of file systems or pooled storage concepts. The same holds for the last step, the application analysis, which is performed on already collected digital artifacts and is thus also independent from the underlying file system.

As described in the previous section, pooled storage file systems need to provide some kind of volume management functionality. This integrated volume manager can be compared to the established volume managers used to create partitions, RAIDs, or volume groups which are dealt with in the volume analysis step in Carrier's original model. Unfortunately, the integrated volume management capabilities of pooled storage file systems have a major limitation from a forensic point of view: when used to access a pool, the corresponding file system is directly mounted to the operating system. This means that a lot of the file system's data, including information about its layout, meta data, and deleted files, are not accessible. A reconstructed RAID or volume group on the other hand, makes up a block device storing the complete data of a file system which can be mounted afterwards. The file system analysis has to be performed on such a block device, because it needs access to all of the file system's data. When using the integrated volume manager functionality of pooled storage file systems, this is no longer possible as we end up with a reconstructed and already mounted file system. We are not given a block device containing the complete data of the pool for this analysis step.

Requirements

For the aforementioned reason, it is necessary to extend the established model by an additional step. This step is responsible for the analysis of the pool and the volume management functionality of pooled storage file systems. This is required in order to close the gap which was created by integrating volume managers into the file systems themselves and, as a consequence, limiting the access to essential data needed for a forensic analysis. A pool analysis needs to provide information about how and where the data is stored across all devices of the pool so that a direct access to the data is assured during the file system analysis step.

In summary, the new step for pool analysis in our extended model needs to be able to achieve the following goals in order to enable a file system forensic analysis of a pooled storage file system:

- Detect members of a pooled storage file system.
- Analyze multiple volumes and identify their corresponding pool.
- Analyze a complete pool consisting of multiple volumes and its configuration.
- Provide functionality to access the correct offsets on the correct members of a pool according to the means specified by the pooled storage file system.
- Give access to all of a file system's structural data (e.g. file system data and meta data).
- Be able to deal with incomplete pools, e.g. when a member is missing.

Extension

As shown in Fig. 2, the pool analysis is added between the volume and the file system analysis steps of Carrier's model. The traditional volume analysis step still performs the detection of common volume structures like partitions, RAIDs, or logical volumes. These volumes are used as the input for the pool analysis step, since pools can also consist of these types of volumes instead of raw devices exclusively. Our newly added pool analysis step analyzes if the input volumes are part of a pool. If they are not, they are passed directly to the file system analysis without further actions. This is the case for established and non-pooled storage file systems. If they are part of a pool, the pool analysis can yield two different results. Importable pools can be reassembled using common tools, which results in a reconstructed pool with limited access (shown on the left in Fig. 2). This pool can be used to go through the most recent version of files only and does not enable a file system analysis. On the other hand, in order to perform a file system analysis, the pool analysis has to result in a pool with direct access. This pool needs to provide the functionality required to perform a file system analysis directly on the pool members including the mapping from logical to physical addresses.

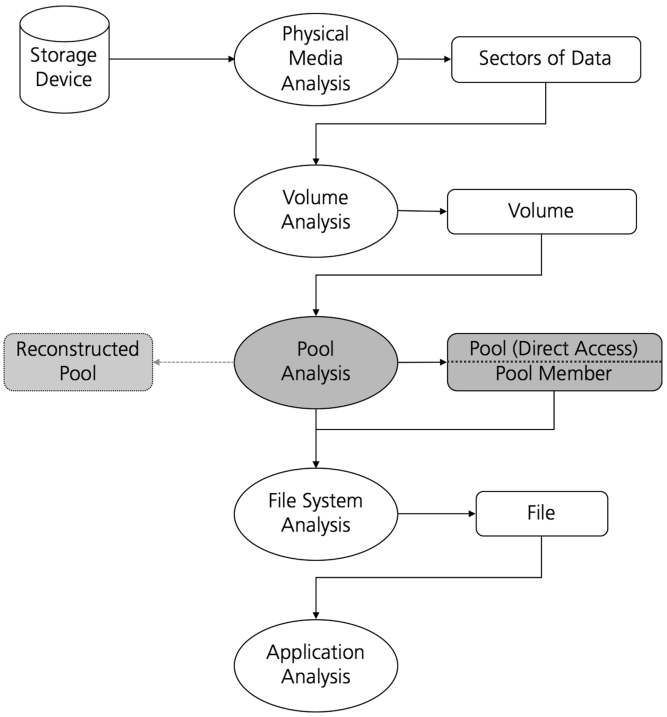

Fig. 2. Extended model for a file system forensic analysis for pooled storage file systems.

Note that the flow of the model is not always strictly monotonic. For instance, a pooled storage file system may enable the user to create own logical volumes as block devices made up of storage from the pool. These block devices can again serve as input for the volume analysis. Similarly, files could contain a file system or belong to a volume group. We chose to adapt Carrier's illustration layout of the model, and decided analogously to exclude these kinds of recursion, which may appear during a forensic analysis.

Implementation

In this section, we describe how we incorporated our extended theoretical model into TSK equipping it with means to analyze file systems with pooled storage. In addition to the general capabilities to analyze pooled storage file systems, we also used our extended TSK version to implement analysis commands for a concrete file system of this class. We chose ZFS as an example here, because it enjoys great popularity (openZFS, 2016; iXsystems, Inc., 2017) and is the oldest and most mature pooled storage file system providing the largest stable feature set when it comes to creating pool structures. Moreover, its source code is open source which enables a detailed analysis of the file system internals.

Extending The Sleuth Kit

In TSK, two main structures are used to access volumes and file systems. `TSK_IMG_INFO` is created after opening a volume and used in order to access its content. As already described, pooled storage file systems do not make a block device available which could be used to create this structure. Hence, a `TSK_IMG_INFO` cannot be created for a storage pool. Similarly, file systems are represented by a `TSK_FS_INFO` structure. It contains information like the number of available blocks in a file system, the address of its first block, or its ID. Since pooled storage file systems do not have a fixed size, some of these attributes are obsolete for them. Nevertheless, information like the used number of bytes or file system type are also available for pooled storage file systems.

For the extension of TSK, we introduce a new structure called `TSK_POOL_INFO`, which is used during the pool analysis (see Fig. 3). It is added between the volume analysis and file system analysis (`TSK_IMG_INFO` and `TSK_FS_INFO` respectively). This structure stores pool information, which should be available independent of the concrete pooled storage file system. Examples are the pool name, ID of the pool, or number of pool members. Furthermore, it creates multiple `TSK_IMG_INFO` objects, one for each pool member. Additionally, these objects can also be of type `TSK_VS_PART_INFO`—a TSK structure for handling detected partitions. Similar to the file system functions in TSK, also the pool analysis is file system dependent. Each concrete implementation of a pooled storage file system might use its own concepts and methods. Therefore, a per file system implementation is unavoidable here unfortunately. This concrete implementation is responsible for abstracting the peculiarities of the file systems providing features such as the detection of pool members, the pool configuration, parsing of internal file system data structures as well as direct access to the pool members.

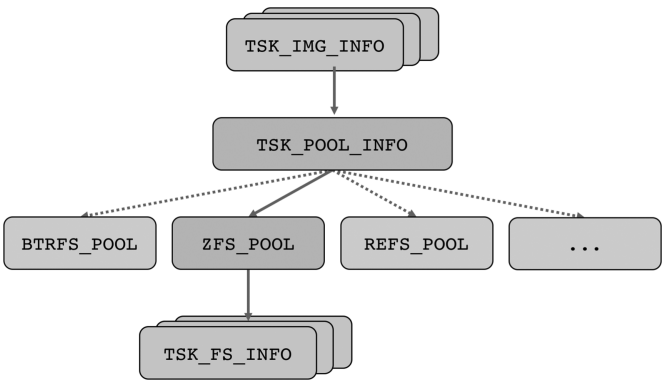

Fig. 3. Implementation of the extended model in The Sleuth Kit.

The functionality provided by the new pool objects is exposed to an analyst via the new TSK command `pls`. An example on how to use it can be found in our evaluation.

Our extension has minimal impact on the rest of TSK and does not affect its previous functionality as well as its commands and usage for established file systems.

ZFS

ZFS was first presented in 2003 (Bonwick et al., 2003) and initially developed for Solaris. Nowadays, it is available for multiple other major platforms including FreeBSD, MacOS, and Linux.

Volume management

A pool in ZFS is referred to as a zpool, which consists of one or more *top-level virtual devices (vdevs)*. Data in this pool is striped across all of these top-level vdevs. A vdev in turn consists of one or more members. These child members store *vdev labels* in their first and last sectors containing information about the corresponding zpool and describing to which top-level vdev they belong. ZFS supports different types of top-level vdevs (The FreeBSD Documentation Project, 2017):

- A *file* is simply a single file, which is used as a member of the pool. No redundancy or increase of efficiency is given in this case.
- A *disk* can be any kind of volume including partitions, RAIDs, or logical volumes created using other volume managers. Similar to files, these top-level vdevs provide no redundancy or increase of efficiency.
- A *mirror* top-level vdev consists of one or more disks or files. The data stored on this top-level vdev is copied to each of its children.
- A *raidz* is a special structure in ZFS, which can be compared to RAID level 3 (Leventhal, 2010). Depending on the chosen type (raidz1, raidz2, or raidz3) it tolerates one, two, or three missing children.
- A *spare* vdev is used to indicate hot spare devices.
- The *log* vdev type is used for devices storing the ZFS Intent Log of a pool.
- *Cache* vdevs are used to store the L2ARC, a ZFS cache type which is used when the primary, in-memory cache is exhausted.

In this paper, we are mainly interested in the first four vdev types: file, disk, mirror, and raidz. These are the ones which actually define how and where data is stored on the pool. The log and cache types on the other hand mainly indicate what is stored and the spare vdev is only used to replace a faulty vdev of an existing pool configuration. While they are undoubtedly of interest during a forensic analysis in general they are not necessary for pool reconstruction and analysis.

Because data is always striped across all available top-level vdevs, the failure of one top-level vdev inevitably results in a loss of data since it is stored nowhere else across the pool. This must not be confused with missing children in a mirror or raidz top-level vdev. In these cases, it may still be possible to recover the data, as it is stored on other children *within* the top-level vdev.

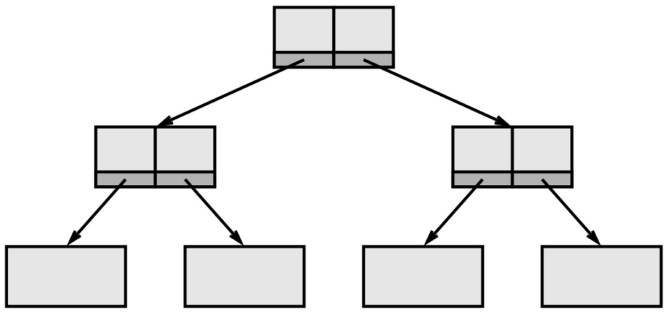

Fig. 4. Tree structure implemented in ZFS (Bonwick et al., 2003).

In order to access specific top-level vdevs, ZFS utilizes *data virtual addresses* (*DVAs*). Each DVA consists of the ID of the top-level virtual device and an offset. For disks and files, the offset refers directly to the offset where the data is stored on the disk or file respectively. The same holds for a mirror top-level vdev, where the data is stored at the specified offset on any of its children. When it comes to raidz, ZFS uses an algorithm to calculate the actual offset from the offset specified in the DVA. These DVAs are stored in structures referred to as *block pointers*. Depending on its importance, ZFS stores up to two additional copies of the data. This feature is referred to as the wideness of a block pointer. Apart from these up to three DVAs, block pointers also contain information about the size of the data which is stored at the location specified by the DVAs.

General structure

ZFS stores its data, meta data, and file system data in a tree-like structure as shown in Fig. 4. The root of the tree is referred to as the *überblock*, which points to multiple *dataset directories*. Each file system in ZFS is implemented by one dataset directory keeping track of snapshots or clones by using *datasets*. A dataset in ZFS points to an *object set* storing multiple *dnodes*, the essential structure describing objects in ZFS. Dnodes are used to describe file system objects like dataset directories or datasets, but also files. Furthermore, ZFS uses the copy-on-write (COW) principle to store data. Each time a data block is changed, a new version of it is stored at a new location in the pool. Afterwards, the corresponding meta data and file system structures are rewritten to point to the new block and also stored at a new location. Finally, the new überblock is stored pointing to the new ZFS tree. This method ensures that in case of a crash, the file system is always in a consistent state.

A more detailed explanation of ZFS, its data structures, and layout can be found in the on-disk specification (Sun Microsystems, Inc., 2006).

Implementing the model for ZFS

Integrating ZFS into the extended version of TSK involves the implementation of two major aspects. First, it is necessary to analyze multiple disks and to detect the corresponding pool and its underlying configuration. Second, direct access to the data has to be provided as this is necessary for the file system analysis.

Pool detection

We detect ZFS pool members by scanning the volumes for vdev labels and parsing their name–value pairs. After finding potential pool members, we check the plausibility of the candidates by validating the parsed values. For confirmed pool members, we extract the unique identifier of the pool along with the unique identifier of the volume. This approach enables the identification of all pool members. Moreover, we are able to easily detect duplicates and volumes which do not belong to a ZFS pool.

In the next step, the pool configuration is reconstructed by examining the top-level vdev data stored in the vdev labels. It contains information about the total number of top-level vdevs in the pool, the type of the top-level vdev a volume belongs to, and the unique IDs of other children belonging to the same top-level vdev. Afterwards, the detected pool configuration is used to evaluate the completeness of each top-level vdev and subsequently of the whole pool. If at least one top-level vdev is not reconstructible (e.g. due to too many missing children), the whole pool becomes incomplete since some of its data is missing. Our implementation stores information about the detected and expected top-level virtual devices and their availability. This information is useful, in cases of double- or triple-wide block pointers containing a DVA, referencing an unavailable top-level vdev. In these cases, our implementation ignores this DVA and chooses one, which points to an available top-level vdev.

Direct pool access

Mapping the DVA stored in a block pointer to the correct offset of a pool member requires the IDs of the top-level virtual devices, which have been obtained in the previous step. Whenever a block pointer and thus a DVA is processed, the implementation directly returns the data depending on the top-level vdev type:

- For **files** or **disks** the data is directly extracted from the pool member at the offset specified in the DVA.
- For **mirrors**, one of the top-level vdev's available children is randomly chosen and the data at the offset specified in the DVA is extracted.
- For **raidz**, ZFS′ algorithm is used to compute the actual offset and member storing the data which is then extracted.

Evaluation

In this section, we provide three case studies of how our implementation enables a forensic analysis of ZFS. To ensure the correctness of our implementation we compared the results with the output of the ZFS debugger, whenever its functionality permitted it.

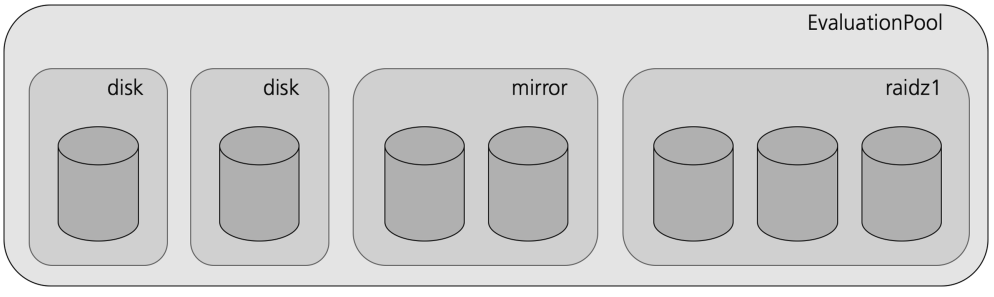

Fig. 5. Example `EvaluationPool` consisting of four top-level virtual devices and seven disks in total.

Scenario A: forensic analysis

In this scenario, we show a full iteration through our extended model using the modified TSK version. That is, we start from raw storage devices and end up with the extraction of the contents of a file stored on the file system.

We use the sample zpool depicted in Fig. 5 consisting of seven storage devices. Using these devices we configured four top-level virtual devices: two disks, a mirror, and a raidz1. This combination of different top-level vdev types is unlikely to be found in practice, but it serves well to show that all different top-level vdev types are supported by our implementation.

Starting with the raw storage devices we use our newly implemented `pls` indicating that the devices belong to a ZFS pool. Additionally, it already displays information about the corresponding pool stored in the vdev labels of the storage device. This information includes the detected überblocks. The most and second most recent überblocks are highlighted by << and < respectively.

Listing 1: `pls` command run on `disk5` of the `EvaluationPool`.

```
$ pls ./EvaluationPool/disk5
Part of zpool:
 name --> EvaluationPool
 [...]
 pool_guid --> 17664212653981624859
 top_guid --> 7111215423496479761
 guid --> 14371670631125040911
 vdev_children --> 4
 vdev_tree -->
    type --> raidz
    id  --> 3
    guid --> 7111215423496479761
    [...]
    children[0] -->
        type --> file
        id --> 0  guid --> 14371670631125040911
        path --> /EvaluationPool/disk5
    children[1] -->
        type --> file
        id --> 1   guid --> 6415778699177469711
        path --> /EvaluationPool/disk6
    children[2] -->
        type --> file
        id --> 2   guid --> 17716271440399145952
        path --> /EvaluationPool/disk7
    [...]

Information about überblocks:
004: 0x21000 | 2017-01-19T17:21:33 | TXG: 4
[...]
112: 0x3c000 | 2017-01-19T18:31:55 | TXG: 20720
114: 0x3c800 | 2017-01-25T11:22:11 | TXG: 22130
117: 0x3d400 | 2017-01-25T11:22:12 | TXG: 22133 <
120: 0x3e000 | 2017-01-25T09:26:19 | TXG: 20728
122: 0x3e800 | 2017-01-25T09:26:20 | TXG: 20730
123: 0x3ec00 | 2017-01-25T11:22:12 | TXG: 22139 <<
```

Afterwards, `pls` is used for the analysis of the whole pool. This is done by specifying a folder containing all of the acquired devices. This step identifies the pool configuration, eliminates possible duplicates, and provides information about the completeness of the pool as shown in Listing 2.

Listing 2: `pls` command run on multiple disks of the `EvaluationPool`.

```
$ pls ./EvaluationPool
Poolname: EvaluationPool
Pool-GUID: 17664212653981624859
Number of Top-Level Vdevs: 4
Detected Number of Top-Level vdevs: 4

disk (ID: 1)
   ./EvaluationPool/disk2 (ID: 1)

raidz1 (ID: 3)
   ./EvaluationPool/disk6 (ID: 1)
   ./EvaluationPool/disk5 (ID: 0)
   ./EvaluationPool/disk7 (ID: 2)

mirror (ID: 2)
   ./EvaluationPool/disk3 (ID: 0)
   ./EvaluationPool/disk4 (ID: 1)

disk (ID: 0)
   ./EvaluationPool/disk1 (ID: 0)
```

Since multiple file systems can be used on a single zpool, `fsstat` enables us to display an overview of the file systems in a pool. Listing 3 presents the output for our test pool also showing that our `fsstat` implementation is able to deal with nested file systems. Furthermore, detailed information about a file system can be read by specifying its complete path.

Listing 3: `fsstat` command run on the `EvaluationPool`.

```
$ fsstat ./EvaluationPool
Using Überblock: TXG 22139

zpool contains the following file systems / datasets:
    MOS
    EvaluationPool
    EvaluationPool/data
    EvaluationPool/data/UserA
    EvaluationPool/data/UserB
    EvaluationPool/boot
```

A file listing can be obtained by using `fls`. It displays the files and directories of all datasets as shown in Listing 4. Datasets are marked with an asterisk, because they can easily be confused with regular directories in ZFS. The number given in brackets refers to the object

number of the file's or directory's dnode in the corresponding dataset. By default, this command traverses through directories and datasets recursively.

Listing 4: `fls` command run on the `EvaluationPool`.

```
$ fls ./EvaluationPool
Using Überblock: TXG 22139

|---data (*Dataset) (7)
    |---UserA (*Dataset) (7)
        |---IMG_00132.jpg (7)
        |---IMG_00134.jpg (9)
        |---IMG_00133.jpg (10)
        |---IMG_00135.jpg (8)
    |---UserB (*Dataset) (8)
        |---Report_2017.pdf (7)
        |---Notes.txt (8)
|---boot (*Dataset) (8)
    |---users (8)
    |---start.conf (7)
```

Detailed information about a specific dnode in a file system can be obtained by using `istat`. A sample output is shown in Listing 5. We have to specify both, the object number as well as the name of the dataset. This is because each dataset has its own object numbers and, therefore, a dnode can only be uniquely identified by these two values together.

Last, the data of a dnode can be extracted by using the `icat` command.

Listing 5: `istat` for object number 8 and dataset `data/UserA` command run on the `EvaluationPool`.

```
$ istat ./EvaluationPool -d
    EvaluationPool/data/UserA -o 8
Using Überblock: TXG 22139

        Type: 19 (ZFS plain file)
        Number of Levels: 3
        Number of BlockPointer: 1
        Bonus Type: 44  | Bonus Length: 168

        DVA[0]: 1 : 0x54f200
        DVA[1]: 2 : 0xa6b600
        DVA[2]: 0 : 0x0
        Logical Size: 16384 bytes
        Physical Size: 512 bytes
        Checksum: 1b0748b427:9a[truncated]
        Compression: LZ4

        Bonus Information:
        Type: znode
        File Creation Time: 2017-01-19T17:26:10
        File's Size: 23000826
        Parent Directory Object ID: 4

        Level 0 Blocks:
        1 : 0x2cf200 | 0x20000P | 0x20000L
        1 : 0x28f200 | 0x20000P | 0x20000L
        2 : 0x52b600 | 0x20000P | 0x20000L
        [...]
        3 : 0x2482800 | 0x20000P | 0x20000L
        3 : 0x24e2800 | 0x20000P | 0x20000L
        3 : 0x24b2800 | 0x20000P | 0x20000L
```

Scenario B: recovering deleted data

Recovery of deleted files is generally possible because file systems usually do not actually remove the corresponding data, but only flag it as deleted in some kind of way. To restore deleted files in established file systems it is possible to scan the meta data for this flag. Starting from structures with this flag, an investigator can then recover parts or possibly all of the data of a deleted file. The important point here is that the meta data structures of deleted files are still a part of the file system.

In ZFS, this is not the case due to the COW principle it implements. Each time a change in the file system occurs, a new block is written to a new location somewhere on the available pool. Afterwards, the meta data is changed accordingly, so that it points to the new block. The old block is still present on the disk until it is overwritten. In contrast to the meta data of the established file systems, these blocks are not reachable from the root of the current ZFS tree anymore.

This means that we have to use an older version of the ZFS tree in order to find and recover deleted data from ZFS. Common tools, however, only provide access to the most recent version of the COW tree. Hence, no old (deleted or overwritten) data nor its meta data is accessible. By providing direct access to the pool, we are able to address the complete data stored in the pool including old versions of the COW structures (as indicated in Fig. 6). This is similar to file recovery for established file systems, where the meta data structures stored on the volume are analyzed.

To evaluate our file recovery procedure, we configured a zpool consisting of five disks and created a file system with the name `data` on it. Then we stored the image file `IMG_00134.jpg` on the `data` file system and deleted it. The upper part of Listing 6 shows that the file is indeed not present anymore if we use `fls` with the most recent überblock with the transaction group number (TXG) 660. In contrast, if we use the second most recent überblock with the TXG 656, `fls` includes the deleted file in its output.

Provided the file contents have not been overwritten, we are now able to recover the file using `icat`, again specifying the older überblock 656. During our small-scale tests we were always able to completely recover deleted files. In write-heavy environments this will most likely not be the case. Moreover, we expect that the exact configuration and size of the pool will certainly have an effect on the success probability of the file recovery process. A more in-depth analysis on the influence of these parameters remains to be performed.

Listing 6: File recovery by specifying an older überblock for the `fls` command.

```
$ fls ./myPool/
Using Überblock: TXG 660

Listing myPool ...

|---data (*Dataset) (12)
    |---IMG_00135.jpg (8)

$ fls ./myPool/ -u 656
Using Überblock: TXG 656

Listing myPool ...

|---data (*Dataset) (12)
    |---IMG_00134.jpg (7)
    |---IMG_00135.jpg (8)
```

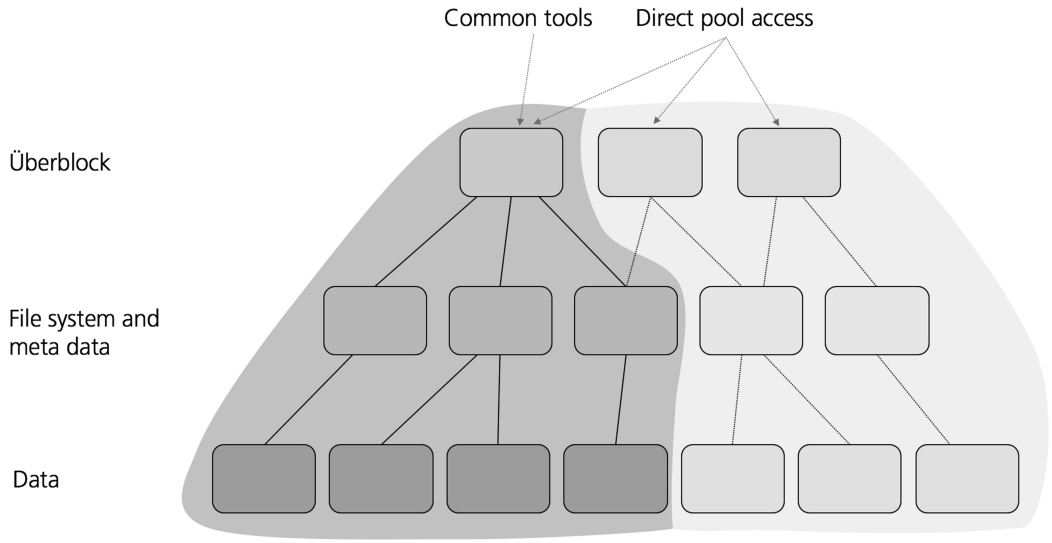

Fig. 6. Most recent ZFS tree accessible by common tools compared to old tree structures accessible after the pool analysis step.

Scenario C: reconstructing an incomplete pool

Whenever a top-level vdev is missing or corrupted, the pool cannot be imported and, consequently, its data cannot be accessed. Especially when only little data was stored on the missing top-level vdev, this behavior becomes a significant limitation. Imagine for instance the pool shown in Fig. 7 consisting of five top-level vdevs each being a single disk. Data stored by ZFS is now striped across all of these disk. This means that if one disk is missing, on average 80% of the pool's data will still be available but cannot be accessed. In fact, for some files even more than 80% may be available in case their data is only stored on the remaining disks. As we already defined in our requirements, a forensic tool should be able to extract the data which is still available.

Our extended implementation of TSK accesses the pool members directly. Therefore, it is not reliant on a successfully assembled pool. Furthermore and since we keep track of the availability of the pool's top-level vdevs, we are able to choose those DVAs of double- or triple wide block pointers, which are still pointing to available top-level vdevs. All of this enables us to analyze the file system data on the remaining disks, reconstruct the ZFS tree, and extract the available data from the incomplete pool. Fig. 8 illustrates the capability of this feature.

We took a ZFS pool as shown in Fig. 7 created a file system and stored an image on it. Then we removed the pool from the system and removed one of the disks. Afterwards, we tried to mount the file system again, which failed with the message: `cannot import 'myPool': one or more devices is currently unavailable`.

On the other hand, when using our extended TSK in a way as presented in Scenario A: forensic analysis, we were able to successfully recover the majority of the image as shown in Fig. 8. This is a scenario where existing state-of-the-art tools would return nothing at all.

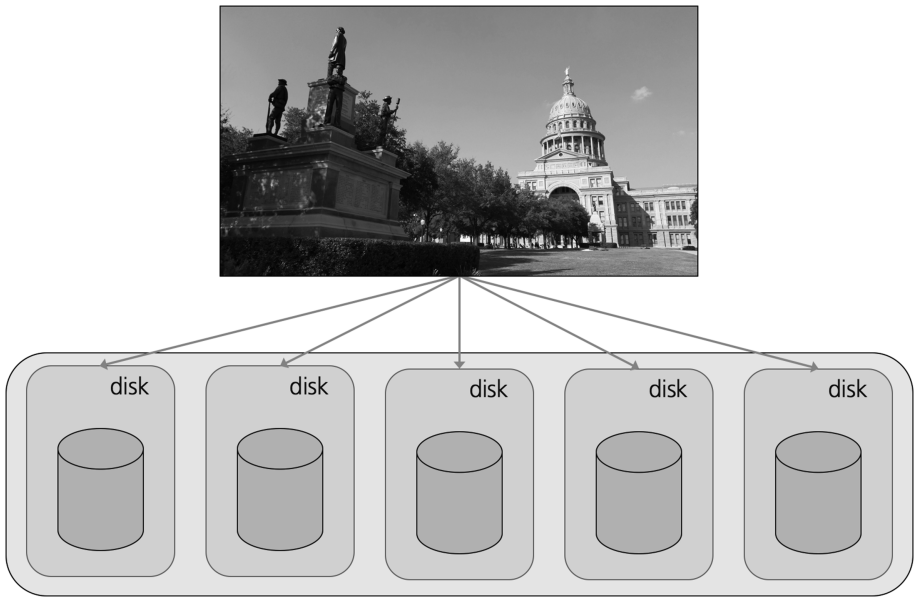

Fig. 7. Example image stored across the five top-level virtual devices of a zpool.

Fig. 8. Extracted example image of example pool with one missing disk.

Limitations

We are confident that our extended model can be applied to any pooled storage file system. However, the exact implementation as well as the possibilities of a forensic analysis strongly depend on the concrete file system, its features, methods, and layout. For example, ZFS stores multiple versions of global data, which we exploited for pool reconstruction in case of a missing top-level vdev. This feature is file system specific and not a part of pooled file systems in general. Furthermore, the aforementioned recovery of data by utilizing old data structures in pooled storage file systems is owed to the COW principle. Pooled storage file systems using another concept may not leave any fragments of old and deleted data behind. Yet they may implement other concepts and structures, which enable file recovery in a similar way.

Related work

Overall, we observed that pooled storage file systems and newer file systems in general did not receive appropriate attention. This holds for both, scientific research as well as forensic tool development.

This very problem was emphasized already in 2009 by Nicole Beebe (Beebe, 2009). In her article, Beebe argues that most of the research and knowledge in the area of digital forensics focused on Microsoft's operating systems and some of the better known Linux distributions. Furthermore, she highlights the importance to also consider non-standard systems and new technological developments. One of these new developments she explicitly mentions, are file systems including ZFS. We share Beebe's view and although we feel that eight years after her article was published at least some of her concerns have been addressed, the area of modern file systems still has not received the attention it deserves.

In the same year, Beebe et al. also published a paper elaborating on the forensic implications of ZFS (Beebe et al., 2009). The authors argue that ZFS introduces several aspects that may be beneficial during a forensic analysis. Here, they mention that ZFS creates multiple copies of data which are likely present in allocated as well as in unallocated space of a disk. Moreover, they reason that snapshots, clones, and the COW concept provide investigators with valuable insight into the chronological states of the file system and the data stored on it. On the other hand, the authors also highlight challenges introduced by ZFS. One of the major problems they describe is the compression of meta and user data built into ZFS. The article raises some very important questions regarding ZFS in a forensic context. However, most of these questions remain unanswered or unverified, which is also acknowledged by the authors. While our paper was not intended to answer the questions raised by Beebe et al., we found a lot of their statements confirmed. For instance exploiting the COW principle—like we did to recover deleted files—shows that ZFS indeed has certain features enabling some degree of “time travel” through the file system's history.

Max Bruning steps through a complete data walk of ZFS from the überblock to the actual data on his blog (Bruning, 2008). This data walk provides excellent insight into ZFS and its data structures. Although his work serves great as a basis for a manual file system analysis, it requires detailed knowledge of ZFS since all of the structures are parsed by hand. This is clearly not an efficient approach for an analyst during an investigation.

Leigh and Shi discussed a forensic timeline analysis of ZFS (Leigh, 2015), but did not focus on the whole process of a digital forensic analysis of ZFS nor its pooled storage functionality. Andrew Li described a forensic file recovery on ZFS, providing a proof of concept that a forensic analysis of ZFS is achievable (Li, 2009). Furthermore, he presented an extension for the ZFS debugger, which performs file recovery without using the file system layer. For this purpose, Li analyzes every ZFS structure until he arrives at the actual data. This principle is similar to the data walk presented by Bruning. Unfortunately, Li's extension only uses the active überblock and is thus not able to recover deleted data. Additionally, it can only deal with importable pools since it is based on the ZFS debugger.

At the time of writing, we were not able to find any scientific publications focusing on a detailed forensic analysis of BTRFS or ReFS. For BTRFS, a forensic toolkit based on TSK commands has been published by Shujian Yang (Yang, 2016). Unfortunately, this implementation is not capable of handling multiple disks in BTRFS. Thus, it lacks the previously described pool analysis step, which is required for a forensic analysis of pooled storage file systems. For ReFS, only non-scientific descriptions of its layout and an exemplary forensic analysis could be found (Head, 2015; Ballenthin, 2013).

This lack of scientific publications and tools for a forensic analysis of pooled storage file systems emphasizes the relevance of our extension of the standard model for file system forensic analysis to provide a basis for further research and tool development.

Conclusion and future research

By extending TSK and its underlying model we enable the analysis of a whole new class of file systems using this popular toolkit. The proliferation of pooled storage file systems such as ZFS, BTRFS, and ReFS suggests that this file system type will definitely play a part in forensic investigations today and in the future. Just like other researchers have already pointed out, we think that newer file systems like the ones mentioned did not receive the attention they deserve from a forensic point of view yet. We are confident that our work is a valuable step to close this gap.

Moreover, we hope to foster more research in this area. One natural next step would be to analyze BTRFS and ReFS to determine what structures are relevant to integrate them into our extended model. Furthermore, there are more file system types which are currently not considered in Carrier's or our extended model, e.g. log-structured file systems like F2FS or NILFS. It is still an open question whether these types can also be integrated into the standard model.

We are also convinced that our toolkit makes future forensic research of ZFS more accessible. Using our implementation already provides parsing capabilities for a lot of the data structures of ZFS, so that researchers can focus on the actual functionality and evaluations.

Last, we also supply the practitioner with a tool to analyze ZFS, including capabilities to use old überblocks for file recovery and the

ability to parse data of an incomplete pool (Hilgert et al., 2017). Again, we hope that our implementation promotes the development of higher level analysis tools for ZFS.

References

Ballenthin, W., 2013. The Microsoft ReFS File System. http://www.williballenthin.com/forensics/refs/index.html.

Beebe, N., 2009. Digital forensic research: the good, the bad and the unaddressed. In: IFIP International Conference on Digital Forensics. Springer, pp. 17–36.

Beebe, N.L., Stacy, S.D., Stuckey, D., 2009. Digital forensic implications of ZFS. Digit. Investig. 6 (Suppl. S99–S107).

Bonwick, J., Ahrens, M., Henson, V., Maybee, M., Shellenbaum, M., 2003. The Zettabyte file system. In: Proceedings of the 2nd Usenix Conference on File and Storage Technologies.

Bruning, M., 2008. ZFS On-Disk Data Walk (Or: Where's My Data). http://www.osdevcon.org/2008/files/osdevcon2008-max.pdf.

Buchholz, F., Falk, C., 2005. Design and implementation of Zeitline: a forensic timeline editor. In: Proceedings of the Digital Forensics Research Workshop (DFRWS).

Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Professional.

Carrier, B., 2017a. Autopsy. https://www.sleuthkit.org/autopsy/.

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

Garfinkel, S.L., 2009. Automating disk forensic processing with SleuthKit, XML and Python. In: Proceedings of the 2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering. IEEE Computer Society, Washington, DC, USA, pp. 73–84. http://dx.doi.org/10.1109/SADFE.2009.12.

Head, A., 2015. Forensic Investigation of Microsoft's Resilient File System (ReFS). http://resilientfilesystem.co.uk/.

Hilgert, J.N., Lambertz, M., Carrier, B., 2017. The Sleuth Kit with Support for Pooled Storage. https://github.com/fkie-cad/sleuthkit.

iXsystems, Inc, 2017. FreeNAS Storage Operating System. http://www.freenas.org/.

Leigh, D., 2015. Forensic Timeline Analysis of the Zettabyte File System.

Leventhal, A., 2010. What Is RAID-Z? https://blogs.oracle.com/ahl/entry/what_is_raid_z.

Li, A., 2009. Zettabyte File System Autopsy: Digital Crime Scene Investigation for Zettabyte File System.

openZFS, 2016. Companies – OpenZFS. http://open-zfs.org/wiki/Companies.

Sun Microsystems, Inc, 2006. ZFS On-Disk Specification – Draft. http://www.giis.co.in/Zfs_ondiskformat.pdf.

The FreeBSD Documentation Project, 2017. ZFS Features and Terminology. https://www.freebsd.org/doc/handbook/zfs-term.html.

Yang, S., 2016. btrForensics. https://github.com/shujianyang/btrForensics.11.