UNIVERSITÄT BONN

# Contemporary File System Forensic Analysis

Dissertation

zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Jan-Niclas Hilgert
aus
Bonn

Bonn, 2025

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät
der Rheinischen Friedrich-Wilhelms-Universität Bonn

| | |
|---|---|
| Gutachter/Betreuer: | Prof. Dr. Peter Martini<br>Universität Bonn |
| Gutachter: | Prof Dr. Elmar Padilla<br>Hochschule Bonn-Rhein-Sieg |

| | |
|---|---|
| Tag der Promotion: | 17. Juni 2025 |
| Erscheinungsjahr: | 2025 |

# Abstract

This work bridges the gap between traditional and contemporary file system forensic analysis by addressing the limitations of Brian Carrier's foundational 2005 workflow for file system analysis [Car05]. While Carrier's model has remained the de facto standard for nearly two decades, it has neither been updated nor its applicability evaluated for file systems commonly used today, such as ZFS, BTRFS, and MooseFS. These modern file systems introduce complexities — such as pooled storage, the concept of stacked file systems, and network-enhanced functionality — that are beyond the scope of the original workflow and forensic tools like The Sleuth Kit.

To address these shortcomings, this research proposes an extended forensic workflow by introducing two new analysis steps: pool analysis and stacked file system analysis. Pool analysis enables the reconstruction and forensic examination of pooled storage file systems, while stacked file system analysis provides a framework for analyzing file systems that store their data on an underlying file system.

Furthermore, this work explores the integration of network analysis to enhance file system forensics, leveraging network protocols like SMB to reconstruct file systems and user activities from network traffic. Tools such as pcapFS and SMB Command Fingerprinting (SCF) are developed and implemented, offering novel capabilities to recover historical file versions or reconstruct user interactions.

Our findings establish the limitations of Carrier's workflow in the context of contemporary file system analysis and demonstrate the efficacy of our extended model. Thus, our work equips digital forensic analysts with the methods and tools necessary to address contemporary file systems and their challenges as well as leverage the unique features they provide.

# Contents

# 1 Publications

The research presented in this thesis was published in the following peer-reviewed conference proceedings:

- Jan-Niclas Hilgert, Martin Lambertz, and Daniel Plohmann. **Extending The Sleuth Kit and its underlying model for pooled storage file system forensic analysis.** In *Proceedings of the Seventeenth Annual DFRWS USA*, 2017. Best Paper Award [HLP17]. DOI: 10.1016/j.diin.2017.06.003
- Jan-Niclas Hilgert, Martin Lambertz, and Shujian Yang. **Forensic analysis of multiple device BTRFS configurations using The Sleuth Kit.** In *Proceedings of the Eighteenth Annual DFRWS USA*, 2018. Best Paper Award [HLY18]. DOI: 10.1016/j.diin.2018.04.020
- Jan-Niclas Hilgert, Martin Lambertz, and Daniel Baier. **Forensic implications of stacked file systems.** In *Selected Papers from the 11th Annual Digital Forensics Research Conference Europe*, 2024. [HLB24]. DOI: 10.1016/j.fsidi.2023.301678
- Jan-Niclas Hilgert, Axel Mahr, and Martin Lambertz. **Mount SMB.pcap: Reconstructing file systems and file operations from network traffic.** In *Selected Papers from the 4th Annual Digital Forensics Research Conference APAC*, 2024. [HML24]. DOI: 10.1016/j.fsidi.2024.301807

# 2 Introduction

During the inaugural Digital Forensic Research Conference (DFRWS) in 2001 the concept of digital forensic science was defined as *"the use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation, and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations"* [Pal+01]. This definition highlights the wide range of tasks in digital forensics, which also vary depending on the type of evidence examined. Consequently, digital forensics is divided into specialized subdomains dedicated to specific types of evidence, such as storage, memory, and network traffic. Storage forensics, for example, focuses on analyzing data stored on devices such as HDDs, SSDs, and USB drives. This area of digital forensics encompasses essential tasks such as device acquisition, as well as the crucial process of data extraction and recovery from these storage devices. The analysis of file systems, which are engineered to efficiently manage read and write operations on persistent storage, is a critical component of this process.

One possibility for file system analysis is to simply mount the file system using existing implementations, such as those provided by an operating system. This method allows an analyst to access a logical version of the file system, which represents its most current state and consequently overlooks elements such as deleted files or unallocated space. To overcome this limitation and achieve a more comprehensive and forensically sound analysis, a more detailed approach to file system examination is necessary. In 2005, Brian Carrier established foundational work in this area with his book "File System Forensic Analysis" and defined the following fundamental goals for file system analysis [Car05]:

- Depicting the current state of a file system
- Implementing methods for the recovery of deleted files
- Viewing of specific data units within the file system
- Extraction of metadata, e.g. timestamps

- Detection of unallocated areas for further analysis
- Analysis of more advanced features such as journals
- Dealing with corrupted file systems

Carrier's work provides both theoretical and practical insights into handling file systems during forensic investigations. Conducting a comprehensive file system analysis to achieve the aforementioned goals typically requires a thorough understanding of the specific structures within the file system to accurately extract, parse, and interpret its data. To facilitate this process, Carrier organizes the data used by file systems into five distinct categories:

- **Content category:** This includes the actual content of the files stored. File systems typically partition the available storage space into data units, such as blocks in Ext file systems or clusters in NTFS or FAT.
- **Metadata category:** Data that describes a file or directory falls into this category. Among other things, this can include timestamps, attributes, or flags. Furthermore, it also contains the reference to the actual content data of a file or directory.
- **File name category:** Also known as the *human interface category*, data in this category includes the names for files and directories including the hierarchy of the file system. Thus, this data usually references data from the metadata category.
- **Application category:** This category comprises additional data responsible for features of the file system that are, however, not essential for its operation. An example includes journaling or quota.
- **File system category:** This is the core data of the file system, describing the locations of its other structures and relevant areas.

Dividing file system data into these categories allows for a structured analysis of file systems and a transparent development of techniques and tools. Although Carrier's work primarily focuses on file system analysis, it also addresses other vital aspects of storage forensics. For this, he outlines a typical workflow for storage forensics that spans from the acquisition of the storage device to the final analysis of the extracted files at the application level, as depicted in Figure 1:

1. **Physical media analysis:** This initial phase involves acquiring the actual persistent storage device, such as an HDD, SSD, or SD-Card, without interpreting its data, resulting in a raw binary blob. Depending on the configuration of the analyzed system, this step may require several acquisitions of multiple storage devices.

2. **Volume analysis:** In the next step, the binary blob is examined to identify any existing volume configurations. Brian Carrier defines a volume as a *"collection of addressable sectors that an operating system or application can use for data storage"*. By this definition, partitions, which are commonly used to split up available data storage, are also classified as volumes. Consequently, the volume analysis step involves identifying existing partition schemes that divide the storage device, with the most notable examples being the Master Boot Record (MBR) and the GUID Partition Table (GPT). In addition to partitioning available storage space, it is also possible to merge multiple volumes into a single one, often for redundancy or efficiency purposes. For this reason, volume analysis also includes the detection and analysis of multiple disk volumes such as RAIDs. At the end of this phase, the volume layout is known and each, possibly reassembled, volume can be analyzed separately.
3. **File system analysis:** In this phase, the file system stored within a volume is further analyzed. First and foremost, this includes the identification of a given file system within a volume. Following this, rather than simply mounting a file system using operating system drivers, the analysis should ideally include a forensically sound implementation that is capable of parsing and extracting relevant file system structures. This enables an analyst not only to list, browse and extract the current state of the file system, but it also facilitates the analysis of unallocated areas and, where possible, the recovery of deleted files. These resulting files are then analyzed in the final phase of the workflow.
4. **Application analysis:** This step focuses on analyzing data at the application level, which involves interpreting files according to their specific file types. Hence, the analysis is heavily influenced by the file format in question. Common tasks include analyzing binary files for malicious behavior or extracting and recovering data from databases such as SQLite.

Following this workflow from start to end takes an analyst from the raw storage device to the artifacts obtained from analyzing data at the application level. As shown in Figure 1, all the steps in this workflow must be performed sequentially since the output of each step is used as input for the subsequent one. A significant advantage of this approach is that each step can be performed independently of the others. For example, physical media analysis is not influenced by the type of volume, file system, or specific file types contained within it, consistently producing one or more binary data blobs. These blobs are then analyzed to determine the volume layout, potentially identifying partitions or reassembled RAIDs independent of both the data's origin and the file systems eventually stored within these volumes. This principle of independence extends to the file system analysis step, where each detected volume is examined

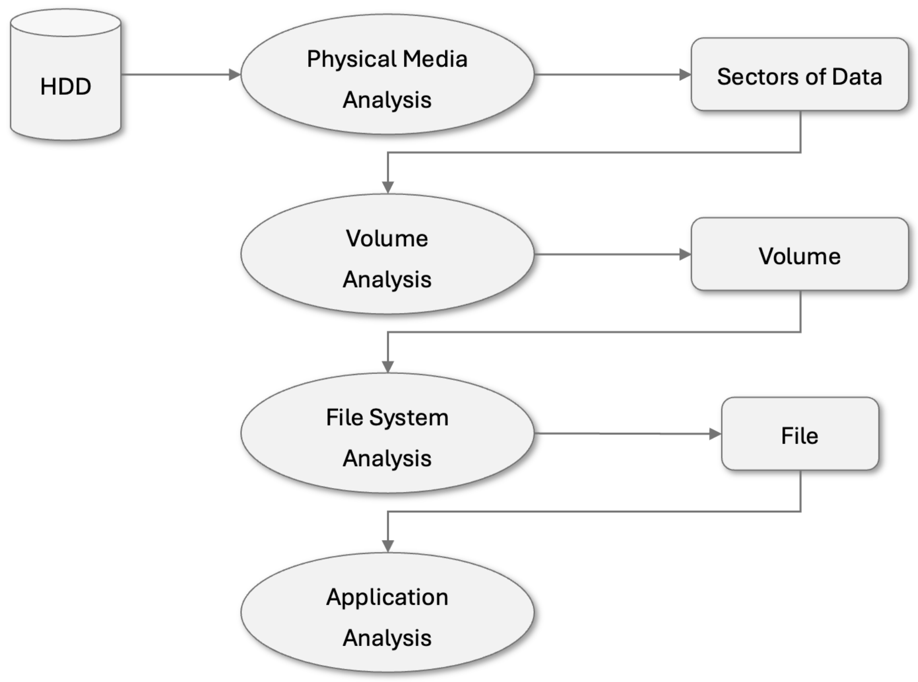


**Figure 1.:** *Workflow for a file system forensic analysis established by Brian Carrier in 2005 [Car05].*

for the presence of a file system, regardless of the type of volume or underlying storage device. Finally, in the last step, the resulting files are examined, and here too, the origin of the data does not affect the analysis.

This structured design of the analysis workflow ensures that each step has clear, distinct goals and maintains transparency throughout the process. Furthermore, this approach also facilitates the development of methods and tools for each of these steps independently from the other. For example, a tool designed for FAT analysis can focus solely on its task without having to address the complexities of volume reconstruction. Leveraging this approach, Brian Carrier developed multiple specialized command-line tools, which he compiled into a comprehensive toolkit known as *The Sleuth Kit*.

Although The Sleuth Kit is well established within the digital forensics community and has undergone various updates over the years, its foundational model has remained unchanged since its introduction in 2005. We define *contemporary file systems* as those commonly used today, nearly 20 years after Carrier's initial work. *Contemporary file system analysis*, therefore, refers to the examination of these modern file systems. While Carrier's work covered file systems still in use today, such as FAT, NTFS, and the ext family, newer file systems like

BTRFS, ZFS and MooseFS have since emerged. Additionally, concepts like the one of stacked file systems, which existed when Carrier published his work, were not covered by him, but still remain relevant today. Consequently, analysts working with these contemporary and possibly underexplored file systems may face challenges or dead ends, when established forensic tools like The Sleuth Kit may either lack support for these systems or fail to analyze their features effectively. This issue may extend beyond mere implementation gaps, as the underlying workflow of many forensic tools may not be fully equipped to handle contemporary file systems, potentially suggesting the need for a new theoretical framework.

In light of this, this work aims to bridge the gap between file system forensic analysis established in 2005 and contemporary file system analysis required today, ensuring that forensic analysts are equipped with the knowledge, methods, and tools necessary to effectively handle modern challenges in file system analysis and take advantage of the features of contemporary file systems during this process.

We begin by identifying examples of modern file systems whose concepts and features deviate from Carrier's original workflow. This includes not only addressing recent developments in file systems, but also incorporating file systems that, while possibly existing at the time, were not part of Carrier's work. We then reassess the applicability of Carrier's workflow to these contemporary file systems, identifying its limitations and gaps. From this analysis, we propose necessary extensions to bridge these gaps to enable contemporary file system analysis. Although Brian Carrier's original workflow includes four steps, his work places particular emphasis on the volume and file system analysis phases. Consequently, our work will focus closely on these two phases, with special attention to their interconnection. In addition to the mere accommodation of these contemporary file systems in an established workflow, this work will also highlight their unique characteristics in a forensic analysis and further explore how they can present new opportunities to enhance forensic capabilities, offering greater value to forensic analysts.

In summary, the research questions of our work are as follows.

- **Research Question 0 ($RQ_0$)**: Which concepts of contemporary file systems are not considered in Brian Carrier's foundational work from 2005?
- **Research Question 1 ($RQ_1$)**: How applicable is Brian Carrier's 2005 workflow for analyzing contemporary file systems?
- **Research Question 2 ($RQ_2$)**: How can Carrier's workflow be modified or extended to better support the analysis of contemporary file systems?

- **Research Question 3 ($RQ_3$):** What features or characteristics of contemporary file systems offer added value or require special consideration during forensic investigations?

## 2.1 Pooled Storage File Systems

As outlined earlier, the analysis of a file system is described as a step independent of the underlying volume. The volume is merely serving as a container for storing the data of the file system. Although a multiple disk configuration like a RAID may distribute the actual data across multiple disks, the analysis of the file system itself is unaffected. This is because the RAID is already reassembled in the volume analysis step and is subsequently treated as a single volume that contains the file system during further analysis. Fundamentally, Brian Carrier's workflow is based on the assumption that a file system is assigned to a single volume, thus bypassing the need for advanced volume management features, such as data distribution across multiple disks, within the file system itself.

Over the years, this "one-to-one" association between a volume and a file system has been challenged by file systems that were not considered by Brian Carrier in 2005, providing an initial answer to $RQ_0$. First and foremost among these are contemporary file systems like ZFS, the default file system for modern FreeBSD operating systems, and BTRFS, which has been integrated into the Linux Kernel. These file systems incorporate their own volume management capabilities, enabling the creation of complex multiple volume configurations such as mirrors or RAIDs of various levels. Because these volumes are combined by the file system to create a unified pool of available storage, we introduce the term *pooled storage file system* to describe any file system that implements its own volume management functionality. Consequently, we refer to the volumes that it uses for data storage as its *pool members*. To create a storage pool, pooled storage file systems allow users to specify one or more volumes across which data is distributed and potentially transformed, depending on the pool configuration. These volumes can include any type described by Brian Carrier, such as partitions, individual disks, or even another RAID volume composed of multiple underlying volumes.

Hence, the reconstruction of a pooled storage file system requires access to all members of the pool across which it is distributed. While this concept is similar to traditional file systems stored on multiple disk configurations such as RAIDs, the key difference lies in the fact that the file system itself manages the distribution and transformation of its data across its corresponding pool members. Brian Carrier's traditional workflow assumes that these processes are handled by an independent implementation, such as a software or hardware

RAID, which could be analyzed separately before proceeding to the file system analysis. However, pooled storage file systems break this assumption by implementing their own volume management. As a result, the standard workflow for file system analysis is inadequate for pooled storage systems and must be adapted to accommodate these complexities. In response to *$RQ_1$*, we evaluate the applicability of Carrier's four-step workflow to pooled storage file systems in detail in our first publication [HLP17], and also address *$RQ_2$* by identifying and proposing the necessary modifications to support pooled storage file systems.

The initial step, the physical media analysis, involves the acquisition of a storage device. Since this step deals with data as a sequence of bytes, independent of file system or pooled storage concepts, it remains unchanged in an adapted workflow. Similarly, the final step, application analysis, involves working with previously extracted and recovered files at the application level. Since this step is entirely independent of the underlying file system, it does not require any modifications and also remains unchanged.

Since pooled storage file systems incorporate their own volume management capabilities, they can operate directly on a disk, making the volume analysis step seemingly unnecessary. However, this scenario is similar to that of a traditional file system that is stored directly on a disk. Even though such situations were possible with the file systems Brian Carrier considered, he included the volume analysis step in his workflow without providing an option to bypass it. In these instances, the volume analysis simply identifies a single volume that corresponds to the entire disk. Additionally, pooled storage file systems may not always operate directly on a disk, instead using other volumes created by common volume managers. For these reasons, the volume analysis step remains an essential part of the workflow, following the physical media analysis.

For the file system analysis step, it is still possible to mount the pooled storage file system using existing implementations. However, this approach suffers from the same limitations as when used with traditional file systems. To conduct a thorough forensic investigation, a file system-specific implementation is required. Thus, the core tasks of the file system analysis remain the same, with the outcome still being a collection of files extracted from the file system which are then passed on to the application analysis. Consequently, in our adapted version of Brian Carrier's model, the file system analysis step and its output remain unchanged.

The key difference in the analysis workflow lies between the volume and file system analysis steps, as the file system analysis no longer necessarily expects a single volume as its input. Instead, due to the volume management capabilities of pooled storage file systems, multiple volumes, or pool members, may be required as input. These pool members form the storage pool, which ultimately contains the file system's data. Unlike a RAID configuration created by an independent implementation, however, it is no longer feasible to separately reassemble the multiple volumes before performing the file system analysis. The pooled storage file system itself can be used to reconstruct the storage pool, similar to how a traditional file system is mounted. To conduct a thorough file system analysis, direct access to all data within the storage pool is required. Therefore, it becomes necessary to reconstruct the pool during the analysis step, which includes tasks such as mapping file system addresses to their precise locations on the pool members. In summary, the tasks we identified in relation to pooled storage file systems include these critical steps:

- Identify when a volume is part of a pooled storage file system.
- Extract detailed information about the pool's configuration, offering valuable insights to analysts, particularly regarding any missing pool members.
- Perform accurate translation of file system addresses to their corresponding locations on the pool members, such as the exact pool member and offset. This involves an implementation of the file system-specific methods used to distribute and transform its data.
- Ensure access to all structural data within the pooled storage file system, including metadata and unallocated areas on the pool members.
- Provide the possibility to deal with incomplete storage pools, such as those with missing members, to extract as much data as possible whenever feasible.

We refer to the part of the analysis responsible for addressing these tasks as the pool analysis. Since pooled storage file systems handle the distribution and transformation of their data themselves, the corresponding pool analysis is also highly file system dependent. Thus, one approach is to integrate the pool analysis directly within the file system analysis step outlined by Brian Carrier. However, we have chosen to incorporate the pool analysis as a distinct step rather than embedding it within the file system analysis, which offers significant advantages. First, it maintains the functional segregation of each step, as initially intended by Carrier. This allows each step to specialize and focus on its specific role — volume analysis on recognizing volume configurations, pool analysis on identifying pool members and providing access to them, and file system analysis on examining the file system leveraging the access provided by the pool analysis. Secondly, maintaining a separate pool analysis step from the file system analysis improves the extensibility for methods dealing with storage pools. For instance, when new volume management methods are introduced or changed within a pooled storage file system, they can be integrated or updated more smoothly as it allows for these updates and enhancements without any modifications to the fundamental file system analysis.

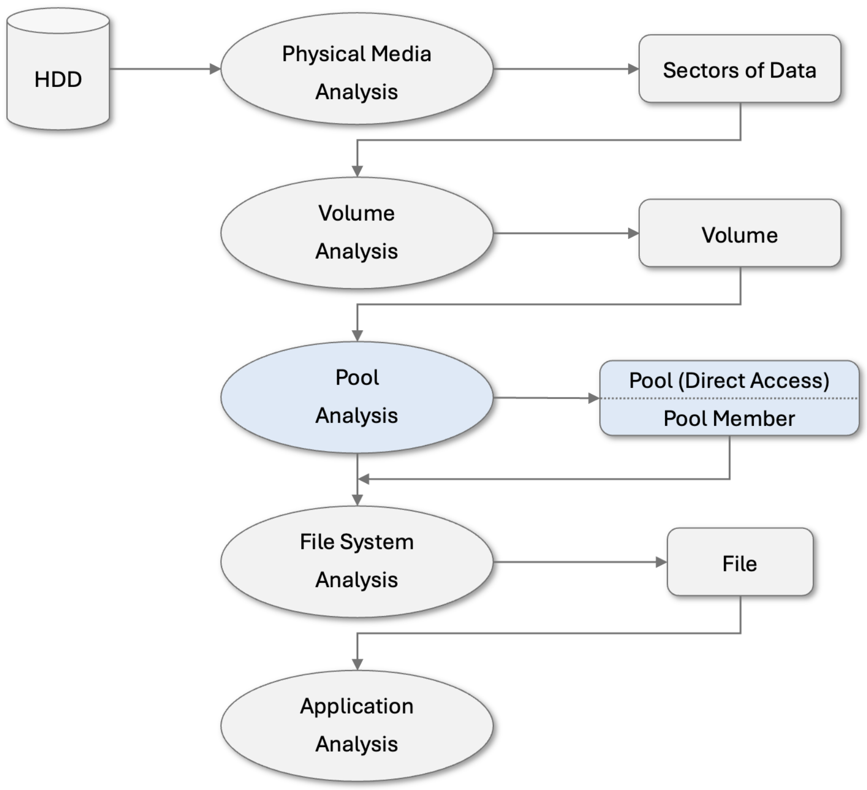


**Figure 2.:** *Our extension to Carrier's workflow for a file system forensic analysis supporting pooled storage file systems [HLP17].*

Figure 2 illustrates our extended model for file system forensic analysis, incorporating the newly added pool analysis step. Traditional volume analysis continues to manage the examination of standard volume configurations, such as partitions, RAIDs, and other logical volumes, which then serve as input for the pool analysis. If volumes are not part of a pool, they are addressed directly in the file system analysis step. During the pool analysis step, individual pool members are identified within a storage pool and any relevant information about themselves, other members, and the pool as a whole is extracted. Additionally, this step can reconstruct the pool to provide direct access, allowing the mapping of addresses to the corresponding locations on the pool members and any necessary data transformation along the way. This direct access enables subsequent file system analysis to independently and thoroughly examine the entire pooled storage file system.

It is also important to recognize that the analysis process may not always follow a strictly linear path. For example, pooled storage file systems may allow users to create logical volumes from the pool's storage, which then loop back into the volume analysis step. However, in line with Carrier's original workflow, which also omitted scenarios where file systems are stored within files, we have chosen to exclude such recursive scenarios for storage pools within our extended model as well.

Our extended model, addressing $RQ_2$, demonstrates how Carrier's workflow can be adapted to support the analysis of pooled storage file systems. Beyond this theoretical framework, we have also implemented both the pool analysis and subsequent file system analysis steps for ZFS, in line with our extended model, demonstrating its practical applicability. We extended The Sleuth Kit, maintaining consistent command names to ensure ease of use and transparency for analysts. Additionally, we utilized our implementation to demonstrate the feasibility of a forensic analysis of ZFS across various pool configurations. This included scenarios where the default file system implementation considered pools incomplete and, as a result, was unable to mount the file system. Despite this limitation, our approach and implementation successfully extracted data from the remaining pool members, highlighting both the importance of a comprehensive forensic analysis for pooled storage systems as well as the effectiveness of our approach.

Our second publication [HLY18] builds upon our extended workflow and examines its application to another pooled storage file system by adding support for the pool analysis step of BTRFS, thereby enabling the analysis of multiple disk BTRFS configurations. We also conduct a practical forensic analysis of this pooled storage file system to demonstrate our model's effectiveness and illustrate how BTRFS' features can be leveraged during forensic investigations, thus providing answers to $RQ_3$. In our work we demonstrate that the file system's snapshot functionality is an important consideration during forensic analysis, since this feature allows users to arbitrarily capture the state of the file system at a specific point in time. Our implementation is capable of listing and analyzing these snapshots, enabling the extraction of older versions of the file system. Additionally, we explore the potential to recover previous versions by leveraging the copy-on-write mechanism, which writes new metadata and content to new locations rather than overwriting existing data. This process creates valuable artifacts that can be used to trace back through earlier versions of the file system. However, this approach is inherently more error-prone, as the file system may not maintain these structures in a consistent state, unlike snapshots, which are specifically designed for this purpose.

## 2.2 Stacked File Systems

As outlined in our first two publications, pooled storage file systems create their own storage pools by distributing and transforming data across one or more volumes, necessitating an additional step before conducting the actual file system analysis. Despite integrating volume management with the file system, a pooled storage file system still relies on an underlying volume to store its data. In contrast, the class of *stacked file systems* employs a different approach to data storage. Instead of writing data directly to a volume, stacked file systems store their data as files within another underlying file system. This method allows for the development of novel file systems built on top of well-established, robust file systems that already offer a distinct set of features.

Thus, since the introduction of this concept in 1994 [HP94], stacked file systems have become a significant area of research, particularly in the development of file systems. A stacked file system, which we also refer to as the *upper file system*, adds an additional layer on top of the underlying or *lower file system*, offering users a variety of enhanced features. An early notable example is eCryptfs, which facilitates the creation of an encrypted file system. eCryptfs leverages the lower file system to store encrypted versions of its files. When the upper file system is mounted with the correct decryption key, it presents the plain, unencrypted versions of the files to the user. Beyond just data transformation, stacked file systems can also distribute their data across multiple lower file systems. This approach is widely utilized in contemporary distributed file systems such as MooseFS or GlusterFS, which are spread across multiple servers. Although stacked file systems had existed for some time when Brian Carrier published his work, they represent a class of contemporary file systems that were not considered and remain unsupported by common forensic tools like The Sleuth Kit, offering an additional answer to $RQ_0$. To address this gap, our third publication [HLB24] explores the integration of stacked file systems into a forensic workflow, providing further insights to $RQ_1$.

By implementing their own methods to distribute and transform data, stacked file systems share some similarities with the concept of pooled storage file systems discussed earlier. Furthermore, stacked file systems can be deployed across multiple lower file systems and, consequently, also across multiple volumes. However, a key difference lies in their dependencies: while a pooled storage file system requires an underlying volume to build its storage pool, a stacked file system is entirely dependent on an underlying file system and cannot be directly deployed onto one or multiple volumes. As a result, the analysis of a stacked file system always necessitates a preceding file system analysis of the corresponding lower file system and must thus also be performed after the pool analysis step.

For these reasons, even our extended workflow, in its current state, does not fully account for the analysis of stacked file systems. While mounting a stacked file system can provide access to its data, it shares the same limitations as mounting a traditional or pooled storage file system resulting in the loss of valuable forensic information. In line with $RQ_2$ and to enable a comprehensive forensic analysis of stacked file systems, we have identified and defined six specific requirements for a stacked file system analysis that must be addressed by an extended model capable of accommodating them:

1. **Detection of Stacked File Systems:** Recognizing and identifying stacked file systems is crucial, as they can easily be overlooked if investigators are unaware of their existence or if their tools lack compatibility. This is further complicated by the fact that a file system can simultaneously function as both a lower and regular file system, potentially obscuring its role as the lower system. For this reason, files and directories extracted during the file system analysis should always be analyzed for unique characteristics hinting at the presence of an upper file system. This step also involves extracting information about any additional lower file systems utilized by the upper file system, for example when data is distributed across multiple servers.
2. **Correlation of File Names:** Establishing the relationship between upper and lower files is essential for accurately reconstructing and extracting the original hierarchy of the stacked file system.
3. **Data Reconstruction:** Similar to pooled storage file systems, stacked file systems may distribute and transform data when storing it across one or more lower file systems. To effectively analyze a stacked file system, it is crucial to understand the methods and algorithms it employs for these processes. This includes addressing the issue of fragmentation, where an upper file is spread across multiple lower files, and reassembling it in the correct order. Additionally, any transformation applied by the stacked file system to store its data in the lower files must be reversed to retrieve the original version of the upper file.
4. **Role of Timestamps:** During stacked file system analysis, it is important to extract timestamps stored by the upper file system. However, the presence of a lower file system introduces an additional source of timestamps that should also be considered. Since each lower file is part of the lower file system, their timestamps may provide valuable information. Especially when data is fragmented across multiple lower files, their timestamps may be used to obtain a more detailed timeline of file changes.

5. **Slack Space in Stacked Systems:** Stacked file systems introduce new complexities in slack space, where lower files may contain padding that does not impact the upper file system and can thus be used to hide data.
6. **File Recovery Methods:** For file recovery, the analysis should take advantage of any features provided by the upper file system, such as built-in trash bin implementations that offer additional recovery options. However, it is also crucial to explore the file recovery opportunities offered by the underlying lower file system.

Performing the aforementioned steps in a stacked file system analysis requires the prior extraction of the corresponding lower files used by the stacked file system, as these files serve as the input for this analysis phase. Consequently, it is not feasible to incorporate the stacked file system analysis within the traditional file system analysis step, as doing so would introduce unwanted recursion in the workflow. Therefore, we have extended Brian Carrier's workflow by adding a dedicated step specialized for stacked file system analysis, which follows the traditional file system analysis as depicted in Figure 3. We refer to this new step simply as the *stacked file system analysis*.

In addition to proposing a theoretical extension to the workflow for accommodating stacked file system analysis, we also provide practical insights by examining the previously outlined tasks for stacked file system analysis using eCryptfs, MooseFS, and GlusterFS as case studies while demonstrating how these tasks can be effectively executed. Our results directly address $RQ_3$ and highlight the additional value of a dedicated stacked file system analysis in forensic investigations, including information such as a more detailed timeline of file changes and additional recovery options for deleted files. By expanding Brian Carrier's workflow for file system forensic analysis by two additional steps, it can now be applied to both pooled storage as well as stacked file systems.

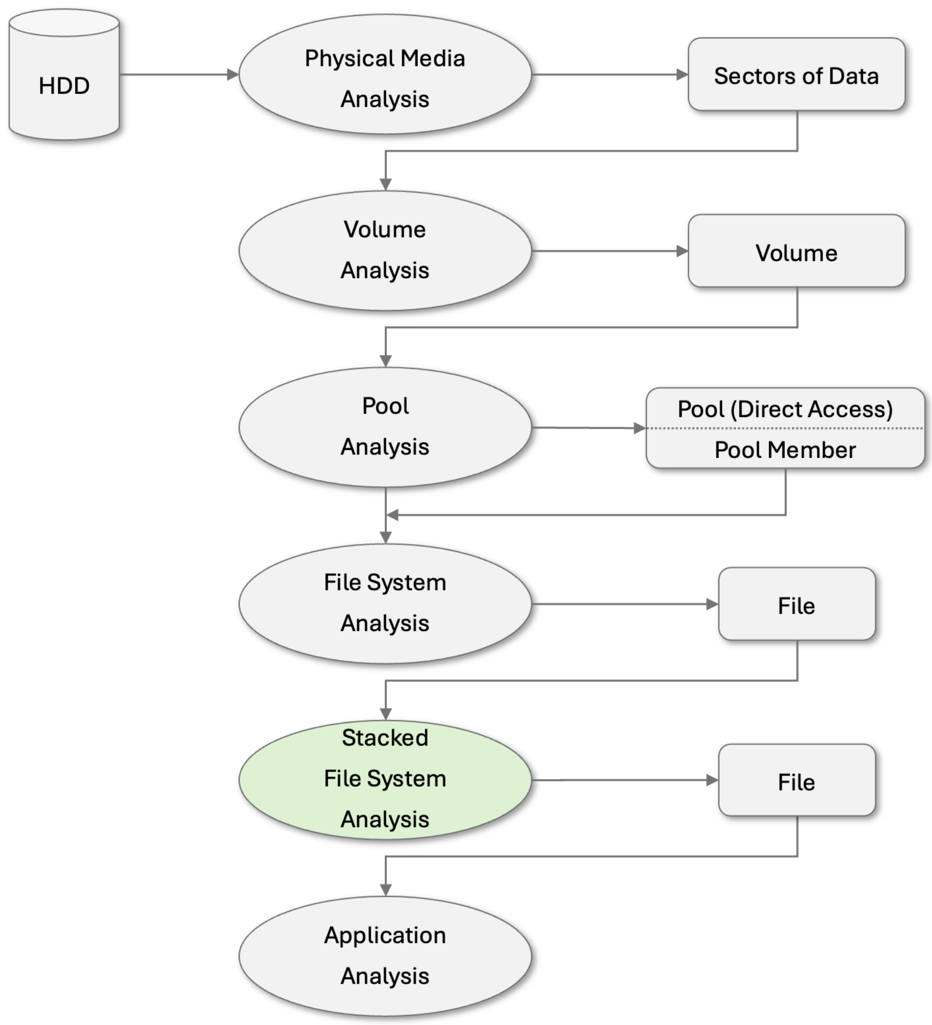


Figure 3.: *Our extensions to Carrier's workflow for a file system forensic analysis supporting pooled storage as well as stacked file systems [HLB24].*

## 2.3 Network-enhanced File System Analysis

Beyond his workflow for file system analysis, Brian Carrier outlined several key analysis types within the broader scope of digital forensics, as illustrated in Figure 4. Although his work focuses on the analysis of persistent storage, including both volume and file system analysis, Carrier briefly mentions memory and network analysis as other possible analysis types. However, these areas are treated separately from file system analysis, and thus, the potential for leveraging them to enhance file system forensics remains unexplored in his work. Memory analysis, for example, has proven to be a rich source of artifacts in forensic investigations, allowing for the extraction of key material, deleted data, and other volatile structures. It has also seen specific applications in file system forensics, with Volatility plugins like `mftscan` capable of retrieving NTFS structures directly from memory. In contrast, network traffic has received little attention in the context of file system forensics, despite its strong relevance to contemporary file systems.

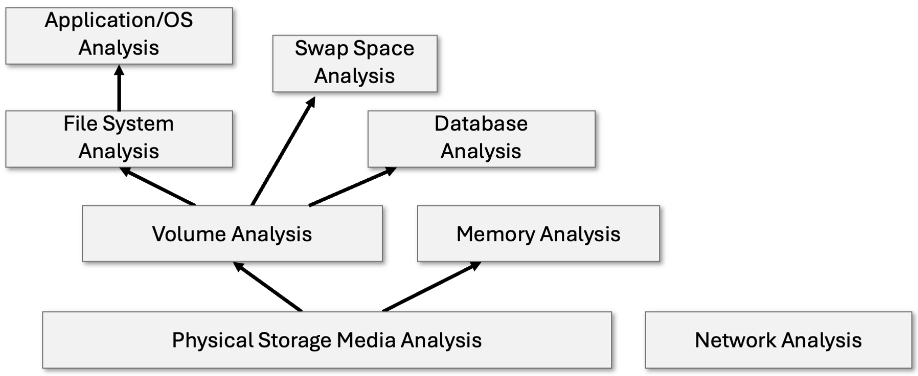


Figure 4.: *Illustration of different analysis types by Brian Carrier [Car05].*

Firstly, some file systems require network functionality to operate effectively. As previously discussed, pooled and stacked file systems can both distribute and transform data across various locations. While pooled storage file systems usually operate on a single local machine, many modern stacked file systems distribute data across multiple underlying file systems hosted on different servers. Known as *distributed file systems*, this class of file systems is defined by the distribution of data across multiple entities. To enable reading and writing to these remote components, such file systems must support or integrate network functionality. Examples include MooseFS, GlusterFS, and the Hadoop Distributed File System.

Moreover, file systems today are often accessed remotely rather than solely on local machines, for instance, when a corporate share is mounted on a remote client. Although some file systems rely on proprietary network protocols for remote access, more commonly standardized protocols are used for this purpose. Notable examples include the Network File System (NFS) and Server Message Block (SMB) protocols, which are widely adopted in Linux and Windows environments, respectively.

As a result, today's network traffic frequently contains data originating from or related to file systems. From a forensic perspective, this network data can provide valuable information beyond what is typically obtained through traditional file system analysis. For example, network captures preserve a continuous stream of data over time, whereas file systems generally only reflect their most recent state, making it difficult to recover older data. Consequently, network traffic may contain information that cannot be recovered using standard file system forensics.

As mentioned before, Brian Carrier barely touches upon the analysis of network traffic in his 2005 work, merely suggesting its use for correlational purposes, such as confirming the timing of file activities. Thus, he maintains a clear distinction between traditional storage forensics and network forensics. Consequently, in addressing $RQ_0$, Carrier's work for file system analysis does not explore the analysis of network traffic generated due to the remote access to file systems or the architecture of contemporary file systems themselves.

To address this gap, our fourth publication [HML24] centers on $RQ_3$, exploring how file system-related network traffic can bridge the divide between network and storage forensics and enhance traditional file system analysis. As an initial case study, we focus on the well-established SMB protocol, which is widely used in Windows environments to provide remote access to a server's file system or shared resources across multiple clients.

As a first step, we explore how network traffic containing file system-related data can be leveraged to reconstruct the corresponding file system. This approach not only provides additional information that may be inaccessible through traditional file system analysis but can also prove valuable in scenarios, where the actual file system is unavailable. Such situations may arise due to limited access or because the file system has been deleted or corrupted. To reconstruct a file system from network traffic, we follow the five file system data categories introduced by Brian Carrier and discuss how each category's data can be extracted from network traffic:

- **Content category:** Extracting the actual content of a file from network traffic relies on the file being transferred at some point during the network capture. This typically occurs during read or write operations involving the file. However, depending on the implementation, these operations may only transfer parts of the file, thus allowing for only partial reconstruction of its content. On the other hand, the initial distribution of a file to its final remote locations naturally involves its entire content.
- **Metadata category:** This category contains all available metadata for a file, including timestamps or attributes. Unlike file content, metadata can be transferred without an explicit read or write operation, such as when browsing directories on a remote file share. To accurately reconstruct the file system, it is essential to match this metadata with its corresponding content, if available.

- **File name category:** Similarly, file names can also be transmitted without explicit read or write operations. Extracted file names from the network capture must be correctly associated with their corresponding metadata and content. Furthermore, it is essential to extract information that allows for the most comprehensive reconstruction of the file system hierarchy.
- **Application and File System Categories:** Data from both the application and file system categories is typically not necessary for network access to a file system, which explains their rare presence in network traffic. The file system category data, which outlines the layout and structure of the file system, is crucial for understanding its architecture, but is not essential when reconstructing a functional, though not exact, replica of the original system. Brian Carrier also considers data from the application category to be non-essential for the basic operations of the file system. Consequently, omitting these categories in our reconstruction process does not compromise the forensic value of the reconstructed file system.

Extracting this information from network traffic enables the reconstruction of the original file system, encompassing all available content, metadata, and the hierarchical structure that can be found within the captured data, even if it does not result in a precise copy of the original file system. However, the reconstructed file system provides data that may not be recoverable through traditional file system analysis. Because network traffic captures are continuous, they may include multiple read or write operations for a single file, potentially revealing various versions of that file. During reconstruction, it is essential to account for them and give analysts access to all versions of a file, including those that may no longer exist within the actual file system itself. Additionally, the reconstructed file system can use information from the metadata and file name categories to create files without content, which refer to as *hollow files*. Although these hollow files are empty and lack content, they retain critical metadata, such as timestamps and are also placed in the correct location within the file system hierarchy, providing analysts with as much contextual information as possible.

To demonstrate the feasibility of this approach, we apply it to the SMB protocol and detail how the necessary information for reconstructing the original file system can be extracted from network traffic. In addition, we provide a practical implementation of this theoretical approach through the development of pcapFS. pcapFS is a framework designed to mount network traffic as a file system, allowing analysts to seamlessly browse through the data captured in the network. While it supports various protocols such as HTTP, FTP, and TLS, pcapFS also includes support for the reconstruction of SMB network traffic, enabling the extraction of multiple file versions and the creation of hollow files.

Beyond reconstructing a file system from network traffic, it is valuable to identify the origins of captured file operations, such as file creation, modification, access, or deletion. This level of detail provides information on the origin and sequence of file-related events, helping to build a comprehensive timeline of actions performed on the file system. Although traditional file system analysis may not always provide this information, network traffic can be leveraged to reconstruct these file operations. To demonstrate the practicality of this concept, we developed and implemented SMB Command Fingerprinting (SCF). This technique involves generating hashes for SMB packets based on their distinctive values and combining them into a set of rules, which allows us to accurately reconstruct user interactions, such as browsing a file share or creating files and directories, entirely from network traffic.

While the use of network traffic is not exclusive to contemporary file systems, it plays a crucial role in their design and functionality, particularly in the case of distributed file systems. To harness this feature for forensic purposes, we have introduced, examined, and implemented two novel approaches. Although these methods for reconstructing file systems and file operations can be applied independently, the most comprehensive insights are achieved when they are combined during an analysis. As demonstrated in our publication, the feasibility of these approaches underscores the significant potential of leveraging network traffic to enhance traditional file system forensic analysis.

# 3 Accompanying Text to Hilgert, Lambertz & Plohmann (2017)

The publication **Extending The Sleuth Kit and its underlying model for pooled storage file system forensic analysis** [HLP17] presents a first examination as well as an adoption of Brian Carrier's established workflow for file system analysis. For this purpose, the paper initially summarizes the current state of file system forensics and presents the de-facto standard model by Brian Carrier consisting of the four steps: Physical Media, Volume, File System and Application Analysis. While each step requires the output of the previous one, they are transparent to each other, i.e. a file system analysis performed on a reassembled RAID volume is identical to a file system analysis of an extracted partition. As a result, the file system analysis step is completely independent of the underlying volume layout. This stems from the traditional "one-to-one" association between a volume and a file system, which was the standard for most file systems, when Carrier published his work in 2005. However, this assumption has been challenged by file systems that either emerged in the years since or were not considered by Carrier. A notable example is ZFS, introduced in 2003 but not included in Carrier's original work.

In ZFS, it is possible to combine one or more top-level virtual devices (vdev) to a single zpool that is used as the storage space for the file system. When data is then stored in the zpool ZFS stripes it across all its available top-level vdevs, which in its simplest form can be files or disks, e.g. volumes as defined by Carrier. Furthermore, ZFS also supports advanced types for top-level vdevs such as mirror and raidz. These top-level vdevs contain one or more files or disks and distribute the data to these according to their type. For file systems like ZFS, which manage their own storage space and implement their own volume management features, we introduce the term 'pooled storage file systems.

Although these file systems enable the use of multiple separate storage devices, they break the aforementioned "one-to-one" association between a volume and a file system. In ZFS, the file system itself manages data distribution across multiple disks, rather than relying on an independent implementation, as assumed in Carrier's workflow. Moreover, established forensic tools like The Sleuth Kit lack support for ZFS, creating a significant gap for investigators. To address this gap and assess whether it is merely an implementation issue, our work evaluates the applicability of Carrier's workflow for analyzing pooled storage file systems. We conclude that neither the theoretical foundation nor the tools based on it, such as The Sleuth Kit, are adequately equipped to handle this class of file systems.

To address this, we extend Carrier's model with an additional step between the volume and file system analysis, which we call pool analysis. As with other steps in the model, the input for this phase is the output of the previous one, i.e. any detected volumes. The pool analysis step determines whether any of these volumes are part of a storage pool. If not, they are simply passed to the file system analysis step. This is typically the case for established file systems that do not support storage pools. This approach is similar to scenarios where a file system is stored directly on a disk, bypassing the volume analysis step.

If pool members are detected during this step, it is crucial to extract basic information such as the total number of pool members and the type of pool. The next step is to reconstruct the pool, which requires understanding how and where data is stored. Similar to reassembling a RAID, the pooled storage file system that originally created the pool can be used for reconstruction. However, since the file system is tightly integrated with volume management, the pool cannot be reconstructed independently. This approach yields a reconstructed pool similar to a mounted file system, providing only limited data access for forensic analysis.

To gain full access to all data within the pooled storage, including deleted files and unallocated space, an advanced pool analysis is essential. This step determines how and where data is stored across the previously detected volumes, requiring a deep understanding of the specific mechanisms the pooled file system uses to distribute and transform data across its pool members. Akin to the file system analysis, the pool analysis is highly dependent on the specific pool and thus also file system implementation.

In our work, we demonstrate how the pool analysis step, including both identification and reconstruction, can be applied to ZFS on a theoretical level. To achieve this, we extended The Sleuth Kit (TSK) with a new command, pls, which accepts one or more disks or volumes as input and analyzes the underlying pool configuration, identifying issues such as missing or redundant disks. Additionally, we reimplemented ZFS's volume management capabilities to enable independent, direct access to pool members for a forensically sound analysis of a ZFS pool. To further evaluate our pool analysis implementation, we also developed methods for performing the file system analysis step specifically for ZFS, integrating these into a customized version of TSK. We show that our implementation not only offers basic functions like listing and extracting files, but also supports file recovery in the case of incomplete pools. An incomplete pool, missing one or more top-level vdevs and typically not reconstructable using the original implementation, can still be analyzed using our approach, allowing access to the remaining data of its pool members.

### Individual Contributions

Overall, the individual contributions within this paper include:

- Examining and extending Brian Carrier's model to support the forensic analysis of pooled storage file systems
- Analyzing the pooled storage functionality of ZFS as a prominent example of pooled file systems.
- Implementing the pool analysis and file system analysis steps for ZFS
- Evaluating the feasibility of a forensic analysis of ZFS, including data reconstruction from incomplete storage pools.

# 4 Accompanying Text to Hilgert, Lambertz & Yang (2018)

The publication **Forensic analysis of multiple device BTRFS configurations using The Sleuth Kit** [HLY18] builds upon the previous preliminary work, which extended Brian Carrier's model for file system forensic analysis and added support pooled storage file systems using ZFS as an example. As outlined and motivated in our previous work, the de-facto standard workflow for file system analysis at that time was not suitable to deal with file systems that do not follow the "one-to-one" association between a file system and a volume. Instead, pooled storage file systems implement their own volume management capabilities making it possible to directly create multiple disk configurations for redundancy or efficiency purposes. While this is a huge advantage regarding usability, the analysis workflow as it was presented by Brian Carrier in 2005 had to be updated and extended in order to support pooled storage file systems not only in theory, but also in existing tools.

To address this, we proposed an additional step within the four-step model labelled pool analysis, which follows the volume and precedes the file system analysis. This step is not only responsible for identifying volumes that have been part of a storage pool, but also provides the possibility to reconstruct the pool. Since this part is highly implementation-specific, it is necessary to analyze the structures and methods used by the pooled storage file system to perform the pool analysis. In our previous work, we examined and implemented the pool analysis as well as the subsequent file analysis step for ZFS, demonstrating the applicability of our extended model. Additionally, we performed a forensic analysis of the ZFS file system across various pool configurations. In this work, we shift our attention to BTRFS, another prominent pooled storage file system, to further assess and validate the applicability of our extended model, while also highlighting novel features analysts should be aware of.

BTRFS, introduced in 2007, has since been integrated into the Linux kernel and is now also available for the Windows operating system. Like ZFS, BTRFS includes built-in volume management capabilities, allowing users to combine volumes into various multi-disk configurations, such as mirrors or different RAID levels. In the initial phase of our research, we examined BTRFS's use of a specialized tree structure for volume management, known as the chunk tree. Firstly, the chunk tree contains a device item for each physical device, or volume, that BTRFS uses in its storage pool. These device items store various attributes, such as the device's size and unique identifier. Secondly, the chunk tree holds chunk items. In BTRFS, the available storage is divided into multiple non-overlapping chunks, which are addressed using a continuous logical addressing scheme. This ensures that each logical address is mapped to a specific chunk. Information like the start and length of each chunk is stored in the corresponding chunk item within the chunk tree.

A chunk item also contains details about the stripes into which a chunk is further divided. For each stripe, it stores the device ID of the corresponding pool member and the offset where the data is located. The distribution of data across these stripes depends on the chunk type. In a RAID0 configuration, data is striped across all available stripes within the chunk, with BTRFS using a specific stripe length to define the size of the data blocks written to each stripe. In a RAID1 chunk, data is mirrored across pairs of stripes, with each pair residing on different devices. Understanding these aspects of BTRFS volume management is critical for conducting a thorough pool and file system analysis. To facilitate this, we followed our extended model and implemented the pool analysis step for BTRFS. Thus, the pls command, introduced in our previous work to analyze one or more disks or volumes and identify pool members, was extended to support the detection of BTRFS pool members. Furthermore, we have implemented the required pool analysis steps to allow direct access to a BTRFS storage pool.

Building on this foundation, we have integrated our pool and file system analysis implementations to enable a comprehensive forensic examination of BTRFS using multiple disks, providing analysts with a practical tool for investigating this contemporary file system. We demonstrate the effectiveness of this implementation through the forensic analysis of various BTRFS configurations, highlighting key characteristics and considerations for investigators.

First and foremost, this includes the copy-on-write principle utilized by file systems such as BTRFS. Copy-on-write is a modern concept designed to keep a file system in a consistent state, even after a crash. Instead of overwriting an existing block when modifications are made, the modified block is written to a new location within the file system. Once the write operation is complete, all associated metadata and pointers that reference this block are updated to point to the new location, also using the copy-on-write principle. While this mechanism primarily ensures the file system's consistency, it also generates numerous data and metadata artifacts that can be of significant forensic value. Our analysis demonstrated that by accessing old superblocks, the entry points for the BTRFS file system, investigators can effectively 'rewind' the file system to previous states, enabling the recovery of deleted files and the examination of earlier versions of the file system.

Furthermore, BTRFS supports the usage of snapshots, which leverage the copy-on-write principle, providing users with a simple and effective way to capture the state of a file system at a specific point in time. Unlike remnants left by previous copy-on-write operations, a snapshot is always maintained in a consistent state by the file system. Our implementation is capable of detecting, listing, and conducting forensic analysis on any given snapshot within BTRFS file system.

In summary, this work underlines the effectiveness of our extended model for file system analysis. In addition, we also provide a practical implementation for the digital forensics community and highlight novel features of contemporary file systems.

### Individual Contributions

Overall, the individual contributions within this paper include:

- Demonstrate the applicability of the extended model for file system analysis for another prominent pooled storage file system
- Examining the intricacies of BTRFS' volume management capabilities
- Implementing the pool analysis step for BTRFS into TSK to enable the analysis of multiple device BTRFS configurations
- Performing a forensic analysis of BTRFS and highlighting its characteristics

# 5 Accompanying Text to Hilgert, Lambertz & Baier (2024)

In our publication "Forensic Implications of Stacked File Systems" [HLB24], we closely examine the forensic challenges posed by a specific class of file systems known as stacked file systems. Unlike traditional file systems, which store data directly on an underlying volume, stacked file systems rely on another file system for data storage. In these cases, we refer to the stacked file system as the upper file system and to the underlying file system as the lower file system. Hence, files within the upper file system are considered upper files, while the lower file system contains the corresponding lower files. This concept is commonly found in many contemporary file systems used for distributed data storage, but it has not been extensively studied from a forensic perspective.

In his foundational work on file system analysis from 2005, Brian Carrier does not address stacked file systems. Furthermore, The Sleuth Kit lacks support for analyzing this class of file systems. In our research, we examine the applicability of Carrier's workflow to stacked file systems and conclude that it does not accommodate them. This limitation arises because Carrier's model assumes a one-to-one assignment between a file system and a single volume, which it uses for data storage, a concept that was already challenged by pooled storage file systems. However, even the extended model for pooled systems does not accommodate stacked file systems, as pooled systems still store data directly on volumes or disks. On the other hand, stacked file systems rely on an underlying file system and therefore require a preceding analysis of this lower file system to enable a comprehensive forensic analysis. Since neither Carrier's original model nor its extended version reflect this, the workflow must be updated to support the analysis of stacked file systems.

For this purpose, we introduce an additional step, which we label *stacked file system analysis*. As previously discussed, this step requires the successful analysis of the underlying lower file system and is therefore placed after traditional file system analysis. Both the input and output of this step are files. In our work, we further define six critical tasks specific to stacked file system analysis that must be carried out during this step in order to achieve a comprehensive analysis:

1. Detection and identification of stacked file systems within the lower file system.
2. Correlation of file names between upper files and their corresponding lower files.
3. Reconstruction of upper files from lower files, addressing fragmentation and data transformation.
4. Analysis of multiple layers of timestamps from both the upper and lower file systems.
5. Detection and extraction of potential slack space within lower files.
6. Utilizing additional recovery methods from both the upper and lower file systems.

In addition to these theoretical considerations, we offer insight into the practical application of this newly added analysis step. To cover a wide variety of implementations, we selected three different stacked file systems, each representing a distinct architectural approach. Based on their underlying architecture, we define three groups, into which stacked file systems can be categorized.

A *local stacked file system* is a stacked file system that operates on the same machine as its corresponding lower file system. An early example for this type is eCryptfs, which facilitated the encryption of data by mounting parts of a lower file system to a new mount point. The other two types of stacked file systems are *managed* and *unmanaged distributed stacked file systems*. As the name suggests, a distributed stacked file system can distribute its data to one or more lower file systems. While these can theoretically also be stored on the same machine, they are in practice deployed on different servers as is the case for file systems such as MooseFS or the Hadoop Distributed File System. While a managed distributed file system like MooseFS requires some entity to orchestrate access to its distributed lower file systems, an unmanaged distributed file system such as GlusterFS contains all necessary information within the lower file systems itself.

Using these stacked file systems as case studies, we conducted experiments and performed the previously described tasks of stacked file system analysis. From this, we present practical key takeaways for investigators having to deal with these file systems:

1. **Identification:** Various indicators from the analysis of the lower file system, such as distinct hierarchies, file names or extensions and extended attributes can aid in identifying a stacked file system. Additionally, the internal structure of lower files may be leveraged to identify it is a lower file.

2. **Correlation of file names:** Our experiments show that in local or unmanaged distributed stacked file systems, original file names and the hierarchy can typically be deduced from the metadata within the lower file system. However, in managed stacked file systems, identifying and extracting metadata from the management component is crucial. Since this process is highly specific to each file system, it requires corresponding forensic tool implementations.
3. **Data Reconstruction:** Our findings indicate that while fragmentation in stacked file systems may be less complex than in traditional file systems, it still requires attention, especially in distributed systems spanning multiple lower file systems. In addition to a file system-specific extraction of information for reassembling lower files, it is also possible to correlate lower files using timestamps, though this is a less reliable alternative. The issues of transformation also highly depend on the stacked file system. In some cases, analysts may benefit from the absence of a transformation layer, allowing for the direct analysis of lower files. However, in cases where data transformation is present, e.g. for encryption or error encoding, retranslating the content of lower files is necessary to obtain the original data. We demonstrated this process for GlusterFS, though the approach varies based on the features of the specific stacked file system at hand.
4. **Timestamps:** For stacked file system analysis, practitioners should leverage both timestamp sources from the upper and lower file systems. Extracting and correlating these timestamps is crucial, particularly in distributed stacked file systems, where data fragmentation allows for more granular timestamp analysis. In cases where the upper file system relies solely on the lower system's timestamps, analyzing the lower system can provide valuable temporal insights, though these timestamps may be more vulnerable to manipulation, as seen with eCryptfs.
5. **Slack Space:** Unlike traditional file systems, stacked file systems generally don't have slack space containing remnants of previous files, as each new upper file creates new lower files. However, our findings suggest that slack space within lower files could be exploited to hide data in some stacked file system implementations. Detecting slack space requires comparing file sizes between the upper and lower systems and cross-referencing replicas of lower files across different systems to identify potential tampering or manipulation.
6. **File Recovery:** Our experiments show that stacked file systems may offer additional file recovery opportunities through their own trash bin features. Investigators should also explore the potential of the lower file system, as original data may still exist there even after deletion from the upper file system. In cases of complete file deletion, other

recovery techniques like file carving can for example utilize the structure of lower files. Therefore, acquiring the lower file system, either physically or logically, provides a more comprehensive recovery approach than focusing solely on the upper file system.

## Individual Contributions

Overall, the individual contributions within this paper include:

- Extending Brian Carrier's workflow for file system analysis to support the analysis of stacked file systems
- Presentation of unique implications of stacked file systems in the context of a file system forensic analysis
- Exploring the implications by performing three case study forensic analysis of stacked file systems
- Discussion of key takeaways for forensic investigators to handle stacked file systems

# 6 Accompanying Text to Hilgert, Lambertz & Mahr (2024)

The publication **Mount SMB.pcap: Reconstructing file systems and file operations from network traffic** [HML24] takes a closer look on how network traffic can be leveraged to enhance traditional file system analysis. Though these two disciplines are usually distinct, captured network traffic may provide valuable information regarding file systems. This is especially important nowadays considering the use of distributed file systems, which read and write data over the network and across multiple clients or servers. While these file systems may utilize proprietary network protocols, sharing of data over the network as well as the protocols for this purpose are not entirely novel. A well-established and commonly used example is the Server Message Block (SMB) protocol.

SMB, which is currently available in version 3, makes use of requests and responses in order to provide a client with access to an SMB share. After a successful session is established, various command types can be exchanged to read and write to the existing file system, which is stored on the SMB share server. In a first step, we examine the possibilities of reconstructing this file system purely from SMB network traffic interacting with it. For this purpose, we define how data for each of the file system data categories presented by Brian Carrier can be extracted using the SMB protocol:

- **Metadata** such as timestamps, file sizes or attributes can already be found in the responses to an `SMB CREATE` request, used to request access to a file or directory.
- While `CREATE` requests also contain information about the hierarchy of the original file system and its **file names**, this information is also found in responses to `QUERY_DIRECTORY` requests. These are typically used to obtain information about all files within a directory matching a certain pattern.
- **Content** of files is typically only found in `READ` and `WRITE` requests. These are matched to the corresponding files using a FileId, which is also found in the associated `CREATE` response.

- While certain SMB requests may yield **file system** information such as its attributes, the data in SMB is not sufficient to allow for an exact replication of the file system layout. The same holds true for the **application data**, which is usually not shared via SMB. However, using information for content, metadata and file names already enables us to reconstruct the most crucial parts of a file system required for a forensic investigation.

In addition to this traditional file system data, utilizing network traffic for the reconstruction of a file system yields two additional and novel sources of information. A file system analysis typically takes place at a certain point in time only providing an analyst with one snapshot of the file system. Since network traffic is a continuous stream of data over time, it may include multiple versions of files that have changed over the course of capturing. Thus, the reconstructed file needs to display all available versions of a file. Secondly, SMB may provide the metadata and file names for a certain file but may not contain its actual data. We refer to these files as hollow files. Since they still may provide value to an analysis, the reconstructed file system should contain them for completeness.

Building upon our knowledge about reconstructing an SMB share's file system from captured network traffic, we have developed and implemented a framework for this purpose. Based on the Fuse library, this framework is capable of taking a network capture, e.g. a pcap file, and mounting it as a common file system. The resulting file system utilizes all of the available information, recreating the original directory layout, assigning the extracted time stamps and providing access to different file versions. The framework not only supports SMB network traffic, but also other protocols such as HTTP, FTP or TLS. Furthermore, it supports various options for customizing the layout of the mounted file system by using network traffic specifics, e.g. source or destination IP.

In addition to reconstructing the file system itself, we have evaluated the possibilities of further enhancing file system analysis by reconstructing file operations. Since the network traffic is a continuous stream of data, it contains information about the actions performed in a chronological order. However, the immediate action causing a sequence of packets to occur is not always obvious. For this reason, we performed experiments involving various Windows API calls on an SMB share and detailing and describing the resulting SMB command sequence. Based on these insights, we introduce SMB Command Fingerprinting (SCF). It uses a combination of characteristics from SMB commands to calculate a fingerprint for an SMB request or response. These fingerprints can then be combined to create a SCF rule matching a certain action performed by a user. As a proof of concept, we created a set of SCF rules and used it to reconstruct user interactions with an SMB share using cmd.exe. Our evaluation showed that this approach is feasible and was able to reconstruct the majority of all executed commands on a very granular level, equipping analysts with a powerful method to understand and analyze SMB network traffic.

### Individual Contributions

Overall, the individual contributions within this paper include:

- Performing an analysis for the reconstruction of a file system using only SMB network traffic information
- Development and implementation of the pcapFS framework for mounting SMB network traffic, also supporting various other network protocols such as FTP and HTTP
- Introducing a novel method and implementation for SMB Command Fingerprinting to reconstruct file operations from SMB network traffic

# 7 Conclusion

With his fundamental work on file system forensic analysis in 2005, Brian Carrier introduced a model and workflow for the analysis of file systems. Alongside his theoretical contributions, Carrier also developed a suite of command-line tools, known as The Sleuth Kit, to facilitate the analysis of a wide range of file systems. However, as file systems have evolved over the past 20 years, Carrier's model has remained unchanged, creating a potential gap in contemporary file system analysis. This gap arises from new concepts and features that his model does not address, as shown by the findings to our first research question.

## $RQ_0$: Which concepts of contemporary file systems are not considered in Brian Carrier's foundational work from 2005?

An initial class of file systems overlooked by Brian Carrier are *pooled storage file systems*. Notable examples include ZFS, the default file system for modern FreeBSD operating systems, as well as BTRFS, which is now part of the Linux kernel. Pooled storage file systems incorporate their own volume management capabilities, diverging significantly from Carrier's workflow, which assumes that a file system is stored in a single volume. For this reason, it is unclear how to reassemble and analyze such a pooled storage file system in a forensic manner.

Similarly, Carrier's 2005 publication did not address the concept of *stacked file systems*, which is now integral to distributed file systems like the Hadoop Distributed File System, MooseFS, and GlusterFS. Like pooled storage systems, these file systems do not align with Carrier's workflow, as they store their data and metadata as files within other file systems. Thus, it is important not only to analyze the stacked file system itself but also to explore how the file system that it uses for data storage should be handled during analysis.

Furthermore, Carrier does not explore the potential of using network traffic to enhance file system analysis, a concept that becomes particularly relevant given that modern distributed file systems inherently rely on network functionality to operate effectively. Even in 2005, remote file system access through file-sharing protocols like SMB, which had been in use for decades, was possible. However, Carrier did not examine this approach within the context of file system analysis, even though it could provide additional and more detailed information during an investigation.

In summary, Carrier's workflow not only omits certain classes of file systems, such as pooled and stacked file systems and lacks guidance on how they should be handled during investigations. Furthermore, the forensic particularities of these file systems, including leveraging features like snapshots, their inherent use of network traffic, and the copy-on-write functionality, remain unexplored, creating a blind spot in contemporary file system forensic analysis. This gap is further highlighted by the lack of tools that support the analysis of these modern file systems and their unique features. To determine whether this is merely an implementation issue or rather an issue with the underlying theoretical foundation, we have discussed the applicability of Carrier's model for these file systems, providing an answer to $RQ_1$.

### $RQ_1$: How applicable is Brian Carrier's 2005 workflow for analyzing contemporary file systems?

To answer this question, we examined the applicability of Carrier's model to contemporary file systems sequentially. We began by focusing on pooled storage file systems, which differ significantly from traditional file system architectures. Pooled storage file systems combine multiple volumes into a single storage pool used to store the file system's data. This approach deviates from the traditional model, where a single file system is typically stored on a single volume, which may consist of multiple volumes, however, by an independent implementation, such as a software RAID.

To evaluate the applicability of Carrier's model for pooled storage file systems, we examined and evaluated each of its steps. Our findings show that steps such as physical media and application analysis remain applicable, as pooled storage systems do not affect the acquisition of data from persistent storage or the handling of resulting files during application analysis. Similarly, we conclude that volume analysis remains relevant and important, since traditional volumes can still serve as pool members within pooled storage file systems. However, because pooled storage systems distribute both data and metadata across multiple underlying volumes using their own volume management, we argue that Carrier's file system analysis step, as originally presented, does not sufficiently address the complexities of analyzing this class of file systems.

Although pooled storage file systems manage their own volume functionality, they still depend on volumes to store data. In contrast, stacked file systems belong to a different class that relies on an underlying file system for data storage. While this concept has existed for many years, modern file systems have started combining it with the distribution of data across multiple lower file systems, often spread across different servers. As with pooled storage systems, the unique characteristics and forensic challenges of stacked file systems were not addressed in Brian Carrier's foundational work. Similarly, forensic tools like The Sleuth Kit lack support for stacked file systems.

Like for pooled storage file systems, we also evaluated the applicability of Carrier's model to stacked file systems. As with pooled storage, the physical media and application analysis steps can remain unchanged. Since stacked file systems do not directly interact with volumes, the volume analysis step is also applicable without modifications. However, unlike pooled storage systems, stacked file systems store their data within an underlying file system. Therefore, in order to conduct a comprehensive analysis of the upper file system, a full analysis of the corresponding lower file system must be performed first. As a result, the traditional file system analysis step remains both critical and necessary during the analysis. However, the analysis of stacked file systems is based on files rather than volumes, as seen in traditional file system analysis. Due to this distinction and other unique characteristics of stacked file systems, we conclude that the traditional file system analysis step, and therefore Carrier's model in its original form, also does not adequately address the analysis of stacked file systems.

Therefore, to effectively support the analysis of both pooled storage and stacked file systems, Brian Carrier's model must be updated, directly addressing our second research question.

## RQ$_2$: How can Carrier's workflow be modified or extended to better support the analysis of contemporary file systems?

To enable support for pooled storage file systems in Carrier's existing model, we have expanded it by introducing an additional step called *pool analysis*. This step not only identifies and correlates pool members, but also examines the behavior of the pooled file system in distributing and transforming its data across these members. As a result, this step is highly dependent on the specific pooled storage file system, similar to the subsequent file system analysis itself. However, direct access to a storage pool provided during the pool analysis allows for a comprehensive file system analysis, including recovery of deleted files and access to unallocated space. Therefore, the inclusion of the pool analysis step in the model paves the way for the forensic analysis of pooled storage file systems.

To demonstrate the practical effectiveness of our extended model, we conducted a detailed examination of the steps required for pool analysis of the pooled storage file systems ZFS and BTRFS. This process involved a deep understanding of their structures and the algorithms they use to distribute and transform data across their pool members. Building on this knowledge, we were able to further develop and refine existing methods for the forensic analysis of these two pooled file systems. Additionally, we implemented the necessary steps to enable a forensic analysis of multiple disk configurations for ZFS and BTRFS within The Sleuth Kit, thereby equipping the digital forensics community with a practical toolset for analyzing these contemporary file systems.

In order to extend the workflow to support stacked file systems, we have furthermore introduced a new step to Carrier's model, which we refer to as the stacked file system analysis. Following the concept of stacked file systems, this step requires files as its input and also results in files akin to a traditional file system analysis. We also define six tasks, based on the intricacies of stacked file systems, that should be addressed during this novel step within the file system analysis workflow to provide a comprehensive analysis. The first and most fundamental task is the detection of stacked file systems, which can be achieved by utilizing the characteristics of its corresponding lower files. Without this detection step, stacked file systems can easily be overlooked in certain analysis scenarios. Another critical task involves data reconstruction, which requires a deep understanding of the methods used by stacked file systems to distribute and transform their data.

To further demonstrate the practical applicability of stacked file system analysis, we selected three examples of them with different architectures as case studies, covering a wide variety, and conducted analyses following the tasks we previously defined. This approach not only offers a theoretical framework for addressing stacked file systems, but also provides practical insights, enabling effective forensic analysis. Additionally, we also identify and demonstrate characteristics unique to stacked file systems that differ from traditional file system analysis, directly addressing our third research question.

## RQ$_3$: What features or characteristics of contemporary file systems offer added value or require special consideration during forensic investigations?

Although using a lower file system for data storage may not typically be seen as a feature of a stacked file system, our research has demonstrated that it can enrich forensic analysis with additional information. For example, utilizing two file systems for data storage can provide an extra set of timestamps from the lower file system. Depending on the specific stacked file system and how it fragments and distributes data, this can offer a more detailed perspective on file modifications. Additionally, we have explored the potential of slack space within lower files to be used for data hiding, a critical aspect that analysts need to consider in their investigation.

During our forensic analysis of pooled storage file systems, we also highlight further novel features that can be found in these contemporary file systems and offer significant additional forensic value. In particular, we explore artifacts generated by the copy-on-write mechanism, which allowed us to access earlier versions of the BTRFS file system. Additionally, we examined the snapshot feature of ZFS and BTRFS, which enables users to capture the entire file system at a specific point in time. Our implementations, built on The Sleuth Kit, are capable of handling snapshots and granting analysts access to these past versions of a file system.

Finally, we examined methods to advance file system forensics by leveraging the "remote character" of contemporary file systems and how they are used. On the one hand, network traffic can be inherent in the file system architecture. Distributed file systems that store data across multiple remote servers depend on network protocols to manage data read and write operations, thus naturally generating network traffic associated with the file system. On the other hand, file systems may be accessed remotely, often achieved by using established protocols such as SMB or NFS. As a result, forensic analysts are presented with network traffic that contains valuable file system-related data. Although network forensics is a critical aspect of digital forensic investigations, it is often conducted separately from file system analysis. In our work, we have shown how to bridge this gap and explored how network traffic containing file system-related data can be leveraged to enhance traditional file system analysis.

To achieve this, we investigated the data required to reconstruct a file system from file system-related data stored within network traffic. As a case study, we focus on the well-established and widely used SMB protocol, which serves as a network protocol to facilitate file system access across multiple clients. We detailed how SMB packets can be utilized to reconstruct the file system and discuss the challenges associated with extracting file system-related data from network traffic. This includes the concept of hollow files, where only metadata is captured in the network traffic without any corresponding content. Additionally, we addressed the presence of multiple file versions within a continuous stream of captured network traffic. Beyond our theoretical exploration and to prove the feasibility of this approach, we introduced pcapFS to the digital forensics community. This framework allows users to mount network traffic as a file system and supports various protocols, including SMB, enabling the reconstruction of a file share's file system directly from network traffic.

Although this approach allows analysts to reconstruct a file system, it does not provide insights into the origins of file modifications. In our work, we have demonstrated that network traffic, as a continuous data stream, serves as a valuable resource for reconstructing file operations that reflect user interactions with the file system. To achieve this, we introduced a novel technique called SMB Command Fingerprinting (SCF), which involves generating hashes, or fingerprints, based on the unique characteristics of SMB network packets and using them to create specific rules that are mapped to specific file operations. As a case study, we executed various commands on an SMB file share using the `cmd.exe` utility and captured the corresponding network traffic. By applying our SCF ruleset, we were able to successfully reconstruct the majority of user interactions, including file and directory creation as well as navigation within the file share thus demonstrating the potential of our approach.

Our work demonstrates that, contrary to Brian Carrier's 2005 perspective, file system and network analysis extend beyond mere correlation. We have shown that our contributions significantly enhance traditional file system forensics by utilizing network traffic to reconstruct not only the file system itself but also the operations performed on it. Our implementations provide forensic analysts with powerful tools that integrate network and file system forensics, enabling a more comprehensive analysis and offering deeper insights into the sequence of events and the various states of the file system.

**Future Challenges** In this work, we extended the existing workflow for file system analysis to encompass contemporary file systems, illustrating how to analyze pooled storage and stacked file systems while highlighting the unique challenges they present. Specifically, we implemented forensic analysis techniques for ZFS and BTRFS, demonstrating how to handle their use of storage pools and novel features such as snapshots and copy-on-write artifacts. Our findings emphasize the critical role of pool analysis in forensic investigations and pave the way for further exploration of other file systems within this category.

We also extended Carrier's theoretical workflow to include stacked file systems and provided practical guidelines for their analysis. Our research highlights not only the importance of correctly identifying stacked file systems but also the necessity and value of a dedicated stacked file system analysis step providing additional forensic insights, such as more granular timestamps and enhanced recovery options. To further empower analysts in handling this class of file systems in the future, developing tailored tools based on our findings is essential.

Building on our theoretical model, which introduces two additional analysis steps, this work also represents a crucial advancement in closing the implementation gap in contemporary file system analysis. Future work should focus on expanding support for additional pooled and stacked file systems in established tools like The Sleuth Kit. However, our research also highlights that Carrier's 2005 model, while foundational, is inadequate for the structured forensic analysis of modern file systems. For this reason, future work should further evaluate its applicability for other contemporary file systems to identify and address possible deficiencies.

In addition to the extended model, we explored the potential of leveraging network traffic to enhance traditional file system analysis. Our study demonstrated the feasibility of reconstructing both the original file system and file operations from network traffic, using the SMB protocol as a case study. These promising results present three major future challenges:

- **Enhancing SMB Command Fingerprinting:** Future research should refine this approach, including automating rule generation and supporting other features of SMB such as encryption. Additionally, the development of an expanded SCF ruleset is essential, covering a broader range of applications, such as file operations performed through Windows Explorer, to increase its applicability and forensic value.
- **Extending Event Reconstruction to other Protocols:** While the focus of this work was on the SMB protocol, many other network protocols facilitate file operations and data transfers. Future research should explore whether similar methodologies can be applied to protocols such as NFS or even proprietary file system-specific protocols used in distributed file systems such as MooseFS. This would broaden the scope of scenarios in which network-enhanced file system analysis could be utilized effectively in forensic investigations.
- **Advancing Cross-Source Forensic Analysis:** This work highlights the potential of leveraging network forensics to enhance file system analysis, but similar opportunities may exist across other digital forensic domains. For instance, memory analysis can reveal volatile data that complements file system investigations, such as file operations and recently accessed or deleted files. Future work should focus on systematically identifying gaps in cross-source analysis in digital forensics, developing workflows that integrate insights from multiple sources, and ensuring that tools are adequately designed for this task. This approach could significantly improve the depth of digital forensic analysis as a whole.

# A Appendix

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.

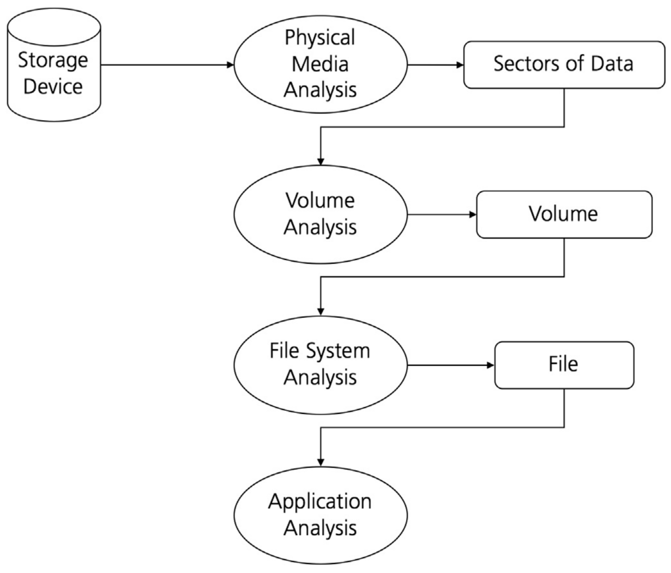


**Fig. 1.** Standard model for a file system forensic analysis by Brian Carrier (Carrier, 2005).

- **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.

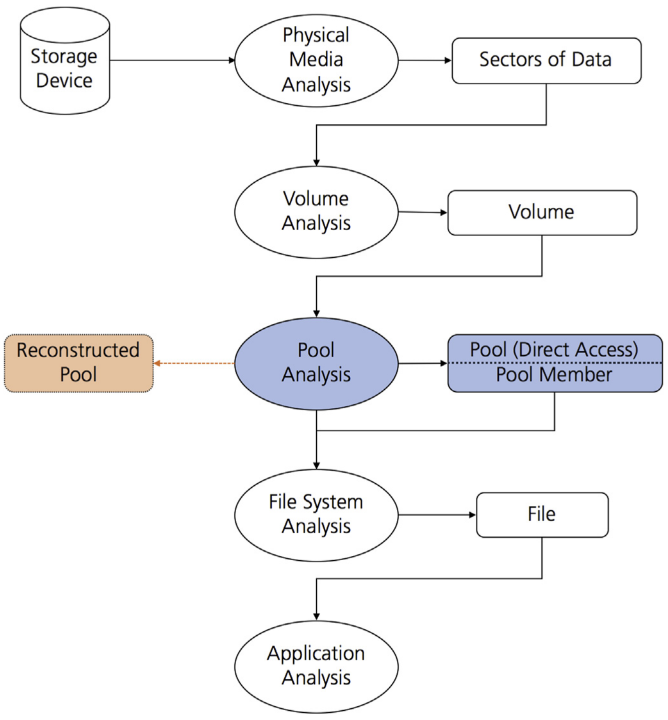


**Fig. 2.** Extended model for a file system forensic analysis for pooled storage file systems.

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.

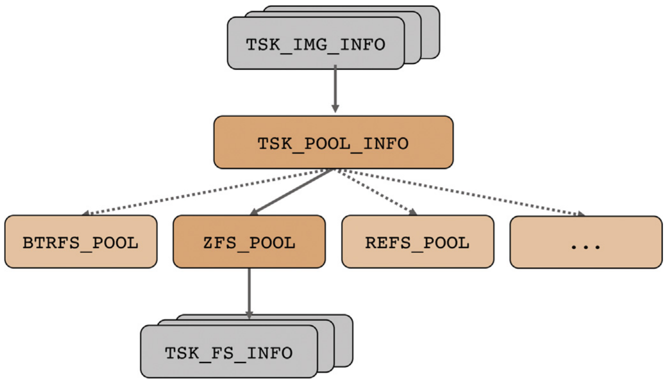


**Fig. 3.** Implementation of the extended model in The Sleuth Kit.

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.

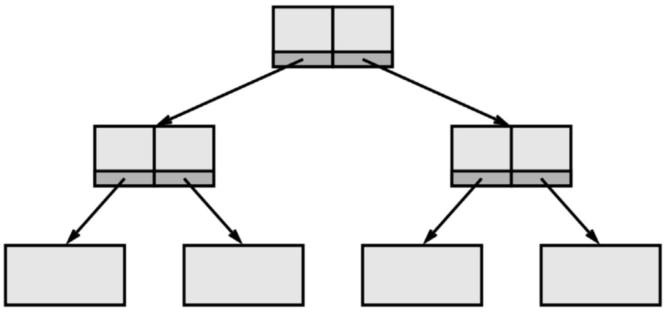


**Fig. 4.** Tree structure implemented in ZFS (Bonwick et al., 2003).

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.

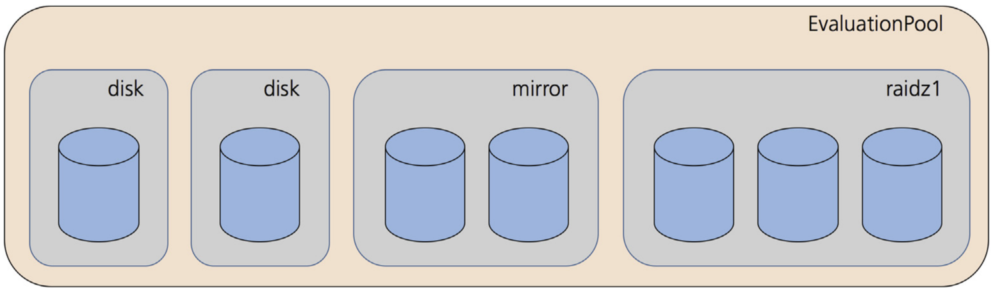


**Fig. 5.** Example `EvaluationPool` consisting of four top-level virtual devices and seven disks in total.

*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)
```

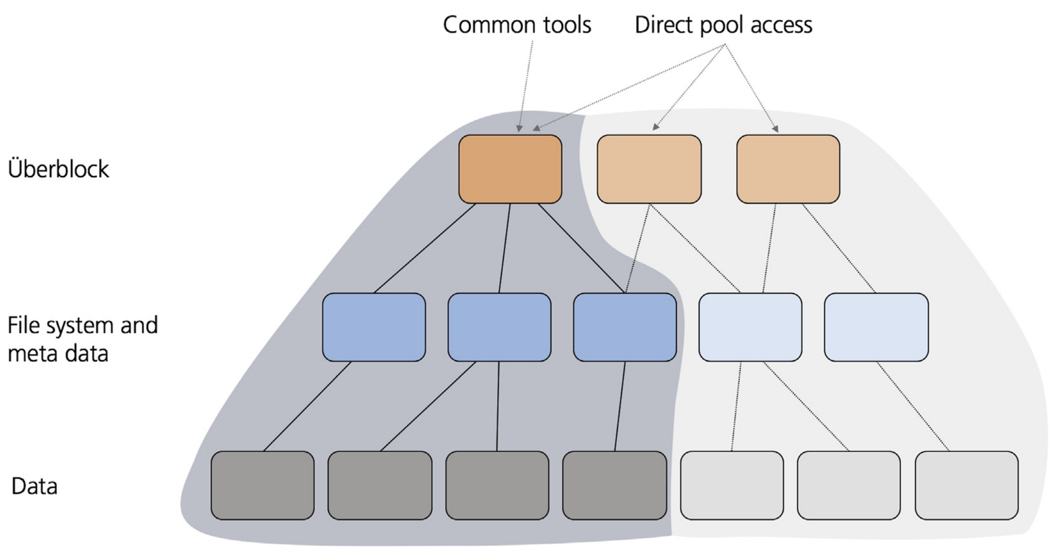


**Fig. 6.** Most recent ZFS tree accessible by common tools compared to old tree structures accessible after the pool analysis step.

*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.

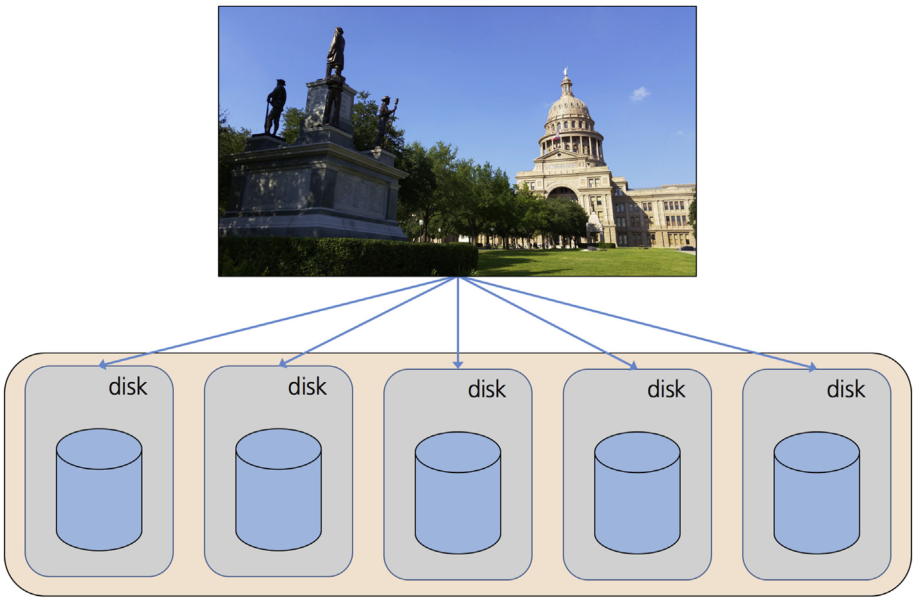


**Fig. 7.** Example image stored across the five top-level virtual devices of a zpool.

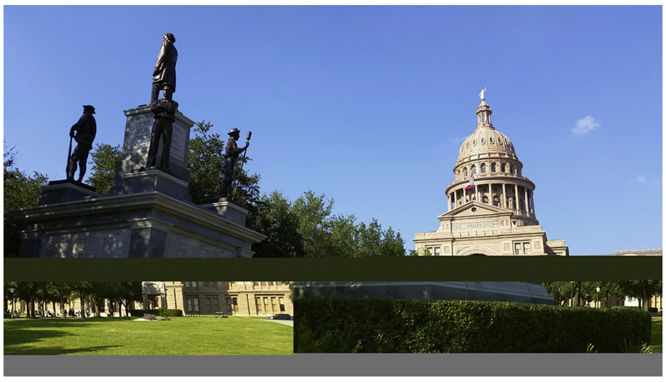


**Fig. 8.** Extracted example image of example pool with one missing disk.

## 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.

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

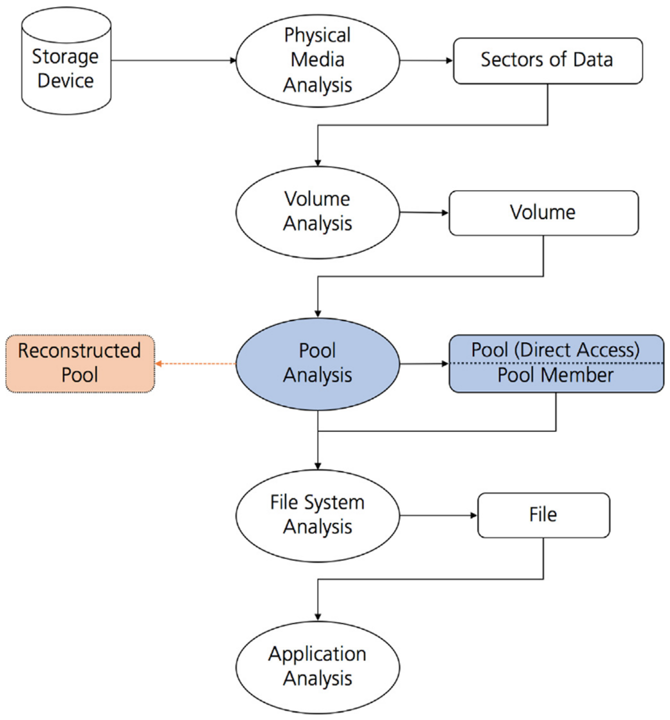


**Fig. 1.** Extended model for a file system forensic analysis of pooled storage file systems Hilgert et al. (2017).

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.

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

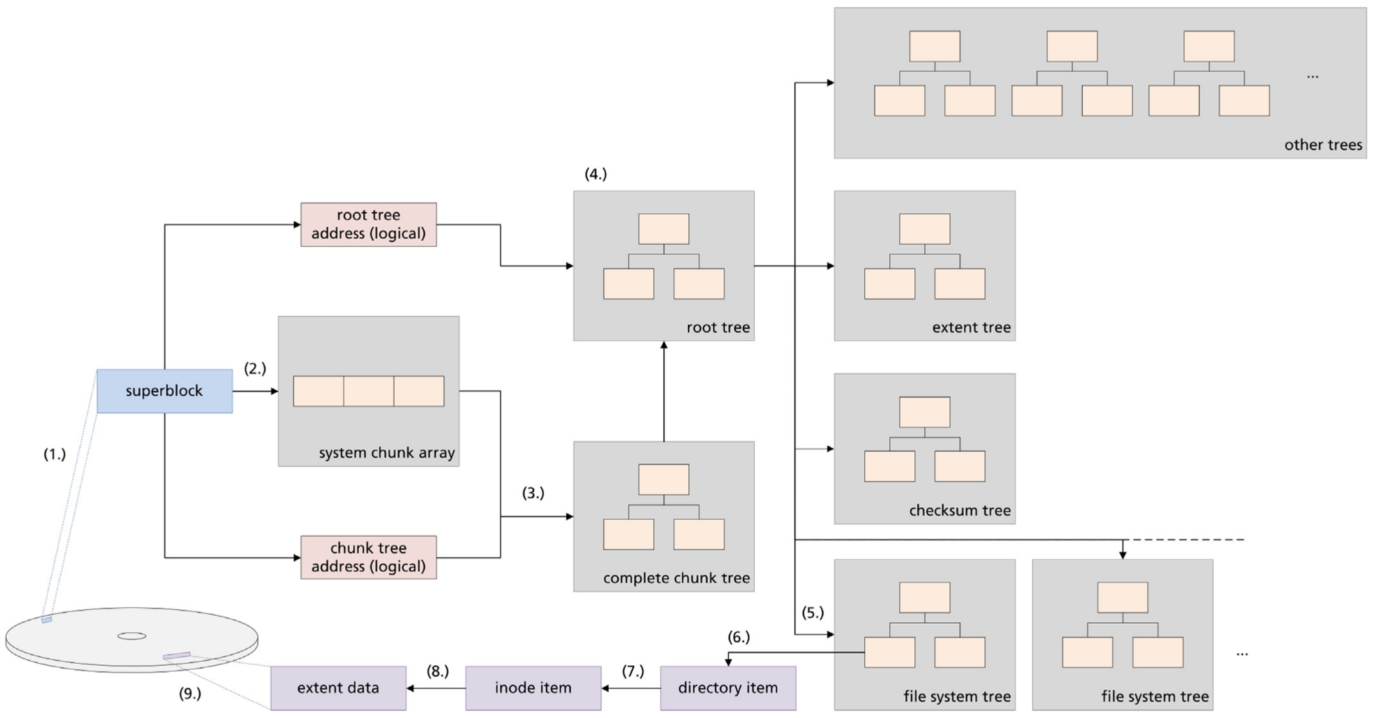


**Fig. 2.** Overview of the most important BTRFS structures used for a file walk.

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 ($\mathtt{t_{log}}$) in the chunk tree. This gives us the logical start address of the chunk ($\mathtt{c_{log}}$).
2. Calculate the **difference** ($\Delta$) between the logical target address and the logical start address of the chunk.

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

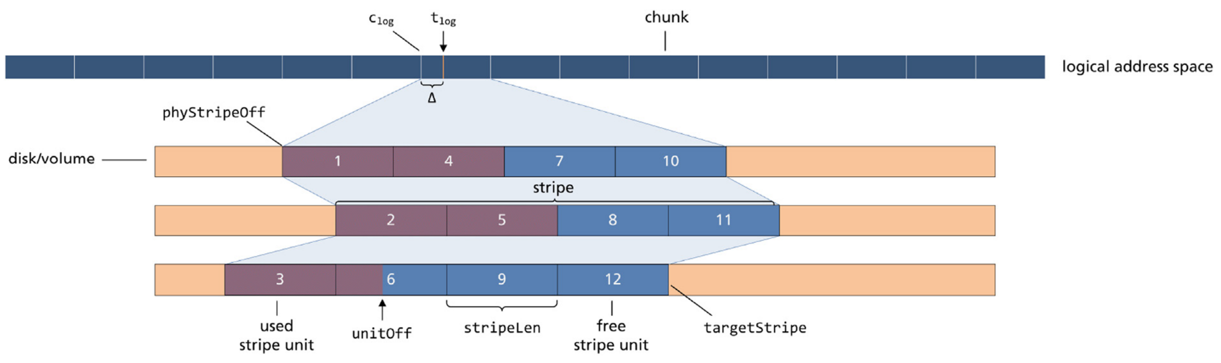


**Fig. 3.** Distribution of data in a RAID0 chunk item using three stripes.

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 stripes 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.

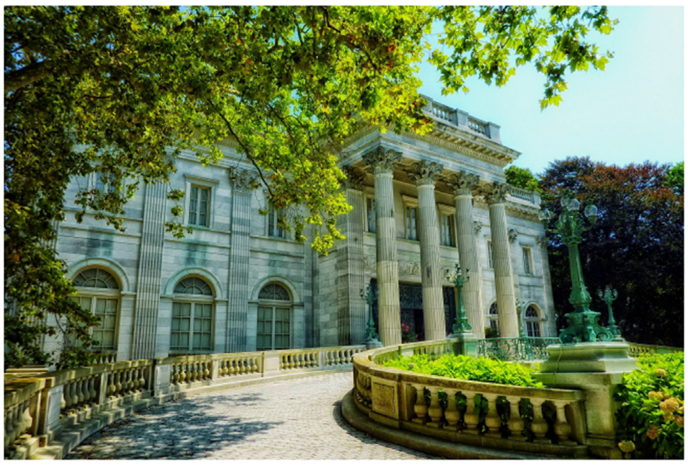
```
  $ fls -P/BTRFS/raid10_5disks/
  r/r 265: 043349.ppt
  d/d 266: data
  + r/r 267: 018367.docx
  + r/r 268: 018370.docx
  + r/r 269: 018371.docx
  $ fls -P/BTRFS/raid10_5disks/-T 41
  using rootTree at logical address: 45629440 (gen-
eration 41)
  r/r 265: 043349.ppt
  d/d 266: data
  + r/r 267: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

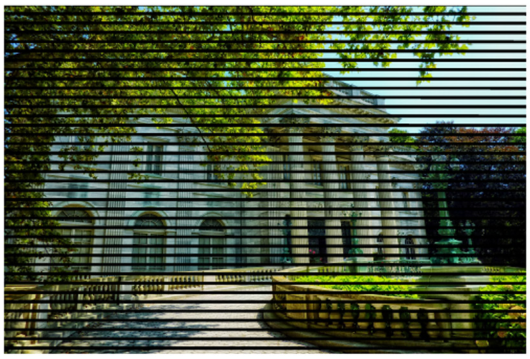
The ZFS extension of TSK by Hilgert et al. provides a parameter to specify an older transaction group number for the recovery of deleted files by using the corresponding überblock stored in the array. In a similar manner, our tool expects the generation number as a parameter, detects the corresponding 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.

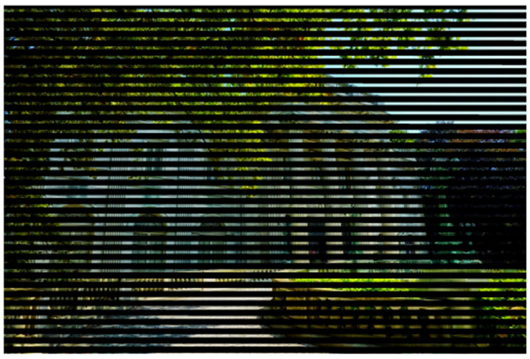

(c) Extracted image after removing two of the disks of the pool disks.

**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.

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS EU 2024 - Selected Papers from the 11th Annual Digital Forensics Research Conference Europe

# Forensic implications of stacked file systems

Jan-Niclas Hilgert [*], Martin Lambertz, Daniel Baier

*Fraunhofer FKIE, Zanderstr. 5, 53177 Bonn, Germany*

ARTICLE INFO

*Keywords:*
Storage forensics
File systems
Stacked file systems
Distributed file systems
moosefs
Glusterfs
Ecryptfs

ABSTRACT

While file system analysis is a cornerstone of forensic investigations and has been extensively studied, certain file system classes have not yet been thoroughly examined from a forensic perspective. Stacked file systems, which use an underlying file system for data storage instead of a volume, are a prominent example. With the growth of cloud infrastructure and big data, it is increasingly likely that investigators will encounter distributed stacked file systems, such as MooseFS and the Hadoop File System, that employ this architecture. However, current standard models and tools for file system analysis fall short of addressing the complexities of stacked file systems. This paper highlights the forensic challenges and implications associated with stacked file systems, discussing their unique characteristics in the context of forensic analyses. We provide insights through three analyses of different stacked file systems, illustrating their operational details and emphasizing the necessity of understanding this file system category during forensic investigations. For this purpose, we present general considerations that must be made when dealing with the analysis of stacked file systems.

## 1. Introduction

File system analysis is undeniably an essential part during any digital forensic investigation involving storage devices. Its goal is to identify and extract files and their corresponding metadata including deleted information. Brian Carrier already laid a profound foundation for this research area almost 20 years ago covering various file systems, some of which are still being used today such as FAT, NTFS and Ext (Carrier, 2005). According to his model for file system forensic analysis, *traditional file systems* store their data on a volume, e.g. a partition or a RAID. Since these volumes are transparent to the file system itself, the underlying implementation creating the volume is responsible for the final transformation and distribution of the actual data. Analyzing these volumes is completely detached from the actual file system at hand and can thus be first addressed in the volume analysis phase, which is then followed by the final file system analysis.

As pointed out by Hilgert et al., these two phases become intertwined, requiring an additional layer in the model when dealing with *pooled file systems* (Hilgert et al., 2017). These file systems utilize multiple disks for redundancy or performance but do not require any extra soft- or hardware for this purpose. In these cases, the file systems themselves handle the distribution of the data across the underlying layer, i.e. volume. Still, the file systems presented in their work stored their data directly on the underlying volume layer.

This work takes a closer look at the forensic analysis of *stacked file systems*. These file systems might also handle the distribution of their data, but they are distinctively characterized by their method of data storage: they do not store their data on a volume or disk but rather on another file system creating new opportunities and challenges for forensic analysis practitioners encountering these file systems. Given the adoption of this concept in distributed file systems like MooseFS and the Hadoop File System, equipping forensic analysts with the knowledge to handle these systems during investigations proficiently is essential.

For this purpose, this paper discusses crucial aspects of the analysis of stacked file systems. To accommodate this, we have revised the standard workflow for file system forensic analysis, making it suitable for the intricacies of stacked systems. We also describe a core set of forensic implications for analyzing stacked file systems, complemented by illustrative findings from three different file systems. The knowledge gathered from our experiments emphasizes the necessity of understanding these implications and is a vital reference for forensic analysts.

## 2. Stacked file systems

*Stacked* or *stackable file systems* store their data on another file system, including both data and metadata, which might be stored in a

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

https://doi.org/10.1016/j.fsidi.2023.301678

Available online 15 March 2024

specialized file format. We denote the stacked file system as the *upper file system* and its files as the *upper files*, which are the files accessible when the file system is mounted. The underlying file system it relies on is termed the *lower file system* storing the *lower files*, as depicted in Fig. 1.

In instances where the upper and lower file systems operate on the same machine, the stacked file system is termed as *local*. Nevertheless, an upper file can encompass multiple lower files, potentially distributed across various detached lower file systems. Given this, the concept of stacked file systems is frequently employed within *distributed* stacked file systems like the Hadoop Distributed File System or MooseFS, as they can be constructed atop a pre-existing and reliable lower file system. Furthermore, distributed stacked file systems can be categorized as either *managed* or *unmanaged*. In a managed setup, a designated entity like a main daemon can be used to orchestrate tasks such as data distribution and managing the metadata of the upper file system. Conversely, in an unmanaged configuration, the systems housing the lower file systems inherently possess all the requisite data to construct the upper file system. Both of these types can be encountered during forensic investigations due to the increasing usage of distributed storage in cloud environments. Hence, comprehending the forensic implications and nuances of stacked file system analysis is crucial.

### 2.1. Related work

A detailed concept of stacking file system layers was already presented in 1994 (Heidemann and Popek, 1994). However, this work focuses on file system development and describes stacking as a method to leverage already existing file systems facilitating the development process of new file systems and features. A few years later, Erez Zadok utilized the concept of stacked file systems to implement a wrapper file system called Wrapfs (Zadok, 1999). While it still stores its data on a lower file system, Wrapfs can be used to create arbitrary upper file systems, for example to provide encryption or prevent deletions of files. In 2007, Zadok together with others discussed various issues of stacked file systems within Linux, such as cache coherency between the upper and lower file system (Sipek et al., 2007). Furthermore, file systems for secure deletion and tracing of file interactions based on the concept of stacked file systems have been proposed (Bhat and Quadri, 2012; Aranya et al., 2004).

While all of the aforementioned research does focus on stacked file systems, it does not cover them from a forensic point of view. Still, limited research on the forensic analysis of distributed stacked file systems has been published (Asim et al., 2019; Harshany et al., 2020). take a closer look at the Hadoop Distributed File System. While their work yields interesting results, such as analyzing various commands and reconstructing distributed data, they do not address the underlying file system used by HDFS. Another analysis of a distributed stacked file system was performed in (Martini and Choo, 2014). During their analysis of XtreemFS, the authors also focused on the Object Storage Devices storing the lower file systems including its identification. However, their work falls short in providing a detailed discussion of general implications of the underlying concept of stacked file systems. Furthermore, the additional value of an analysis of the underlying file system is not examined.

### 2.2. Forensic analysis of stacked file systems

In addition to the research gap, it is important to note that this deficiency extends to forensic tools as well. Current tools, like The Sleuth Kit, are equipped to analyze various lower file system types but lack the functionality to associate them with any existing upper file system. This limitation underscores the need for an updated standard workflow for file system forensic analysis, as depicted in Fig. 2, to effectively handle the complexities of stacked file systems.

In the revised workflow, the initial steps shown in white remain, but we introduce an additional phase, highlighted in purple, specifically dedicated to the analysis of stacked file systems, building upon the results from the prior analysis of the lower file system. This emphasizes that the detection and analysis of traditional file systems continue to be the foundational elements of the process. However, these steps may now yield multiple lower files or metadata files associated with stacked file systems, requiring thorough examination in the newly added step to ensure a comprehensive forensic analysis. Crucially, the results from the stacked file system analysis must also be correlated with information derived from the lower file system analysis and vice versa.

The remainder of this paper deals with the additional step of stacked file system forensics and integration into forensic investigations. In particular, we look at six specifics, we believe are essential for file system analysis: 1) Identification of Stacked File Systems, 2) Correlation of File Names, 3) Data Reconstruction, 4) Timestamps and their Update Behavior, 5) Slack Space and 6) Possibilites for File Recovery.

### 2.3. Experimental setup

To derive the most comprehensive guidance possible, it is crucial to include a diverse range of stacked file systems in the experiments.

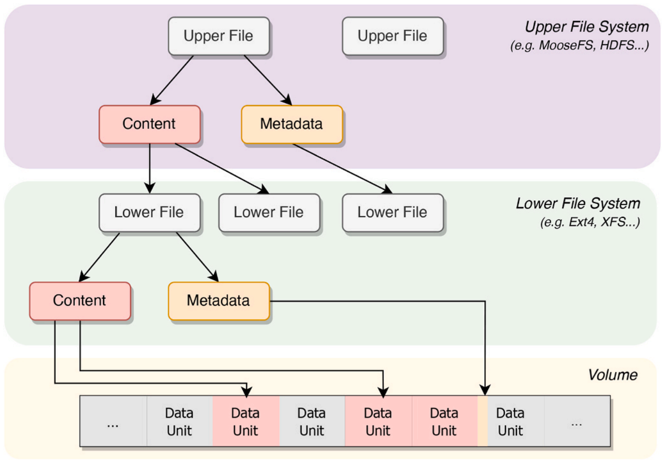


**Fig. 1.** Overview of a stacked file system utilizing a traditional lower file system for data storage.

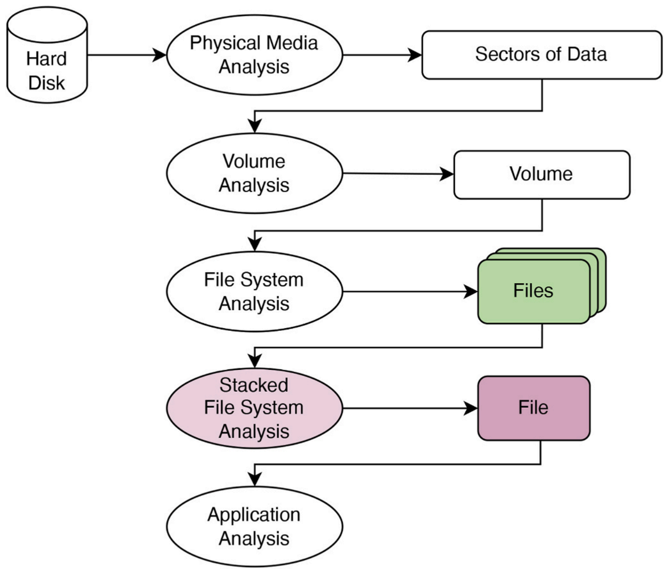


**Fig. 2.** Extension of Brian Carrier's model for the applicability of stacked file systems.

Accordingly, three distinct stacked file systems, previously overlooked in research, were selected as representative examples:

**MooseFS** released in 2008, is an open-source, *managed, distributed stacked file system* designed for big data storage. Its architecture includes Chunk Servers that store data, a Master Server managing metadata, Metaloggers for metadata backup, and a client interface for mounting the file system. In MooseFS, large files are split into smaller chunks distributed across multiple servers.

**GlusterFS** is an *unmanaged, distributed stacked file system that*differs from MooseFS in that it lacks a dedicated master server. Instead, its storage servers form a *trusted pool* by connecting directly to each other. It supports any file system as a *brick*, the lower file system for storing data. These bricks are combined to create a *volume*, which is subsequently mounted by a client.

**eCryptfs** introduced in 2005 as a cryptographic file system to operate on top of an existing file system (Halcrow and ecryptfs, 2005), was integrated into the Linux kernel in version 2.6.19. Although superseded by other mechanisms such as LUKS, eCryptfs remains a notable early example of stacked file systems. It functions as a *local stacked file system*, not used in a distributed manner, and is mounted by specifying a source directory from the lower file system to store its data.

This variety ensures a thorough exploration of the potential scenarios forensic experts may encounter. For our experiments, the stacked file systems were setup, mounted and populated with arbitrary data. Specifics of each experiment are presented in the corresponding section. As a lower file system during the experiments, we utilized Ext4 due to its widespread use and to keep the results comparable. Drawing on these findings, the following sections also outline practical key takeaways to aid forensic investigators in their work with stacked file systems.

## 3. Identification of stacked file systems

As described in Section 2.3, it is crucial to identify a stacked file system following the analysis of the lower file system. During these experiments, the lower file systems were analyzed for any indicators hinting at the usage of a stacked file system.

### 3.1. Findings

#### 3.1.1. MooseFS

As soon as a file system is being used as part of a chunk server in MooseFS, a distinct hierarchy of directories from `00` up to `FF` is created on it. These directories are used to store the chunks, which in turn utilize a file name pattern like `chunk_0000000000000001_00000001.mfs` consisting of an identifier, the chunk ID, a corresponding version and the file extension. Lower files in MooseFS can also be identified by their internal structure that can be inferred by taking a look at the open source code of the file system. In the default MooseFS installation, i.e. not the light version, each chunk begins with a `0x2000` bytes long header. It starts with either a signature of `MFSC 1.0` or `MFSC 1.1`, followed by eight and respectively four bytes representing the chunk ID and version, both of which can also found within the chunk's file name.

#### 3.1.2. GlusterFS

A similar behavior can be observed on servers of a GlusterFS pool, when a volume is created and started. This includes a hidden `.glusterfs` directory storing directories named `00` up to `ff`. Each upper file in GlusterFS is assigned a UUID referred to as the *GlusterFS internal file identifier (GFID)*. This GFID names each lower file inside the hidden hierarchy. GlusterFS also mirrors the upper system's structure in the lower system using hard links as depicted in Fig. 4. While GlusterFS does not make use of any specific internal structure within its lower files, it uses extended attributes to store meta information about its files.

#### 3.1.3. eCryptfs

eCryptfs on the other hand does not create a unique hierarchy on the lower file system. Instead, the hierarchy of the files and directories of the upper file system are stored in an identical way on the lower file system. If file name encryption is enabled, the distinct prefix `ECRYPTFS_FNEK_ENCRYPTED` defined in the Linux kernel source is used for each lower file. Lower files in eCryptfs contain magic markers stored in a special header. These markers can be detected by performing an XOR operation on bytes 9–12 of the file at hand using the magic `0x3c81b7f5`. The resulting 8 bytes should match bytes 13–16 in case of an eCryptfs lower file.

### 3.2. Key takeaways

Depending on the stacked file system at hand various types of indicators resulting from the analysis of the lower file system can be used for its identification. This includes distinct hierarchies, file structures as well as certain extended attributes. Furthermore, the internal structure of lower files can be used to identify them directly, for example in cases in which they are included in a backup outside of the lower file system.

Once identified, investigators can mount the stacked file system using its native software or conduct an in-depth forensic examination. However the current shortfall in forensic tools specifically designed for stacked file system analysis necessitates manual reconstruction of the system under investigation at the moment.

## 4. Correlation of file names

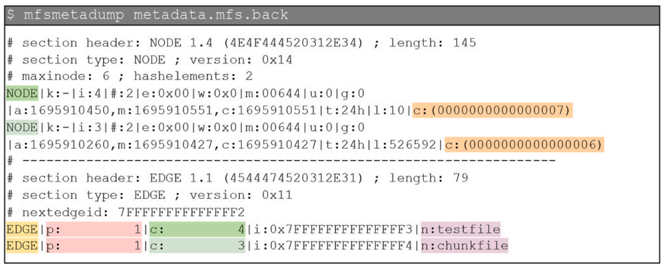
For a more comprehensive analysis and deeper understanding, it is essential to establish the relation between the names of upper files and the corresponding lower files that represent them. During this experiment, we analyzed if and how this connection could be determined. Furthermore, we examined how the entire hierarchical structure of the upper file system is reflected within the lower file system.

### 4.1. Findings

#### 4.1.1. MooseFS

In MooseFS, neither the header, the file name or any other metadata of a lower file contain any reference to the original upper file. In order to obtain this relation and thus also the file name, it is necessary to analyze information stored on the Master Server. By default, chunk servers use the DNS name `mfsmaster` to connect to the Master Server. However, this can be configured within the chunk server's configuration stored in`/etc/mfs/mfschunkserver.cfg`. The Master Server stores its metadata files within the directory`/var/lib/mfs`, including `metadata.mfs.back`. This metadata file can be extracted and subsequently inspected or analyzed using the `mfsmetadump` tool. During our experiments, recent MooseFS updates were not instantly reflected in the metadata file, requiring a Master Server restart to save these changes.

Fig. 3 illustrates the `mfsmetadump` utility output. In MooseFS, file names are stored as `EDGE`s in the filesystem tree, found in the `EDGE` section. Each line represents a file, detailing the parent inode, child inode, and file's name. The child inode number can be used to link an



```
$ mfsmetadump metadata.mfs.back

# section header: NODE 1.4 (4E4F444520312E34) ; length: 145
# section type: NODE ; version: 0x14
# maxinode: 6 ; hashelements: 2
NODE|k:-|i:4|#:2|e:0x00|w:0x0|m:00644|u:0|g:0
|a:1695910450,m:1695910551,c:1695910551|t:24h|l:10|c:(0000000000000007)
NODE|k:-|i:3|#:2|e:0x00|w:0x0|m:00644|u:0|g:0
|a:1695910260,m:1695910427,c:1695910427|t:24h|l:526592|c:(0000000000000006)
# ----------------------------------------------------------------
# section header: EDGE 1.1 (4544474520312E31) ; length: 79
# section type: EDGE ; version: 0x11
# nextedgeid: 7FFFFFFFFFFFFFF2
EDGE|p:           1|c:           4|i:0x7FFFFFFFFFFFFFF3|n:testfile
EDGE|p:           1|c:           3|i:0x7FFFFFFFFFFFFFF4|n:chunkfile
```

**Fig. 3.** Excerpt of the `mfsmetadump` tool displaying metadata of a MooseFS file system.

entry to a corresponding `NODE` section entry, which represents an upper file.

#### 4.1.2. *GlusterFS*

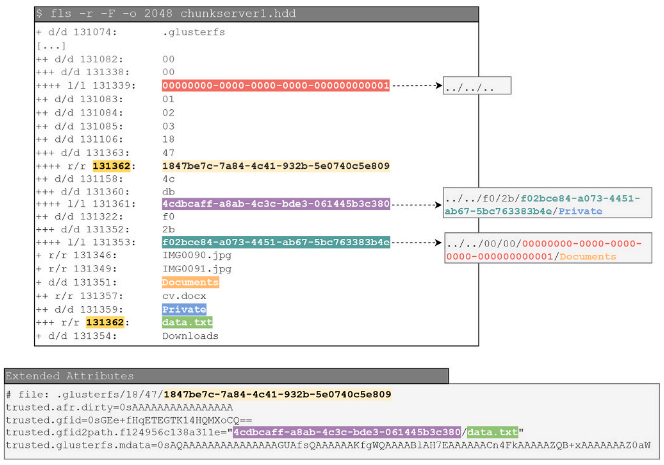
Fig. 4, the lower files `1847be7c-7a84-4c41-932b-5e0740c5e809` and `data.txt` share the same inode number. Additionally, GlusterFS also utilizes extended attributes and soft links, which can be analyzed to infer the hierarchy. The extended attributes of the lower file contain a reference including the original file's name as well as the GFID of the directory, in which it was stored. The lower file belonging to this directory is again a soft link pointing to its own parent directory and so on.

#### 4.1.3. *eCryptfs*

If the eCryptfs file system is mounted files within the upper file system can be matched to the files in the lower file system by comparing the corresponding inode numbers. This is already implemented in the `ecryptfs-find` utility. By default, the file names of the lower files are identical to the file names of the corresponding upper files. In case of file name encryption, eCryptfs utilizes a file name encryption key (FNEK), which is required to reveal the original file name of the lower file at hand. However, eCryptfs stores a hex signature of the utilized FNEK within all of the encrypted files names. For this reason, it is still possible to infer, which lower files were encrypted using the same FNEK and thus probably belonged to the same mounted file system. The signature of the FNEK is encoded within the FNEK-encrypted file name, also referred to as a Tag 70 packet and follows the packet type `0x46` and the length of the packet. By decoding the file name it is possible to extract the signature of the FNEK used, which can be used for further analyses.

### 4.2. *Key takeaways*

Our experiments indicate that in local or unmanaged distributed stacked file systems, it is generally possible to deduce the original file names and file system hierarchy. This is to be expected as for these kinds, the corresponding metadata can be found within the lower file system. In contrast, with stacked file systems that incorporate a management component, e.g. a dedicated server, it becomes vital to identify and extract the metadata that holds this information. Our findings demonstrate how this analysis can be executed for stacked file systems like MooseFS, enabling the determination of the relationship between upper and lower files. However, this task varies significantly depending on the specifics of the stacked file system, necessitating customized implementations within forensic tools.



```
$ fls -r -F -o 2048 chunkserver1.hdd
+ d/d 131074:       .glusterfs
[...]
++ d/d 131082:      00
+++ d/d 131338:     00
++++ l/l 131339:    00000000-0000-0000-0000-000000000001 ------> ../../..
++ d/d 131083:      01
++ d/d 131084:      02
++ d/d 131085:      03
++ d/d 131106:      18
+++ d/d 131363:     47
++++ r/r 131362:    1847be7c-7a84-4c41-932b-5e0740c5e809
++ d/d 131158:      4c
+++ d/d 131360:     db
++++ l/l 131361:    4cdbcaff-a8ab-4c3c-bde3-061445b3c380 ------> ../../f0/2b/f02bce84-a073-4451-
                                                                   ab67-5bc763383b4e/Private
++ d/d 131322:      f0
+++ d/d 131352:     2b
++++ l/l 131353:    f02bce84-a073-4451-ab67-5bc763383b4e ------> ../../00/00/00000000-0000-0000-
                                                                   0000-000000000001/Documents
+ r/r 131346:       IMG0090.jpg
+ r/r 131349:       IMG0091.jpg
+ d/d 131351:       Documents
++ r/r 131357:      cv.docx
++ d/d 131359:      Private
+++ r/r 131362:     data.txt
+ d/d 131354:       Downloads
```

```
Extended Attributes
# file: .glusterfs/18/47/1847be7c-7a84-4c41-932b-5e0740c5e809
trusted.afr.dirty=0sAAAAAAAAAAAAAAAA
trusted.gfid=0sGEe+fHqETEGTK14HQMXoCQ==
trusted.gfid2path.f124956c138a311e="4cdbcaff-a8ab-4c3c-bde3-061445b3c380/data.txt"
trusted.glusterfs.mdata=0sAQAAAAAAAAAAAAAAAGUAfsQAAAAAAKfgWQAAAABlAH7EAAAAAACn4FkAAAAAZQB+xAAAAAAAZ0aW
```

**Fig. 4.** Example of a hierarchy on a lower file system in GlusterFS.

## 5. Data reconstruction

For the reconstruction of upper files from their corresponding lower files, analysts need to tackle common problems such as fragmentation and data transformation.

### 5.1. *Fragmentation*

In most cases, the content of a file does not fit into a single data unit, which is why file systems allocate multiple data units. While different allocation strategies may be used, it often results in file fragmentation. Thus, traditional file systems need to keep track of the exact data units used by a file as well as the order in which they belong. This fragmentation not only complicates forensic efforts but has also been a long-standing focal point of research (Garfinkel, 2007; van der Meer et al., 2021). Yet, the topic of fragmentation in stacked file systems has not been explored. To address this, we have created multiple large-sized upper files, aiming to analyze and understand the fragmentation patterns in the stacked file systems under study.

#### 5.1.1. *Findings*

*5.1.1.1. MooseFS.* Our experiments demonstrated that MooseFS splits files larger than 64 MiB into multiple lower files, irrespective of the number of chunk servers. Although mountable with a single chunk server, MooseFS ideally operates with multiple, and it is advised to use at least three, as done in our experiments. By default, each chunk is replicated onto two of the three available chunk servers. Consequently, large files, fragmented into multiple chunks, may be distributed across all chunk servers within the MooseFS file system. Since information within the chunks themselves did not suffice to reassemble an upper file, it is necessary to consult the Master Server metadata to efficiently assemble fragmented upper files. As depicted in Fig. 3, the `NODE` section stores a list of chunks composing the upper file, each identified by a unique ID, which is also reflected in the chunk name on the lower file systems. Since it is unique to each chunk, it can also be used to identify replicas of chunks across multiple chunk servers.

*5.1.1.2. GlusterFS.* Depending on the type of volume used, fragmentation as well as replicas of lower files can be encountered. The most important volume types are:

- **Distributed:** In this default mode, upper files are not fragmented, but stored randomly across all available bricks, i.e. all available lower file systems.
- **Replica:** This mode is used to ensure redundancy by storing unfragmented upper files across multiple bricks similar to RAID mirrors. The corresponding lower files stored across multiple lower file systems can be correlated by their GFID file name.
- **Dispersed:** A dispersed volume can be compared to a RAID5-like volume. Data is split and stored across multiple lower file systems along with parity information. Again, fragments belonging together can be matched by their GFID. When creating a dispersed volume, it is possible to configure the number of bricks used for redundancy, i.e. how many bricks can be lost without causing any data loss.

The first two modes do not cause any fragmentation of upper files. However, the replicated mode causes multiple copies of the same files to be stored on multiple lower file systems, i.e. storage servers. To identify these lower file systems for further analysis, the configuration of the GlusterFS at hand can be utilized. A directory for each volume of a storage server can be found in its `/var/lib/glusterd/vols/` directory. It stores the volume configuration in an `info` file and consists of a `bricks` subdirectory that offers configurations for each associated brick. These files appear across all storage servers in the GlusterFS pool that created the volume, detailing the server's hostname and brick path. Notably, the values `listen-port` and `brick-fsid` only seem to exist in the brick configuration of the respective server. This also allows for pinpointing the exact GlusterFS server at hand.

When dealing with dispersed volumes, upper files become fragmented within GlusterFS. An efficient way to identify a lower file of a dispersed volume is to analyze its extended attributes. Each chunk belonging to a dispersed file utilizes extended attributes in the `trusted.ec` name space, e.g. `trusted.ec.size` which stores the real size of the corresponding file. However, they do not contain any information about the order in which they should be reassembled. Furthermore, GlusterFS uses Erasure coding for dispersed volumes, which requires an additional step to obtain the original version of the file described in the next section.

*5.1.1.3. eCryptfs.* In eCryptfs upper files are not split into multiple lower files and thus no fragmentation occurs.

#### 5.1.2. Key takeaways

Though our findings illustrated that fragmentation may not be as complex as with traditional file systems, it still has to be considered especially when dealing with distributed stacked file systems spanning across multiple lower file systems. In these cases, it is crucial to identify, which other servers were part of the stacked file system at hand in order to adequately extend the acquisition process.

Furthermore, our experiments showed that the upper file system's metadata plays a crucial role for an efficient reassembly of fragmented files highlighting the importance of dedicated approaches for stacked file system analysis. In absence of this information, correlating lower files via their timestamps is an alternative though less reliable due to discrepancies as discussed in Section 6.

### 5.2. Transformation

Unlike early file systems, more advanced ones like APFS or ZFS started to implement features such as encryption or compression. This resulted in some kind of transformation between a file's original content and the content stored on disk. A similar concept may also be employed by stacked file systems for various purposes like encryption or the utilization of erasure coding, when data is distributed. For this analysis, we compared the content of lower files to the original content stored within the upper files of the stacked file system.

#### 5.2.1. Findings

*5.2.1.1. MooseFS.* Besides the inclusion of an extra `0x2000` byte chunk header, the open-source MooseFS leaves the original data unaltered.

*5.2.1.2. GlusterFS.* In distributed and replicated volumes, GlusterFS leaves the original content in lower files unchanged as well. However, for dispersed volumes, it employs a Reed-Solomon based Erasure coding. For an efficient recovery of dispersed files, we recreated the relevant GlusterFS setup to tackle the fragmentation as well as transformation hurdle. After the original GlusterFS configuration is identified as described in the previous section, it is possible to recreate a new GlusterFS volume using identical parameters. Afterwards, the obtained lower file systems can be copied to the freshly created GlusterFS bricks. It is essential to preserve extended attributes; failure to do so will lead GlusterFS to misidentify dispersed files. Additionally, the sequence of declaring bricks is crucial, i.e. the original first lower file system's data should populate the first brick in the new volume and so on. Any inconsistency led to reconstruction failure in GlusterFS in our tests.

*5.2.1.3. eCryptfs.* Since eCryptfs's main feature is encryption, file contents found on the lower file system are naturally encrypted. Additionally, the cryptographic context for each file is stored in a header preceding the encrypted data. The minimum size for this header is defined as 8192 bytes, resulting in slightly larger files on the lower file system compared to the original stacked file system. Though 8192 bytes is only the minimum size, we did not encounter any larger header sizes in our experiments including files up to 1 GB. The size of the original file is not encrypted and can be found in bytes 0–7 of the header, which starts directly at offset 0 of the lower file. Further information within the header includes the version as well as the encrypted session key used for the encryption of the file's content.

#### 5.2.2. Key takeaways

Depending on the stacked file system at hand, practitioners can benefit from the absence of a transformation layer during their analysis. This enables them to analyze a the files of a lower file system without the need to perform an analysis of the upper stacked file system. However, in certain cases, when encryption or error encoding is utilized, it is required to retranslate the content of lower files to obtain the original file content. We have illustrated various considerations that have to be made when using native software to perform this task for GlusterFS. Yet again, this strongly depends on the features of the stacked file system at hand.

## 6. Timestamps and their update behavior

In traditional file systems, timestamps are stored along the metadata of the files and include information about the last Access, Modification, Change and in some cases Birth time of a file. The intricacies and challenges of interpreting timestamps are well-recognized within the digital forensic community (Raghavan, 2013).

Naturally, these timestamps retain their critical importance in the context of stacked file systems. However, we encounter an additional layer of timestamp sources:

- **Upper file system:** Timestamps of the upper file system refer to the upper files and consist of one set of timestamps per upper file. The way this meta information is stored is completely specific to the upper file system itself.
- **Lower file system:** Employing an additional file system to store file content introduces an extra layer of timestamps stored along the lower files in the lower file systems.

Moreover, it is equally important to grasp the timestamp update behavior within both the upper and lower file systems as well as how they affect each other. In our experiments, we conducted fundamental file operations like creating and modifying files to examine how the stacked file systems in question update timestamps. This investigation encompassed both the lower and upper file systems, with a particular focus on understanding how timestamps in the latter could be accurately retrieved. We kept a multi-server configuration for the distributed file systems to observer the timestamp update behavior across multiple lower file systems.

### 6.1. Findings

The initial part of this section details the findings on timestamp sources, while the subsequent sections explore the timestamp update behavior of the corresponding file system.

#### 6.1.1. Timestamp sources

MooseFS keeps track of the timestamps for all of its upper files within the metadata that can be found on the Master Server or Metaloggers. This information can be extracted by using the `mfsmetadump` utility as shown previously in Fig. 3. In GlusterFS, this information is not stored in an external file, but directly within the extended `trusted.glusterfs.mdata` attribute of the corresponding lower files across all bricks. Any change to the upper file's timestamps inevitably results in an update of the metadata of the lower files. The actual timestamps can be extracted from the decoded Base64 string stored within the extended attribute. It holds 8 byte timestamps in seconds followed by the timestamp for nanoseconds following the big-endian format as shown in Fig. 5 eCryptfs relies solely on the timestamps already present in the lower file system, without storing any additional timestamp information.

### *6.1.2. Update behavior for MooseFS*

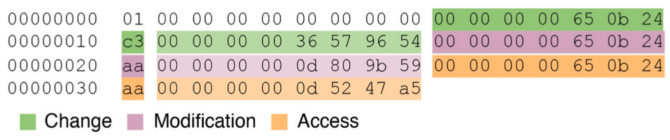
When a file smaller than the maximum chunk size is created, two identical chunk copies are made on two out of three chunk servers. Although MooseFS sets the Modification and Change timestamps of the upper file identically, the Access timestamp appeared slightly earlier in our tests. This pattern was also seen in timestamps of the corresponding chunk servers. Notably, the birth timestamp from the lower file system is not reflected in MooseFS. Furthermore, it was observed that different chunk servers displayed varying timestamps for the same chunk.

When an upper file is modified, its Modification and Change timestamps are updated to the same value. The same holds true for the corresponding chunks stored within the lower file systems. However, timestamps might again vary across chunk servers. If the upper file's timestamps are changed without data alteration, e.g. by utilizing the `touch` command in Linux, the chunk timestamps remain unaffected.

The update of File Access timestamps is rather complex and depends on multiple of factors:

- **MooseFS Configuration:** The `ATIME_MODE` in the Master Server config determines the Access time update policy for upper files. Default is always, with options like "always for files" or "never" (similar to Linux's `noatime`).
- **Client Caching:** When mounting MooseFS, it is possible to set a *data cache mode*. Options include `DIRECT` (no caching) and `YES` (always use cache). Default is `AUTO`, which behaved like `YES` in our tests.
- **Chunk Pre-Fetch:** For performance, MooseFS uses pre-fetch and read-ahead algorithms on chunk servers to pre-load expected chunks into the OS memory. This is hardcoded and cannot be changed.
- **Lower File System Configuration:** The lower file system on the chunk server has its own Access timestamp policy. In Linux, the default is `relatime`, which doesn't update Access times with every access.

In MooseFS, the file access timestamps for upper files are influenced by its configuration and client caching. It was observed that when *client caching is disabled*, every file access updates the Access timestamps on the client, which the Master Server adopts in the default configuration. If *client caching is enabled* however, the Access timestamp stamp of a file is only updated on its first access or when it gets reloaded into cache. If MooseFS is however configured to never Modification the Access timestamps, client-side updates aren't stored on the Master Server and are lost almost instantaneously. In our tests, Access timestamps for lower files were updated upon the chunk server daemon's initial start, provided its access time mode was set accordingly, e.g. using the `strictatime` option. With 10,000 files (and corresponding lower files), Access timestamps changed post-daemon start without client read requests. This is likely due to MooseFS's pre-fetch algorithms reading data in memory for some time, though no clear order was discernible.



```
00000000  01 00 00 00 00 00 00 00 00  00 00 00 00 65 0b 24
00000010  c3 00 00 00 00 36 57 96 54  00 00 00 00 65 0b 24
00000020  aa 00 00 00 00 0d 80 9b 59  00 00 00 00 65 0b 24
00000030  aa 00 00 00 00 0d 52 47 a5
```

**Fig. 5.** Structure of the `trusted.glusterfs.mdata` extended attribute containing timestamps in GlusterFS.

Reading all files from the client (with caching off and default MooseFS settings) resulted in the updating of all 10,000 Access timestamps in the lower file system. Yet, in setups with fewer upper files, Access timestamps of chunks only updated during daemon startup, not during later client requests. The cause for this disparity is still unclear and requires further research. Given these complexities, interpreting Access timestamps on MooseFS's lower file systems demands caution.

On the other hand, Modification timestamps were consistently accurate and updated as anticipated, which is especially relevant for large upper files generating multiple chunks.

*Large files:* In MooseFS, files exceeding the maximum chunk size are divided into multiple chunks. When such a large file is created, the timestamps in MooseFS and the underlying file system are set in the manner previously detailed. Thus, a 200 MB upper file results in eight (four distinct but replicated) lower files, each with unique timestamps, spread across three chunk servers.

In our experiment, we modified the first bytes of the 200 MB file. While MooseFS only holds a singular set of timestamps for the upper file, the Modification and Change timestamps are updated the same way regardless of the position, in which the file is modified. On the lower file systems however, only the Modification and Change timestamps of the impacted chunk, the first of four, were updated across the two chunk servers hosting that chunk. A similar pattern was observed when other sections of the file were altered: only the relevant chunk's timestamps changed. This level of granularity provides a more intricate view into file modifications on the upper file system.

In our MooseFS experiments, we observed an unexpected behavior where chunks sometimes moved between chunk servers after file modification or idling periods. While MooseFS naturally rebalances chunks across servers, the reasons for these specific movements were unclear. Crucially, this behavior has implications for timestamps. When a chunk is relocated to a new server, it behaves as if it's newly created, thus resetting all its timestamps to the time of the relocation.

### *6.1.3. Update behavior for GlusterFS*

When a file is created in GlusterFS, the Modification and Change timestamps of the upper file are set to the same value, while the Access timestamp was always set to a value a little earlier. For lower file systems, the behavior of the initial timestamps depended on the volume mode. For a replicated volume, all timestamps were set to same value, while a dispersed volume resulted in different Change and Modified timestamps. Furthermore and as expected, the timestamps across the lower file systems stored on multiple servers varied. Additionally, the Birth timestamp was utilized by the lower file system, but also not populated to the upper file system.

After the modification of a file, the Modification and Change timestamps of the upper file were updated to the same values. Furthermore, the Modification, Change and Access of the lower files were updated.

Since the Access timestamp of an upper file has to be propagated to each lower file, GlusterFS doesn't by default keep track of Access times preventing any performance drops. It was however observed, that access to an upper file could update the Access timestamp of a corresponding lower file depending naturally on the `atime` configuration of the lower file systems. In a setup with three replicated bricks, the specific accessed lower file alternated for each access. Furthermore, the timestamp was not updated for each access, most likely due to again some kind of caching performed within the client. Caching within the GlusterFS servers itself was not observed.

### *6.1.4. Update behavior for eCryptfs*

When a file is created, the Access, Modification and Change timestamps in eCryptfs are all set to the same value, which is the moment the file was written and thus created. The exact same timestamps can be found on the corresponding lower file system. Though it is populated, eCryptfs as well does not utilize or return the Birth timestamp stored in the lower file system.

After the modification of a file, the Modification and Change timestamps were updated and contained the same values within eCryptfs as well as the lower file system. The same holds true for a file access and metadata modification, updating the corresponding Access and Modification timestamps respectively. Consequently, all of the timestamp modifications performed directly on the lower file system were also mirrored to the stacked file system.

### 6.2. *Key takeaways*

For stacked file system analysis, we advise practitioners to harness both potential sources of timestamps within the upper and lower file system. Extracting timestamps from the upper file system is crucial, as outlined in our previous section for MooseFS and GlusterFS. In addition, timestamps of the lower files should also be extracted and analytical methods need to be able to correlate both timestamp sources. This approach is particularly beneficial in distributed stacked file systems, where data fragmentation leads to a more detailed level of timestamp granularity for each file.

Furthermore, in situations where the upper file system depends solely on the lower file system's timestamps, two aspects should be considered: First, analyzing the lower file system can already provide valuable temporal insights. Second, as our eCryptfs example shows, these timestamps may be more susceptible to manipulation.

Moreover, akin to conventional file system forensics, understanding the behavior of timestamp updates in both the upper and lower file systems is essential.

## 7. Slack

In traditional file systems, whenever a file's size does not align with the end of a data unit, some unused space between the file's end and the data unit is created. This *file slack*, if not overwritten properly, can contain artefacts of previously stored data within this data unit or can simply be used to intentionally hide data. The exploration of slack space, including its possibilities, detection, and analysis, has already been conducted across various file systems, including NTFS (Huebner et al., 2006), BTRFS (Wani et al., 2020) or APFS (Göbel et al., 2019).

Yet, this scrutiny has not been extended to stacked file systems, which uniquely store a file's content in other files rather than in traditional data units. These lower files may be aligned with a certain *extent* size, e.g. always being a multiple of 4 KiB, resulting in slack space similar to the previously described file slack in traditional file systems. We refer to this slack space as *lower file slack* since it occurs between the actual end of the file and the end of the lower file. Additionally and unlike in traditional file systems, data can be directly appended to a certain lower file directly, since it is represented as a file itself. This way it may be possible to hide data, which is not considered by the upper file system. We define this type of slack as *extra lower file slack*. Fig. 6 illustrates these different types of slack. It is important to understand, if these types of slack exist within a stacked file system and how they can be detected and extracted. During the following experiments, we have evaluated the feasibility of slack space within stacked file systems by utilizing it to hide data.

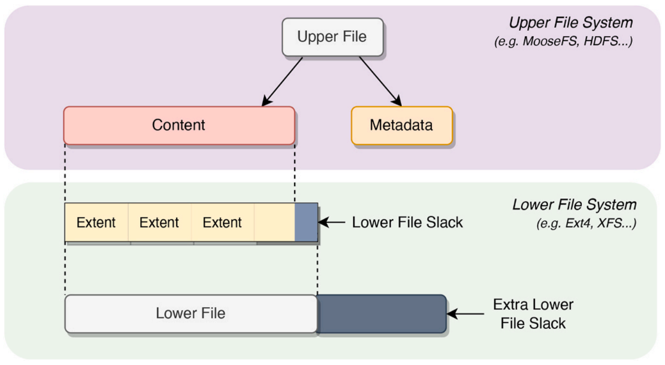


**Fig. 6.** Overview of possibilities for slack within stacked file systems.

### 7.1. *Findings*

This section is divided into two parts: the first presents the findings related to the lower file slack space, while the second focuses on the extra lower slack space resulting from expanding the size of a lower file.

#### 7.1.1. *Lower file slack*

*7.1.1.1. MooseFS.* Chunks start with a `0x2000` byte header, followed by the upper file's content in `0x10000` byte blocks. The final `0x1000` bytes of the chunk header store CRC checksums: four bytes for each block, accommodating up to 1024 blocks. This results in the maximum chunk size of 64 MiB, plus the header size. Given the large block size, MooseFS's lower file slack can be used to hide up to 64 KiB of data without altering the chunk size. Data hidden here doesn't affect the upper file's accessibility or its displayed size. However, inserting data causes a mismatch of the CRC checksums, which led to the chunk marked as INVALID upon server restarts during our experiments. For effective concealment, it's crucial to update these checksums. Furthermore, modifying the upper file doesn't affect the data hidden in the lower file slack. However, if the file expands, reducing the chunk's slack, the concealed data is overwritten.

*7.1.1.2. GlusterFS.* In distributed and replicated mode, GlusterFS does not utilize any padding and thus the stored lower files are of the exact same size as the corresponding upper files. For this reason, there is no available slack space that can be exploited for data hiding. In dispersed mode, it was observed that the size of a resulting lower file is always a multiple of 512 bytes, theoretically creating slack space that could be used to hide data. However, due to the implemented Erasure coding algorithm, the position and amount of the padding that can be used to reliably hide data varies. Hiding data in the wrong part could lead to modified data within the upper file sometimes even displaying the hidden data in our experiments.

*7.1.1.3. eCryptfs.* For eCryptfs, the minimum file size of a file stored on the lower file system was always 12 KiB, which includes the 8 KiB header. Its size is then increased in steps of 4 KiB, as this is the *default extent size* used by eCryptfs. The actual extent size can also be found within the header at the start of the lower file. If the data size is not a multiple of the extent size, padding is used and also encrypted. When adding data to this lower file slack, it is still possible to mount eCryptfs and access the file without any problems. For the default extent size, this results in roughly 4 KiB of lower slack space that can be used to hide data. However, as soon as the upper file is modified, the whole contents of the padding is rewritten and the hidden data is lost.

#### 7.1.2. *Extra lower file slack*

*7.1.2.1. MooseFS.* With stacked file systems, data can also be hidden in the extra lower file slack by appending it to an existing lower file. Since MooseFS utilizes a maximum size for its lower files, it is also ensured that storing data past this offset is protected from being overwritten due to any modifications of the upper file. In our experiments, we filled the space up to the maximum size with zeros and placed data in the succeeding space. Subsequent modifications to the upper file did not overwrite the hidden data in the extra lower file slack. However, as soon as a chunk was transferred to another chunk server, the hidden data did not persist and was lost.

*7.1.2.2. GlusterFS.* For GlusterFS, hiding data in extra lower file slack proved impractical across all modes. In distributed and replicated modes, data added to the lower file also appears when reading the corresponding upper file, though the upper file size remains unchanged. For replicated volumes with only one replica containing hidden slack data, the upper file consistently reveals this data. When multiple replicas have extra slack data, the upper file reads from the largest lower file. To hide data, one might place it in any replica, but ensure another copy has more benign data, like null bytes. In dispersed mode, data concealed in the slack of one file in a three-brick setup vanished upon reading the upper file, while adding data to two lower files caused an I/O error.

*7.1.2.3. eCryptfs.* In eCryptfs, an upper file remained accessible with its file size unchanged when the data was stored in the corresponding extra lower file slack. Notably, this appended hidden data persisted when the upper file was modified or when new data was added, provided it did not surpass the padding limit. If the file grew beyond the available padding, the appended data was overwritten. To avoid this, one can add ample padding before the hidden data, ensuring that any growth of the original file only replaces this 'dummy' padding, thereby preserving the hidden data.

### *7.2. Key takeaways*

Stacked file systems differ from traditional ones in that slack space does not contain remnants of previous files, primarily because new lower files are created for each new upper file. However, our findings suggest that exploiting slack space in lower files, or even in additional lower file slack, could be a viable tactic in certain stacked file system implementations. Consequently, forensic practitioners should not only focus on the upper file system but also thoroughly examine the lower file system during their analyses.

Detecting file slack requires a detailed comparison between the file sizes recorded in the upper file system and those of the corresponding lower files. Additionally, cross-referencing replicas of lower files across various lower file systems is critical to identify any discrepancies that may indicate tampering or manipulation.

## 8. Possibilities for file recovery

Besides operating system or application specific concepts such as trash bins, file deletion is completely file system specific. Some file systems such as older versions of Ext may keep references to the actual data blocks, while others may wipe these entirely. In these experiments, we circumvented the operating system's Trash bin by directly deleting files from the stacked file system using the `rm` command. This approach allowed us to examine the file deletion processes of the stacked file systems in question, thereby identifying the potential methods available for file recovery.

### *8.1. Findings*

#### *8.1.1. MooseFS*

MooseFS's own trash mechanism holds deleted files for 24 h by default. When an upper file is deleted, it becomes inaccessible, but its chunks in the lower system persist. These deleted files are labeled as *trash files* in the `NODE` metadata section on the Master Server. Furthermore, the Change timestamp of these deleted upper files stored in the metadata can be used to infer the time of deletion. Notably, even with a trash duration set to zero, chunks stayed active for a couple of minutes. During this time the upper files got tagged as *sustained files* in the metadata indicating they were deleted but still open. The Change timestamp of these files can hint at their deletion time. Once a file was fully deleted, its chunks were too.

#### *8.1.2. GlusterFS*

In its default configuration, GlusterFS does not utilize its *Trash translator* feature. Thus, as soon as an upper file is deleted, the corresponding files in the lower file system are removed as well and the possibilities of recovery depend on the lower file system. Enabling this feature results in the creation of a `.trashcan` directory on each of the bricks, which is used to hold deleted upper files and is also mounted within the upper file system. After the deletion, the GFID-named lower file remains intact, while the hard link in the original hierarchy is removed from the lower file system. Instead, a new hard link within the `.trashcan` directory is created, whose name consists of the original upper file's name and the actual time of deletion. Furthermore, the original path hierarchy of the deleted file is also recreated within the trash directory.

#### *8.1.3. eCryptfs*

Following a file deletion within eCryptfs, the corresponding lower file was also deleted instantaneously in our experiments.

### *8.2. Key takeaways*

Our research reveals that stacked file systems can offer an extra opportunity for file recovery through their own trash features. Understanding the specific structure and metadata of the stacked file system is key, and the data from these trash bin mechanisms should be factored into the analysis process.

Investigators should consider the new opportunities presented by the presence of an additional lower file system. Even if content is deleted from the upper file system, the original data might still exist as lower files within the lower file system. While file recovery becomes wholly dependent on the lower file system following a complete file deletion, the inherent structure of these lower files can be utilized for advanced recovery techniques, such as file carving. In summary, these findings imply that acquiring the lower file system, either physically or logically, is more advantageous than merely performing a simple logical acquisition of the upper file system.

## 9. Conclusion

Contrary to traditional file systems, the concept of stacked file systems utilizes an additional file system for data storage. Given its integration into various modern distributed file systems, encountering stacked file systems is inevitable in present and future forensic investigations. In this paper, we focused on the forensic analysis of stacked file systems and presented an updated model that is capable of handling this class of file systems. Complementing this, we presented various forensic implications based on traditional analysis techniques and explored them using three representative stacked file systems as examples.

Our findings reveal that understanding the architecture, mechanisms, and features of stacked file systems is crucial for effective analysis. We demonstrated basic procedures like identification and metadata extraction in our findings, noting that further research is essential for a more comprehensive understanding of these systems. The significance of the underlying file system was also emphasized, particularly its potential to enhance investigations with finer details, such as more precise timestamps. Notably, even when access to the upper file system itself is hindered, for example by encryption or incomplete distributed structures, valuable data can still be retrieved from the lower file system.

To fully leverage these insights, it is imperative for current forensic methodologies and tools to adapt. Our research lays a solid groundwork for future exploration in this area and aims to increase awareness among forensic investigators regarding the complexities and opportunities presented by stacked file systems.

## Acknowledgement

We thank our shepherd and our anonymous reviewers for their invaluable feedback on this paper.

## References

Aranya, A., Wright, C.P., Zadok, E., 2004. Tracefs: a file system to trace them all. FAST 129–145.
Asim, M., McKinnel, D.R., Dehghantanha, A., Parizi, R.M., Hammoudeh, M., Epiphaniou, G., 2019. Big data forensics: Hadoop distributed file systems as a case study. Handbook of Big Data and IoT Security 179–210.
Bhat, W., Quadri, S., 2012. Restfs: secure data deletion using reliable & efficient stackable file system. In: 2012 IEEE 10th International Symposium on Applied Machine Intelligence and Informatics (Sami). IEEE, pp. 457–462.
Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Professional.
Garfinkel, S.L., 2007. Carving contiguous and fragmented files with fast object validation. Digit. Invest. 4, 2–12.
Göbel, T., Türr, J., Baier, H., 2019. Revisiting data hiding techniques for apple file system. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, vols. 1–10.
Halcrow, M.A., ecryptfs, 2005. An enterprise-class encrypted filesystem for linux. Proceedings of the 2005 Linux Symposium 1, 201–218.
Harshany, E., Benton, R., Bourrie, D., Glisson, W., 2020. Big data forensics: Hadoop 3.2. 0 reconstruction. Forensic Sci. Int.: Digit. Invest. 32, 300909.
Heidemann, J.S., Popek, G.J., 1994. File-system development with stackable layers. ACM Trans. Comput. Syst. 12 (1), 58–89.
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.
Huebner, E., Bem, D., Wee, C.K., 2006. Data hiding in the ntfs file system. Digit. Invest. 3 (4), 211–226.
Martini, B., Choo, K.K.R., 2014. Distributed filesystem forensics: Xtreemfs as a case study. Digit. Invest. 11 (4), 295–313.
Raghavan, S., 2013. Digital forensic research: current state of the art. Csi Transactions on ICT 1, 91–114.
Sipek, J., Pericleous, Y., Zadok, E., 2007. Kernel support for stackable file systems. In: Proc. Of the 2007 Ottawa Linux Symposium, vol. 2. Citeseer, pp. 223–227.
van der Meer, V., Jonker, H., van den Bos, J., 2021. A contemporary investigation of NTFS file fragmentation. Forensic Sci. Int.: Digit. Invest. 38, 301125.
Wani, M.A., Bhat, W.A., Dehghantanha, A., 2020. An analysis of anti-forensic capabilities of b-tree file system (btrfs). Aust. J. Forensic Sci. 52 (4), 371–386.
Zadok, E., 1999. Stackable File Systems as a Security Tool. Tech. Rep.; Citeseer.

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS APAC 2024 - Selected Papers from the 4th Annual Digital Forensics Research Conference APAC

# Mount SMB.pcap: Reconstructing file systems and file operations from network traffic

Jan-Niclas Hilgert [*], Axel Mahr, Martin Lambertz

*Fraunhofer Institute for Communication, Information Processing and Ergonomics FKIE, Fraunhofer FKIE, Zanderstr. 5, 53177, Bonn, Germany*

ARTICLE INFO

*Keywords:*
File systems
Network forensics
File extraction
Digital forensics
Server message block

ABSTRACT

File system and network forensics are fundamental in forensic investigations, but are often treated as distinct disciplines. This work seeks to unify these fields by introducing a novel framework capable of mounting network captures, enabling investigators to seamlessly browse data using conventional tools. Although our implementation supports various protocols such as HTTP, TLS, and FTP, this work will particularly focus on the complexities of the Server Message Block (SMB) protocol, which is fundamental for shared file system access, especially within local networks.

For this, we present a detailed methodology to extract essential file system data from SMB network traffic, aiming to reconstruct the share's file system as accurately as the original. Our approach goes beyond traditional tools like Wireshark, which typically only extract individual files from SMB transmissions. Instead, we reconstruct the entire file system hierarchy, retrieve all associated metadata, and handle multiple versions of files captured within the same network traffic. In addition, we also investigate how file operations impact SMB commands and show how these can be used to accurately recreate user activities on an SMB share based solely on network traffic. Although both methodologies and implementations can be applied independently, their combination provides investigators with a comprehensive view of the reconstructed file system along with the corresponding user activities extracted from network traffic.

## 1. Introduction

File system analysis, as described by Brian Carrier in 2005, is a fundamental part of any forensic investigation (Carrier, 2005). It involves the analysis of a given file system, including its structures, to recover deleted files, extract metadata such as timestamps, or harness certain specific features such as journals or snapshots (Kim et al., 2012; Hilgert et al., 2018). In certain scenarios, performing file system analysis may not be practical, for instance, when there is no physical access to the device or when critical files on persistent storage have already been modified or deleted. In these instances, the use of network traffic can help bridge this gap.

In general, network forensics deals with a multitude of tasks, such as the identification of relevant IP addresses, the analysis of protocols, and, consequently, the extraction of data. Since data can be transferred over the network in arbitrary ways, there is no universal solution for file extraction, and dedicated methods must be implemented to deal with transferred files. Besides network protocols supporting file transfer, such as HTTP, SMB or FTP, the rise of distributed file systems has resulted in more and more file systems utilizing a network for data sharing, either by building on top of existing protocols or by implementing their own. Consequently, many file system artifacts can be present within captured network traffic.

Currently, standard tools such as Wireshark[1] provide only limited possibilities to deal with and analyze these files in transit. Typically, they only support their extraction from the network capture. However, we found that in most cases, more information valuable for forensic investigations such as file system hierarchies, timestamps, or other metadata is contained within these transmissions, which is usually neglected.

For this reason, this work aims to close the gap between file system and network forensics. In this research, we focus on the Server Message Block protocol, which is extensively used for file transfers on the Windows operating system. SMB is frequently used within local corporate

* Corresponding author.
*E-mail addresses:* jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert), axel.mahr@fkie.fraunhofer.de (A. Mahr), martin.lambertz@fkie.fraunhofer.de (M. Lambertz).

[1] https://www.wireshark.org.

https://doi.org/10.1016/j.fsidi.2024.301807

Available online 18 October 2024

networks, offering clients access to shared files and directories. Therefore, analyzing SMB is critical for reconstructing events in incidents such as ransomware attacks or data exfiltration. Although Wireshark is already capable of extracting files transferred via SMB from network captures, it does by no means harness all of the information available.

To address this, we created a methodology to recreate a file system representation from SMB network traffic, including the files' content and reconstructing its original hierarchy, timestamps, and other metadata. Moreover, we developed a framework that implements our methodology and is capable of mounting SMB network traffic as a file system. In addition to SMB, this framework also supports other network protocols, such as FTP and HTTP.

Complementing the reconstruction of a file system from network traffic, we take a closer look at the relationship between a user's actual file operations and the resulting SMB network traffic. Knowing and understanding this relationship enables us to reconstruct user interactions from captured network traffic. In general, the contributions of our work are as follows.

- An analysis of the steps required to **reconstruct a file system** from SMB network traffic.
- A framework for **mounting SMB network traffic** as a file system, including its original hierarchy and metadata, which also supports FTP and HTTP Hilgert et al. (2024a).
- A novel method and implementation for **SMB Command Fingerprinting** used to reconstruct user file operations from SMB network traffic Hilgert et al. (2024b).

Section 2 will provide an overview of the fundamentals of the SMB protocol. In Section 3, we will show the steps necessary to reconstruct the original file system of the SMB share from captured SMB network traffic and present our implementation that allows investigators to mount network captures in Section 4. Afterwards, in Section 5, we will explore the possibilities of reconstructing actual file operations from captured SMB commands. Section 6 presents related work in this area, before we conclude in Section 6.

## 2. Server Message Block protocol

The SMB protocol versions 2 and 3 were introduced with Windows Server 2008; 2012, respectively and are described in Microsoft's specification Corporation (2024), which includes information about supported commands and parameters, as well as descriptions of the network packet structures for sending requests and responses. This section provides a basic overview of the SMB protocol to aid in understanding subsequent discussions on file system and file operation reconstruction.

*Packet Structure*. Every SMB request and response starts with a 64 byte SMB header that features a protocol identifier, flags (such as to indicate whether it's a request or response), and two bytes that denote the SMB command type. Compound requests or responses can be used to include multiple commands linked together in a single packet. In these instances, the header will contain an offset pointing to the subsequent 8 byte-aligned SMB header in the packet. Additionally, to correlate requests with their responses, each SMB header contains an 8 byte message ID.

Moreover, SMB headers include a 4-byte field that indicates the status of a response. In the case of requests, this field is disregarded and must be zeroed out. An exhaustive list of possible status codes is available in the [MS-ERREF] document by Microsoft. A status field filled with zeros denotes a successful response.

*Connection Setup*. All SMB dialects, that is versions, support direct TCP as their transport protocol, typically using port 445 on the server side. Dialect 3.1.1 also introduces support for QUIC. Initially, the client sends an `SMB2 NEGOTIATE` request to inform the server of the SMB dialects it supports. The server then selects its preferred dialect for subsequent communications in its `SMB2 NEGOTIATE` response. This is followed by `SMB2 SESSION_SETUP` requests and responses to establish an authenticated session, which include key details about the domain, host, and user name used within the session. To access a specific server share, the client sends `TREE_CONNECT` messages with the full path of the share. If successful, the `TREE_CONNECT` response provides the tree ID, which is used in the SMB header for subsequent requests related to this share.

*Commands*. In the SMB protocol specification, Microsoft lists several commands that fall under the *File Access* category of SMB messages. The most important ones for the upcoming sections will be introduced next.

- `SMB2 CREATE` requests are used to request access or the creation of a file or directory. It includes 4 Bytes to specify the desired access, given in the SMB2 Access Mask encoding. Additionally, it also contains flags to indicate what actions the server should take, if the file already exists, further options relevant for opening or creating the file as well as file attributes given in the [MS-FSCC] specification by Microsoft. The response to a `SMB2 CREATE` request contains information about the status of the operation, e.g. success as well as create, last access, last write and change timestamps of the file. It also returns a 16 Byte FileId, which is used to identify the accessed or created file in subsequent requests.
- `SMB2 CLOSE` requests are sent by a client to close an opened file or directory by specifying its FileId.
- `SMB2 READ` requests contain the FileId of the file a client wants to request data from. The request contains the offset as well as the length that should be read. Consequently, the response, if successful, contains the requested data.
- `SMB2 WRITE` requests work in a similar way and are used to write data of a certain length to a certain offset of a file, identified by its FileId. The successful response then contains the number of bytes that have been written.
- `SMB2 IOCTL` commands can be used by the client to issue file system (FSCTL) or device control (IOCTL) commands to the server over the network. A list of permitted FSCTL commands can be found in Section 2.3 of the [MS-FSCC] specification.
- `SMB2 QUERY_INFO` requests, known as `GetInfo` requests in Wireshark, are utilized to gather details about files, quotas, security, or the underlying storage system, based on the specified 1 Byte *Information Type*. Additionally, the specific information requested is determined by the 1 Byte *File Information Class*, such as `FileBasicInformation` for timestamps and attributes. When the information type is `SMB2_0_INFO_FILESYSTEM`, the response includes detailed information about the share's file system. Requesting the `FileFsAttributeInformation` class for instance would provide the file system's attributes and its name.
- `SMB2 SET_INFO` commands are used to update specific information on files and other objects. The details to be updated are defined by the information type and information class, along with the actual data to be applied. For instance, setting the `FileDispositionInformation` is used to mark files for deletion.
- `SMB2 QUERY_DIRECTORY` requests, known as `FIND` requests in Wireshark, are used to retrieve details about the contents of a directory. In addition to the FileId of the target directory and the specific information class to be returned, the request includes a Unicode search pattern, which can also be a wildcard. The server provides the requested specific information for each match to this search pattern.

In subsequent sections, we will use abbreviated forms of these commands, e.g. `CREATE` for `SMB2 CREATE`.

### 2.1. Create context

Within a `CREATE` request, the client can also include *Create Context Structures* to request additional information. Some common ones are.

- **Maximal Access Request (MxAc)**: In this request, the client requests the maximal access it has on the opened file or directory based on the current session. The response includes the corresponding access mask.
- **Query On Disk ID (QFid)**: If this is sent, the server responds with the corresponding 8 Byte FileID as well as the VolumeID to which the opened file belongs.
- **Request Lease V2 (RqLs)**: The client requests a lease for the opened file.

Leases were introduced in SMB 2.1 to enhance client-side caching, effectively replacing *OPLOCKs*. To utilize this feature, a client requests a lease for an opened file, specifying the desired mode—such as read, write, or handle cache. In response, the server grants the appropriate lease, allowing the client to cache reads and writes locally and thereby reduce network traffic associated with SMB operations. When a lease is broken — for instance, due to external changes in a directory for which a client has an open file handle — the server issues a *Lease Break Notification* to the client. The client must then act based on the lease's mode. For example, if a read cache lease is broken, the application is required to purge all cached data. More detailed information on lease breaks is available in the SMB specification.

## 3. File system reconstruction

In order to reconstruct a file system from network traffic, it is important to consider what data actually makes up a file system. According to Brian Carrier, the data of a file system belongs to one of the five data categories presented within his reference model Carrier (2005).

- **Metadata Category:** Metadata encompasses data describing files such as their timestamps or access rights.
- **File name Category:** File as well directory names and their relationship to each other are stored in this category, which is why it basically describes the file system *hierarchy*.
- **Content Category:** The actual content of files within the file system belongs to this category.
- **File system Category:** Data in this category defines the structure of the file system itself, e.g. its size or where other data is stored.
- **Application Category:** This category consists of all the data the file system does not necessarily need to read and write data, but is added for special features, e.g. journaling.

In the subsequent sections, we will outline our approach for data extraction from SMB network traffic corresponding to the previously mentioned categories of file system data. In addition, we will discuss certain peculiarities encountered during the reconstruction of a file system from SMB network traffic.

### *3.1. Metadata*

Most metadata, such as timestamps or file size, can be obtained from the corresponding `SMB2 CREATE` response. While it also includes file attributes, these do not necessarily match all attributes of the share's original file system. Instead, the file attributes used in SMB are detailed in [MS-FSCC] as mentioned earlier. To associate extracted information from subsequent requests with a specific file or directory, we also extract the 16 Byte FileId from the `CREATE` response, along with the corresponding TreeId, and store them in an internal mapping table.

`QUERY_INFO` requests and responses can provide additional metadata as this command is used to retrieve various types of file information. Timestamps and file attributes can be obtained from the `BasicInformation` class, while the `StandardInformation` class includes details such as the allocation size and the end-of-file value, which indicates the file's first unoccupied byte, i.e., its end. Further metadata can also be found in `QUERY_DIRECTORY` responses as described in the next subsection. Finally, metadata can also be extracted from `SET_INFO` requests targeting metadata like timestamps.

### *3.2. File names*

Our main method for obtaining file and directory names is through `CREATE` requests. These not only include the name of the requested file or directory but also its complete file path relative to the root directory of the share, which is derived from the `TREE_CONNECT` request, provided it is present in the network capture. This method enables us to reconstruct parts of the share hierarchy, including the parent directories of the requested file. However, this reconstruction is only performed when a corresponding and successful `CREATE` response is received, ensuring that only existing or newly created files are reconstructed.

Another crucial command for hierarchy reconstruction is `QUERY_DIRECTORY`. The output of this command typically includes matches to a specified search pattern. For standard interactions with the share, this pattern is usually set to the wildcard *. Consequently, the server returns all available files in a directory up to a specified buffer length. The details stored in the corresponding responses are then used to expand the file hierarchy. Additionally, depending on the query sent, this information contains at least the basic metadata for the files matching the pattern. Extracting files in this manner results in the creation of *hollow files* as described in Section 3.5. Similar to metadata, we also use `SET_INFO` requests to gather information about files that have been renamed.

### *3.3. Content*

File contents can mainly be retrieved leveraging `READ` and `WRITE` commands. To achieve this, we first identify all such command types and correlate them with the actual files by matching their FileIds against our internal mapping table. Then, we use the offset and length fields within the commands to accurately reconstruct the file content.

### *3.4. File system and application data*

Extracting information about the file system of the underlying share can be achieved through `QUERY_INFO` responses when a *File System Information Class* is requested. Section 2.5 of the [MS-FSCC] specification provides a detailed overview of the available classes and their corresponding data. In our upcoming experiments, we have primarily encountered requests and responses for the *FileFsVolumeInformation* and *FileFsAttributionInformation* classes. These classes provide details about the volume on which the file system is mounted, such as its creation date or serial number, and a list of attributes describing the file system, respectively. Since each file system has a unique layout and internal structures, the data on file system details in SMB network traffic does not allow for an exact replication of the original file system. This also applies to any data that belongs to the application category. However, as shown in the previous sections, this is not necessary for reconstructing the most critical data for forensic analysis.

### *3.5. Hollow files*

A *hollow file* is a file whose content does not appear in the SMB network traffic. Nevertheless, as mentioned previously, various SMB commands already contain extensive metadata, which we use to create a hollow file that includes the correct file name, path, attributes, and timestamps. This method aims to provide the most comprehensive view possible of the original file system on the share. If a corresponding `READ` or `WRITE` request for a hollow file is identified, we populate the file with its content, thereby making it a regular file. Fig. 1 shows an example of three SMB requests and responses and how we use their information to reconstruct the file system.

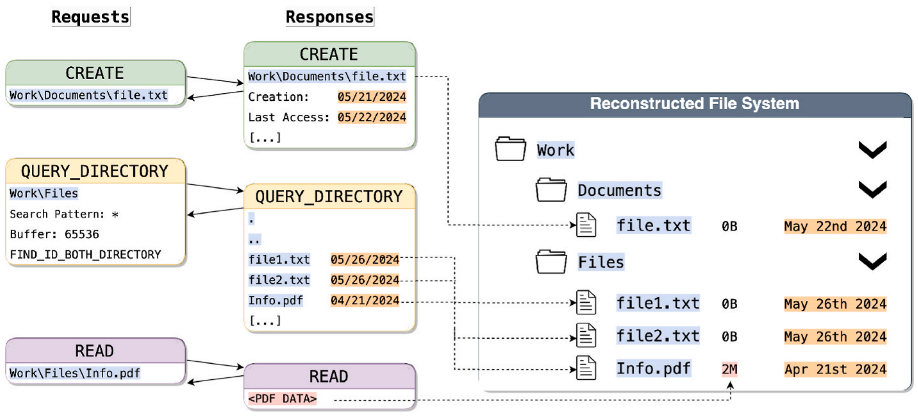


**Fig. 1.** Reconstructing file systems from SMB requests and responses.

### 3.6. *Version history*

Unlike traditional file systems, which typically provide a snapshot of files and directories at a specific point in time, network captures contain data over a continuous period. Consequently, the same file can be accessed multiple times during a capture period. If the file changes during this time, different versions, including older and more recent versions, may be present in the network capture. Since these previous versions might not be recoverable from persistent storage of the network share itself, it is crucial to extract these versions when reconstructing the file system.

To facilitate this, we monitor the timestamps associated with each file in our reconstructed file system. A change in these timestamps indicates a modification to the file. In such instances, we generate a new version of the file, denoted by appending ”@<version>” to its filename. It is important to note that new versions arise not only from files being read but also from write operations detected in the network traffic.

## 4. Mounting network traffic

After detailing the process of reconstructing an original file system from SMB network traffic in the previous section, this section outlines our implementation for mounting acquired network traffic to achieve such reconstruction.

Our approach extends traditional network forensics, which typically focuses on packet-level or protocol-level data analysis. Instead, we enable an analysis similar to traditional storage forensics, where investigators can navigate through network data using standard forensic tools and techniques. This includes operations such as calculating hashes, searching for YARA signatures, or employing other sophisticated tools.

Furthermore, our solution tackles a major challenge in network forensics: the performance drop due to the extensive size of network traffic captures, which can consist of countless packets and require lengthy loading periods in analysis tools such as Wireshark. This is achieved by utilizing a specialized index file that stores the layout of the reconstructed file system. This eliminates the need for repeated parsing and examination of the network capture upon mounting, thereby enhancing performance and accelerating the analysis process.

### 4.1. *Overview*

Our implementation utilizes the Filesystem in Userspace (FUSE)[2], which facilitates the creation of customizable and mountable file systems. Unlike traditional storage forensics, where a volume is mounted, we process network captures, supporting the PCAP and PCAPNG formats. We analyze and parse the information within these captures so they can be mounted and accessed as a regular file system. For this purpose, our implementation creates *virtual files* for each network protocol it supports, e.g., TCP or SMB files. As outlined in the previous section, this involves extracting content, metadata, and filenames and integrating these components using the methods provided by libfuse. Naturally, our file system is read-only and thus does not allow writing or altering the data.

To enhance our implementation's modularity, we utilize a recursive approach to analyze various network protocols within network captures. In a first run, virtual files are created for the network capture files themselves. Then, other protocols, typically TCP and UDP, are parsed within these files and new corresponding virtual files are created. These virtual files contain a set of offsets and lengths that point directly into the lower virtual file, as depicted in Fig. 2. When accessing data, such as reading a TCP file, our implementation leverages these pointers to retrieve and assemble the data efficiently.

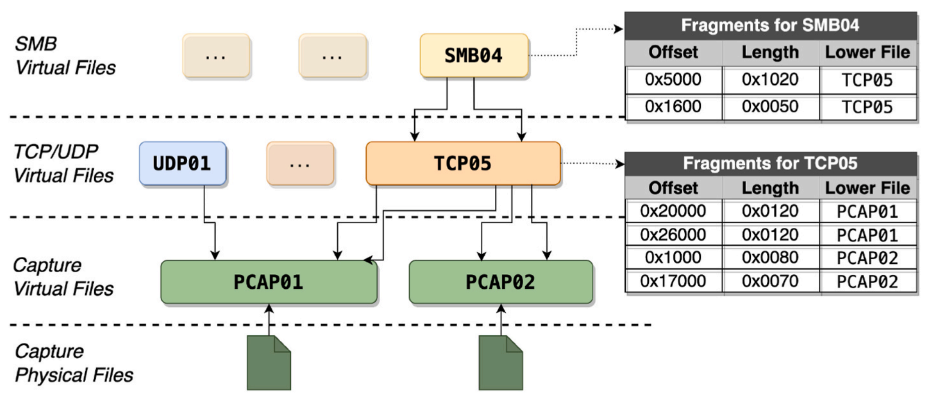
Similarly, for SMB files, pointers within the SMB file point to data in lower files, e.g., TCP files. Metadata for SMB files is extracted during an initial parsing step and then stored for each SMB file. Since this can be a time-consuming task, our implementation utilizes an *index file*, which stores all relevant information about the detected files, their set of offsets, as well as any metadata for these files and is typically only a fraction of the size of the associated network capture file.

Additionally, our implementation supports arbitrary transformation steps between virtual file layers. For instance, if data is encrypted, reading a virtual file may first access the encrypted data from a lower file, decrypt it—provided that decryption keys are available—and then present the decrypted data seamlessly in the mounted file system, maintaining transparency throughout the process. This concept allows for the support of more complex network protocols such as TLS.

### 4.2. *Structure*

By default, a separate directory is automatically created within the mounted file system for each supported protocol, in which the corresponding parsed virtual files are stored, as detailed in Listing 1. File names start with the index of the source file — for UDP and TCP files, which usually directly reference the network capture, the index remains uniformly '0' in our example, indicating a single `capture-file.pcap`. This index is followed by the offset at which the file begins. For example, `TCPFILE12` starts at offset 770 within the network capture file. This naming pattern also extends to other protocols, such as the HTTP `banner.svg` file, which points to TCP file 31 and starts at offset 22434. All necessary offsets for file construction are initially stored in memory, but can optionally be written to a special *index file* on disk to facilitate faster mounting by avoiding repeated data parsing.

[2] https://github.com/libfuse/libfuse.



| Fragments for SMB04 | | |
|---|---|---|
| Offset | Length | Lower File |
| 0x5000 | 0x1020 | TCP05 |
| 0x1600 | 0x0050 | TCP05 |

| Fragments for TCP05 | | |
|---|---|---|
| Offset | Length | Lower File |
| 0x20000 | 0x0120 | PCAP01 |
| 0x26000 | 0x0120 | PCAP01 |
| 0x1000 | 0x0080 | PCAP02 |
| 0x17000 | 0x0070 | PCAP02 |

**Fig. 2.** Overview of data access within our implementation for mounting network captures.

```
$ ./pcapfs capture-file.pcap /mnt/pcap
$ tree /mnt/pcap
.
├── udp
│   ├── 0-119_UDPFILE2
│   ├── 0-126_UDPFILE3
│   └── 0-98_UDPFILE1
├── tcp
│   ├── 0-770_TCPFILE12
│   ├── 0-797_TCPFILE13
│   └── 0-910_TCPFILE16
└── http
    ├── 10-311_wpad.dat
    ├── 29-4286_style-wide.min.css
    ├── 31-1142_init.min.js
    ├── 31-22434_banner.svg
    ├── 31-3300_style.min.css
    └── 43-1130_favicon.ico
```

**Listing 1.** Example hierarchy of a mounted network capture.

Depending exclusively on protocols to organize a file system hierarchy has a limitation: essential network capture artifacts like IP addresses are concealed from the investigator. To mitigate this issue, we introduced a `sortby` feature. This feature allows for the creation of a tailored hierarchy that includes critical details such as source and destination IPs or ports, as well as protocol-specific elements such as domains or URIs. Listing 2 shows an example where the hierarchy includes the source and destination IP addresses and the domain for the HTTP protocol. This approach elevates conventional network forensic filters to the filesystem level, improving both accessibility and usability for thorough analysis.

```
$ ./pcapfs capture-file.pcap /mnt/pcap
    --sortby=/protocol/srcIP/dstIP/domain
$ tree /mnt/pcap
└── http
    ├── 146.190.62.39
    │   └── 192.168.178.85
    │       └── httpforever.com
    │           ├── 29-4286_style-wide.min.css
    │           ├── 31-1142_init.min.js
    │           ├── 31-22434_banner.svg
    │           ├── 31-3300_style.min.css
    │           └── 43-1130_favicon.ico
    └── 192.168.178.1
        └── 192.168.178.85
            └── wpad
                └── 10-311_wpad.dat
```

**Listing 2.** The `-sortby` parameter can be used to create arbitrary hierarchies for the mounted network capture.

### 4.3. *Mounting SMB*

When mounting SMB network traffic, our implementation organizes the data by creating a directory for each detected SMB server and subdirectories for each share, or tree. If a `TREE_CONNECT` request is detected, the share is named using the provided name; otherwise, it uses the TreeId. Listing 3 illustrates an example in which a network capture of the `SMBSHARE` server is mounted. The `test_share` represents a user-defined share, whereas `IPC$` is a default share created by Windows to facilitate anonymous user activities such as share enumeration.

```
$ ./pcapfs smb.pcap /mnt/pcap
$ tree /mnt/pcap
├── smb
    └── SMBSHARE
        ├── IPC$
        [...]
        └── test_share
            ├── Documents
            │   ├── file1.txt
            │   ├── file2.txt@0
            │   ├── file2.txt@1
            │   ├── file2.txt@2
            │   └── file3.txt
            ├── Images
            │   ├── Animals
            │   ├── file10.txt
            │   ├── file1.JPG
            │   ├── file2.JPG
            │   ├── file7.JPG
            │   └── Landscape
            ├── Other
            └── Work
```

**Listing 3.** Example for mounted SMB traffic.

As shown for `test_share`, the share's hierarchy is reconstructed as previously detailed. Using the parameter `–show-metadata` during mounting, hollow files are enabled and displayed in lighter orange, offering a detailed representation of the SMB share including file names, hierarchy, and metadata such as file timestamps.

The file `file2.txt`, highlighted in darker green, contains actual data from the network capture. Our approach also handles the reconstruction of multiple file versions within the capture, as demonstrated by the three versions of `file2.txt` in the mounted share. Common file system tools such as `ls` can be utilized to retrieve metadata, helping to determine the timestamp of each file version.

While presenting multiple file versions as multiple files already addresses the dynamic characteristic of data in network captures, we have further enhanced our implementation with a `snapshot` feature. This feature can be invoked using the `snapshot` argument with the `-sortby` option, adding a new layer of directories to the hierarchy. This structure mimics snapshots in traditional file systems, enabling investigators to access and navigate the file system as it appeared at specific moments in time. This functionality is particularly useful for tracking changes like renames or deletions, which prompt the creation of a new snapshot — essentially a new directory within this layered hierarchy.

To offer a more complete understanding of the reconstructed file system data, it is crucial to comprehend its origins by extracting file operations from network traffic. The subsequent section introduces a novel methodology for identifying user activities within SMB network traffic. This approach can be utilized in conjunction with mounting the network capture for a more comprehensive view or separately.

## 5. Reconstructing file operations

File operations are any interaction an application has with a file or directory. As a result, there is a strong connection between file operations and user interactions, since every user interaction may initiate a series of file operations. This section details how analyzing captured SMB network traffic can provide insights into file operations and, by extension, the underlying user interactions.

### 5.1. Methodology

For this purpose, we divide the process into three steps.

1. **Windows API Analysis:** We begin by examining the influence of various Windows API calls on the resultant SMB commands. The Windows API offers a diverse set of functions that enable applications to interact with the Windows file system, playing a crucial role in all file operations within Windows.
2. **SMB Command Fingerprinting (SCF):** Building on our understanding of the Windows API, we propose a novel technique to detect the execution of a Windows API call on an SMB share, exclusively through the analysis of intercepted network traffic.
3. **Case Study with `cmd.exe`:** To demonstrate the effectiveness of our approach, we employ *SCF rules* to reconstruct specific user interactions, starting with the widely used command line utility, `cmd.exe`. This tool is selected for its ubiquity, simplicity, and versatile file system manipulation capabilities.

For our experiments, we used two systems running Windows 11 Pro Build 22621.3155, configured as an SMB share and an SMB client, respectively. We captured their network traffic using Wireshark and further analyzed application behavior through the `frida-trace`[3] utility to track the API calls made by applications.

### 5.2. Windows API

The Windows API offers a wide array of functions for various tasks including data access, system management, and networking. Functions within the Windows API that handle character data typically appear in three forms: a variant ending in `A` that utilizes Windows code pages for text processing, a variant ending in `W` that accommodates Unicode, and a basic form without suffix. Given that the standard form ultimately relies on one of these specific API calls, our emphasis will be on the more contemporary W-versions of these APIs where relevant. The following subsections will detail the SMB commands observed when we executed a compiled C version of the single Windows API call.

#### 5.2.1. CreateFile

Since many Windows API methods require a file handle, it is often necessary to first open the file using the `CreateFile` call. In addition to the file name, it requires the desired access and share mode, the creation disposition, and flags or attributes as arguments. These arguments thus need to be adapted to the actual use case, e.g. a read or write.

In our experiments, we have found that the arguments given to the `CreateFile` API call can highly influence the resulting SMB commands sent via the network. For this reason, we present the most crucial results from our experiments.

- Calling the `CreateFile` API call results in at least one `CREATE` request.
- The specified *file share access*, *create disposition* and *file attributes* are reflected in the corresponding fields of the `CREATE` request.
- *File attributes* do not influence the sequence of SMB commands sent.
- Similarly, *file flags including the security flags* did not change the SMB commands sent. Instead, most of the file flags are represented in the create options field within the `CREATE` request.
- The desired *share mode* does not have an impact on the sequence of SMB commands either.
- The API parameters `OPEN_EXISTING`, `OPEN_ALWAYS`, `CREATE_NEW` and `CREATE_ALWAYS` for the disposition are mapped to the `FILE_OPEN`, `FILE_OPEN_IF`, `FILE_CREATE` and `FILE_OVERWRITE_IF` dispositions in SMB commands.
- Using `OPEN_ALWAYS`, `CREATE_NEW` or `CREATE_ALWAYS` as a disposition adds an additional `CREATE` request to the sequence of SMB commands targeting the parent directory.
- If write access is requested in an API call using the `OPEN_EXISTING` disposition, an additional `QUERY_INFO` requesting the normalized name of a file is issued.

#### 5.2.2. FindFirstFile

This API call is used to search a directory for a specific file name or pattern, including a wild card, and returns a search handle for subsequent searches, as well as the file information for the first matching file. Performing this call on an explicit file name or directory name results in a `CREATE` command for its *parent directory* followed by two `QUERY_INFO` commands, which were sent as a compound request in our

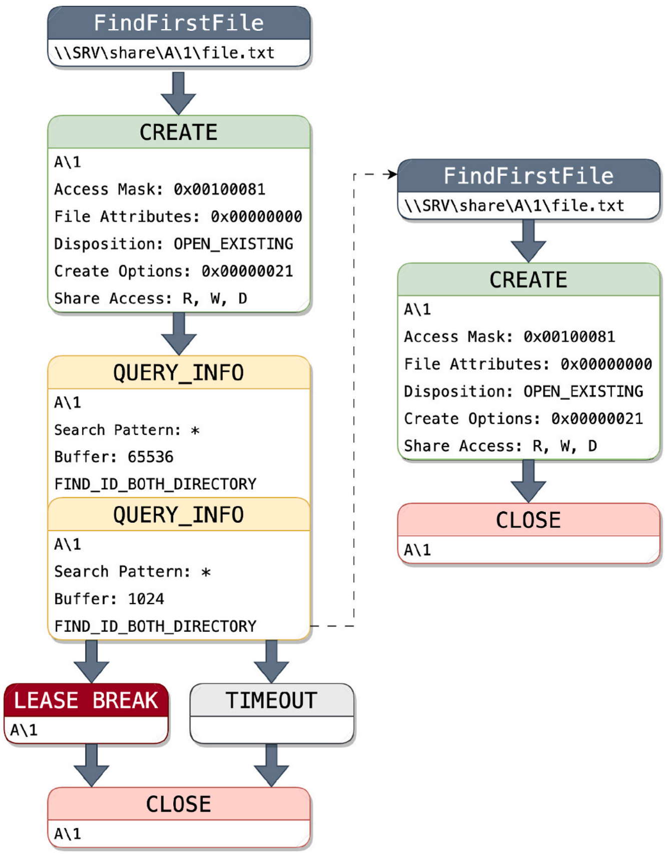


**Fig. 3.** SMB commands triggered by a `FindFirstFile` API call. The right side illustrates the outcomes when the call is made with the prior file handle remaining open.

[3] https://frida.re/docs/frida-trace/.

experiments, as illustrated in Fig. 3. If the client receives a response indicating a NO_MORE_FILES status after the initial QUERY_INFO requests, it does not perform additional ones. However, if the server still returns files, it performs additional QUERY_INFO requests using a buffer length of 1048612 Bytes until all files are returned. Most interestingly, regardless of the search pattern specified in the API call, the SMB commands seem to always utilize the wild card parameter *, thus returning information for all files within a directory.

Since the CREATE also requests a lease, the file handle is kept open and the CLOSE request is only sent when the DormantDirectoryTimeout is reached, which by default is set to 10 min. Alternatively, the client also sends this command as soon as a *Lease Break Notification* for the directory is received. During the time the file handle is still open, performing this API call still creates the CREATE request as shown in the second example in Fig. 3. However, the QUERY_INFO commands are omitted in this case, and the CLOSE request is sent immediately.

The FindFirstFileEx call additionally allows us to specify attributes that need to match the returned file. In our tests, the resulting SMB commands were identical to the ones observed for FindFirstFile.

Subsequently, the FindNextFile API call is usually used to obtain the next file that matches the search pattern. This call requires the search handle returned by FindFirstFile. However, since this API function initially requests all matching and even nonmatching files through SMB, using FindNextFile does not trigger any additional commands in the network compared to just using FindFirstFile.

### 5.2.3. *GetFileAttributes and SetFileAttributes*

The GetFileAttributes API call is a straightforward method to retrieve the attributes of a file or directory based on its name, eliminating the need for a preceding CreateFile call. This operation triggers a single CREATE command for the target file, with the parameters in the SMB packet set automatically. This is immediately followed by a CLOSE command.

Conversely, to modify file attributes, the Windows API offers the SetFileAttributes call, which requires a file name and the new attributes to apply. Following the CREATE request, which employs parameters distinct from those for attribute retrieval, a QUERY_INFO command is issued to obtain FileNormalizedNameInfo. Subsequently, attribute modifications are made using a SET_INFO command directed at FileBasicInfo. The sequence ends with a CLOSE command.

### 5.2.4. *ReadFile*

To read a file, the Windows API provides the ReadFile function, which requires a file handle and a specified number of bytes to read. For our experiments, we obtained the file handle by performing the CreateFile API call with standard GENERIC_READ settings, yielding SMB commands as detailed in Section 5.2.1.

When ReadFile is called, it causes an additional READ command. In particular, the number of bytes requested in the SMB command is always rounded up to the nearest multiple of 4096 or the actual file size of the file, if it is lower. For example, an API call to read only 50 bytes of a large file will actually request 4096 bytes over SMB. For read operations that exceed 2,097,152 bytes, multiple READ requests are issued, using the offset parameter to request the next parts of the file. These requests are transmitted consecutively without pausing for the server's response.

While the ReadFile function lacks a direct parameter to set a read offset, this can be accomplished by adjusting the file pointer using the SetFilePointer function. This adjustment also affects the offset utilized in the SMB READ commands. Similarly to the read length, any specified offset is rounded down to the nearest multiple of 4096 bytes in the respective READ command.

### 5.2.5. *WriteFile*

Writing data to a file in the Windows APIs is performed using the WriteFile function. This function requires three key inputs: a file handle, a pointer to the buffer containing the data, and the number of bytes to write. The file handle must be obtained first, with the correct access rights set for writing. For our experiments, we created the handle using GENERIC_WRITE and OPEN_ALWAYS.

The WriteFile operation itself triggers two additional SMB requests: A WRITE request, which includes the actual data to write, follows the CREATE response for the target file. If the data length exceeds 3,473,408 bytes, the operation is handled through multiple WRITE requests. These requests utilize the length and offset fields in the SMB commands to indicate which part of the data is sent. Once the write operation is complete, a QUERY_INFO command is issued to retrieve the FileNetworkOpenInfo, which provides details about the file status post-write.

### 5.2.6. *CreateDirectory and RemoveDirectory*

The API calls CreateDirectory and RemoveDirectory are intended for creating and deleting directories, respectively.CreateDirectory generates a single CREATE request followed by a CLOSE request. As illustrated in Fig. 4, invoking RemoveDirectory initiates a CREATE request, which is then succeeded by a SET_INFO request that sets the FileDispositionInformation to explicitly mark the directory for deletion. Finally, a CLOSE request is issued.

### 5.2.7. *DeleteFile*

The DeleteFile API call requires the name of the file to be deleted. The resulting SMB commands are similar to those of the RemoveDirectory command. However, the parameters for the CREATE request are different, and there is an additional QUERY_INFO command issued to retrieve the FileNormalizedNameInformation class. This command sequence is illustrated in Fig. 4.

## 5.3. *SMB Command Fingerprinting*

Our research has shown that each Windows API call generates a distinctive sequence of SMB commands. These sequences are

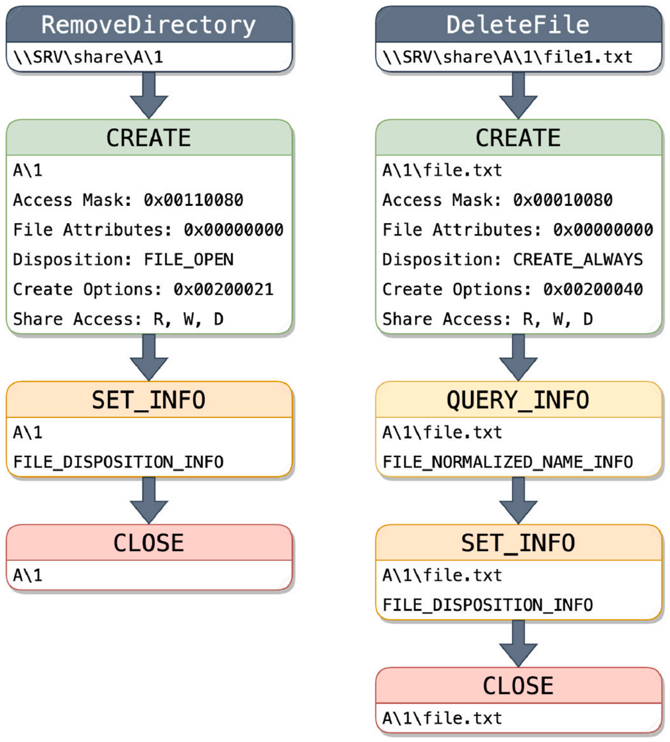


**Fig. 4.** SMB commands originating from a RemoveDirectory and DeleteFile API call.

characterized by two key aspects: the specific types of SMB commands issued and the parameters set within these commands. This is because API calls that require a filename generally initiate a file operation using unique parameters, such as file attributes or desired access levels. We can use this information to associate SMB command sequences with their respective API calls.

To facilitate the analysis, we propose an SMB Command Fingerprinting approach. This method calculates an MD5 hash for each SMB packet, simplifying the identification of distinct command sequences. To ensure that the hashes are both precise and universally applicable, i.e., independent of dynamic fields, we selectively hash values based on the specific type of SMB command. Fig. 5 illustrates which parameters are used to calculate an SCF for an SMB Create request.

For compound requests containing multiple SMB requests or responses in a single SMB packet, we calculate the individual SCF for each SMB command, concatenate them, and calculate the final hash of this result. To facilitate this process, we developed a utility that calculates the SCFs for SMB packets in a given network capture file automatically. Examples of SCFs resulting from various Windows API calls can be found in Table 2.

## 5.4. *Case study:* `cmd.exe`

While reconstructing specific Windows API calls from SMB network traffic yields valuable insights, the true strength of our approach lies in reconstructing explicit user interactions. For this purpose, we developed a set of *SCF rules* that comprise one or more SCFs. Thus, these rules consider not only the individual parameters of an SMB command but also the sequence in which these commands are sent. Listing 4 provides an example of an SCF rule. This rule identifies a sequence that includes a CREATE request, a successful CREATE response, a WRITE request, and a QUERY_INFO request, while considering the specific parameters for creation, as well as the information type and class through the use of SCFs.

```
{
    "application": "cmd",
    "description": "Create a file with echo",
    "command": "echo >",
    "rule": [
        "ebe5eb76fabfe0e41eb63d4fbd06bcd1" ,
        STATUS_OKAY_SCF ,
        "d4b9e47f65b6e79b010582f15785867e" ,
        "80c2cc1529acacebb810ec4014119967"
    ]
}
```

**Listing 4.** An SCF rule for the detection of file creation using e.g. `echo “Data” > file` in SMB network traffic.

To demonstrate the practicability of this methodology, we utilized

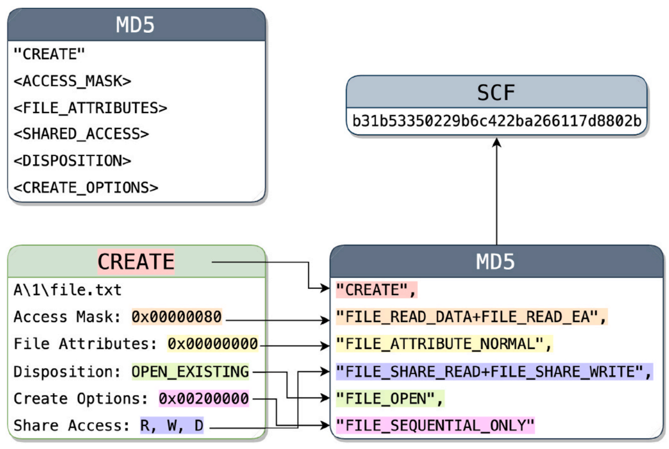


**Fig. 5.** Example calculation of an SMB2 Create Command Fingerprint.

the Windows Command Line Utility `cmd.exe`, a ubiquitous tool across Windows systems that can be used to perform various file operations. We executed a series of commands on a mounted SMB share, captured the corresponding network traffic, and used our SCF rules to reconstruct the file operations. Although only 18 commands were executed over a span of about 2 min, the generated SMB traffic included roughly 250 SMB packets, making a manual analysis impractical and unscalable. Table 1 provides a comprehensive summary of the events that were automatically reconstructed purely from network traffic, together with the original commands executed and their timestamps.

Our findings show that our approach successfully identified 16 of 18 commands executed using `cmd.exe`. The exceptions were the `cd ...` commands, which did not generate SMB commands, likely due to caching mechanisms, hence they could not be reconstructed. All other commands, including other simple directory changes, were accurately reconstructed. Notably, the `mkdir Files\Other` command was reconstructed as two separate events, reflecting the recursive nature of directory creation in this scenario.

It is important to note that the timestamps of reconstructed events typically lag behind the actual execution times due to the inherent delay in capturing the corresponding network packets. Therefore, the precision of these timestamps in real-world scenarios can vary significantly depending on the network configuration.

## 6. Related work

Over the years, multiple research efforts have explored methods to facilitate network traffic analysis. In digital forensics, research has, for example, explored the extraction of HTTP traffic events (Gugelmann et al., 2015). Other studies range from employing neural projections for the visualization of network traffic for intrusion detection (Corchado and Herrero, 2011), incorporating 3D representations that integrate the temporal aspect (Clark and Turnbull, 2020), to using relational graphs for enhanced data exploration (Cermak et al., 2023). A 2021 study emphasized the difficulties in using visualization techniques for anomaly detection in network traffic, highlighting the persistent challenges in this research area (Corchado and Herrero, 2011).

File extraction from network traffic is a well-established practice, with current methods capable of extracting various file types from different network protocols, similar to the functionalities provided by tools such as Wireshark (Choi et al., 2015; Hansen et al., 2018). However, these methods either do not support or are inadequate in extracting and presenting all available information for protocols like SMB.

Although initially not developed with digital forensics in mind, a conceptually similar approach to our SMB Command Fingerprinting already emerged in 1992. The researchers introduced a toolkit to approximate the activity of the file system by analyzing the network traffic of the NFS (Network File System) (Blaze, 1992). Over the years, various research on NFS tracing has evaluated and refined these methods, enhancing the ability to recover file system traces from passive monitoring of network traffic (Moore, 1995; Ellard and Seltzer, 2003). However, this concept has not been extended to protocols like SMB, nor has it aimed to establish a universally applicable set of rules to detect diverse user actions across different applications, as we propose with our SCF Ruleset.

Furthermore, the broader domain of network traffic fingerprinting has traditionally focused on identifying specific applications rather than user interactions (Dai et al., 2013; Taylor et al., 2016). Our research tries to identify precise user behaviors, thus expanding the forensic capabilities of network traffic analysis.

## 7. Conclusion and future work

In this work, we introduced a novel method for network forensics by integrating it with traditional file system analysis. To achieve this, we created a framework enabling analysts to mount a network capture file, allowing them to navigate the data and use standard file-based tools easily. Our implementation facilitates this process by offering various options for customizing the file system hierarchy, such as using IP addresses or ports, thereby merging network and file system forensics.

Although our framework supports multiple protocols that can be mounted, including HTTP, FTP, and TLS, we particularly emphasized the SMB protocol due to its common use for file sharing. We have outlined a methodology for extracting critical data from SMB network traffic, which can be used to accurately reconstruct the file system of the share. Unlike other analysis tools such as Wireshark, which only allow for the extraction of transferred files, our approach enables the reconstruction of the file system hierarchy and metadata by leveraging all available information in the captured traffic. Therefore, using *hollow files* that lack actual data, our method offers a more comprehensive representation of the original file system.

As an additional method for SMB network analysis, we examined the unique SMB sequences resulting from Windows API calls and proposed *SMB Command Fingerprinting*. This method enables the identification of Windows API call executions purely from SMB network traffic and the accurate reconstruction of user activities. For this purpose, we created *SCF rules* that allow the precise reconstruction of commands executed through the Windows command utility. While this was merely a case study to demonstrate the applicability of our approach, it is essential to expand on this research in the future.

For example, it is crucial to broaden the SCF ruleset to encompass other applications and explore the feasibility of distinguishing between them, such as determining which application was responsible for creating or deleting a file. In this context, it is also vital to examine different operating systems, considering various implementations of the SMB protocol, such as Samba on Linux. Furthermore, it is necessary to investigate whether failed attempts, such as unsuccessful file access, can be accurately reconstructed from SMB network traffic. To support research in this field, both our framework for mounting network traffic and our implementation for calculating SCFs, reconstructing file operations, and our current rule set are available in our repositories Hilgert et al. (2024a,b).

## Appendix

**Table 1**
Comparison of actual executed `cmd.exe` commands and the reconstructed commands from SMB network traffic.

| cmd.exe Timestamp | Command Executed | Reconstructed Timestamp | Reconstructed User Activity |
|---|---|---|---|
| 18:00:36.27 | `dir` | 18:00:37.02 | listing of directory (dir)/ |
| 18:00:47.79 | `mkdir Files\Other` | 18:00:47.93 | creation of directory (mkdir): Files |
| | | 18:00:47.93 | creation of directory (mkdir): Files\Other |
| 18:00:52.49 | `dir` | 18:00:52.59 | listing of directory (dir)/ |
| 18:01:00.00 | `cd Files` | 18:01:00.16 | changed directory (cd) to Files |
| 18:01:04.59 | `dir` | 18:01:04.75 | listing of directory (dir) Files |
| 18:01:12.36 | `cd` | | |
| 18:01:17.11 | `dir` | 18:01:17.26 | listing of directory (dir)/ |
| 18:01:23.28 | `mkdir Documents` | 18:01:23.51 | creation of directory (mkdir): Documents |
| 18:01:29.15 | `cd Documents` | 18:02:31.10 | changed directory (cd) to Documents |
| 18:01:36.35 | `echo "abcd" > test.txt` | 18:01:36.51 | creation of file using echo Documents\test.txt |
| 18:01:41.49 | `more test.txt` | 18:01:41.73 | view of file Documents\test.txt |
| 18:01:53.60 | `mkdir Work` | 18:01:53.80 | creation of directory (mkdir): Documents\Work |
| 18:01:58.18 | `dir` | 18:01:58.35 | listing of directory (dir) Documents |
| 18:02:07.24 | `echo "efgh" >> test.txt` | 18:02:07.46 | appending to file using echo Documents\test.txt |
| 18:02:13.96 | `cd` | | |
| 18:02:18.68 | `dir` | 18:02:18.83 | listing of directory (dir)/ |
| 18:02:24.04 | `ren docs old` | 18:02:24.18 | renamed (rename) directory docs to old |
| 18:02:30.89 | `del Documents\test.txt` | 18:02:31.11 | deletion of file (del): Documents\test.txt |

**Table 2**
SMB Command Fingerprints for various Windows API calls.

| WinAPI | SCF | Description |
|---|---|---|
| FindFirstFile | e128601506b19689cfea77f8e57fa33d<br>e6f1d54f04f3f80e8c008be45ddb89f1 | CREATE<br>Compound Request<br>QUERY DIRECTORY — QUERY DIRECTORY |
| GetFileAttributes | 3df084742fb18607089dd93e01da07bb | CREATE |
| SetFileAttributes | ec10dfc12cab368e93459f451fd6b2dc<br>56100944eac20b9e9e3229bee6916e1b<br>5a4606aac7612839c39162746e8655a0 | CREATE<br>QUERY INFO FileNormalizedNameInformation<br>SET INFO FileBasicInformation |
| CreateDirectory | cb7e84430eef9f80aa038dfa3679fd91 | CREATE |
| RemoveDirectory | 26b162f9c78b0d1095d94d55dfb9bc69<br>472410aa272671f7d8a103954703cc5a | CREATE<br>SET INFO FileDispositionInformation |
| DeleteFile | 17221fcc0857e79e60545bb409f37497<br>56100944eac20b9e9e3229bee6916e1b<br>472410aa272671f7d8a103954703cc5a | CREATE<br>QUERY INFO FileNormalizedNameInformation<br>SET INFO FileDispositionInformation |

(*continued on next page*)

**Table 2** (*continued*)

| WinAPI | SCF | Description |
|---|---|---|
| CopyFile (to server) | 901e16b20a7cf536d5279df8a06e2a0a | CREATE |
| | c10eadfd4fc2a82352888ea761f9ce54 | Compound Request<br>QUERY INFO — QUERY INFO |
| | 35127603ad78335a7290598c9070e7f7 | SET INFO FileEndOfFileInformation |
| | d4b9e47f65b6e79b010582f15785867e | WRITE |
| | 5a4606aac7612839c39162746e8655a0 | SET INFO FileBasicInformation |
| | 80c2cc1529acacebb810ec4014119967 | QUERY INFO |
| MoveFile (to server) | e7449ce5b9915ecfadc3293625d087ad | CREATE |
| | c10eadfd4fc2a82352888ea761f9ce54 | Compound Request<br>QUERY INFO — QUERY INFO |
| | 35127603ad78335a7290598c9070e7f7 | SET INFO FileEndOfFileInformation |
| | d4b9e47f65b6e79b010582f15785867e | WRITE |
| | 5a4606aac7612839c39162746e8655a0 | SET INFO FileBasicInformation |
| | 80c2cc1529acacebb810ec4014119967 | QUERY INFO |

## References

Blaze, M., 1992. Nfs tracing by passive network monitoring. In: Proceedings of the USENIX Winter 1992 Technical Conference, pp. 333–343.

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

Cermak, M., Fritzová, T., Rusňák, V., Sramkova, D., 2023. Using relational graphs for exploratory analysis of network traffic data. Forensic Sci. Int.: Digit. Invest. 45, 301563.

Choi, Y., Lee, J.Y., Choi, S., Kim, J.H., Kim, I., 2015. Transmitted file extraction and reconstruction from network packets. In: 2015 World Congress on Internet Security (WorldCIS). IEEE, pp. 164–165.

Clark, D., Turnbull, B.P., 2020. Interactive 3d visualization of network traffic in time for forensic analysis. VISIGRAPP 177–184, 3: IVAPP.

Corchado, E., Herrero, Á., 2011. Neural visualization of network traffic data for intrusion detection. Appl. Soft Comput. 11, 2042–2056.

Corporation, M., 2024. [ms-smb2]: Server Message Block (Smb) Protocol Versions 2 and 3. https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/5606ad47-5ee0-437a-817e-70c366052962.

Dai, S., Tongaonkar, A., Wang, X., Nucci, A., Song, D., 2013. Networkprofiler: towards automatic fingerprinting of android apps. In: 2013 Proceedings Ieee Infocom, IEEE, pp. 809–817.

Ellard, D., Seltzer, M., 2003. New nfs tracing tools and techniques for system analysis. In: Proceedings of the 17th Large Installation Systems Administration Conference. USENIX Association.

Gugelmann, D., Gasser, F., Ager, B., Lenders, V., 2015. Hviz: Http (s) traffic aggregation and visualization for network forensics. Digit. Invest. 12, S1–S11.

Hansen, R.A., Seigfried-Spellar, K., Lee, S., Chowdhury, S., Abraham, N., Springer, J., Yang, B., Rogers, M., 2018. File toolkit for selective analysis & reconstruction (filetsar) for large-scale networks. In: 2018 IEEE International Conference on Big Data (Big Data). IEEE, pp. 3059–3065.

Hilgert, J.N., Lambertz, M., Yang, S., 2018. Forensic analysis of multiple device btrfs configurations using the sleuth kit. Digit. Invest. 26, S21–S29.

Hilgert, J.N., Mahr, A., Lambertz, M., 2024a. pcapFS – Mounting Network Data. URL: https://github.com/fkie-cad/pcapFS/tree/main.

Hilgert, J.N., Mahr, A., Lambertz, M., 2024b. SCF - SMB Command Fingerprinting. URL: https://github.com/fkie-cad/SCF.

Kim, D., Park, J., Lee, K.g., Lee, S., 2012. Forensic analysis of android phone using ext4 file system journal log. In: Future Information Technology, Application, and Service: FutureTech 2012, vol. 1. Springer, pp. 435–446.

Moore, A.W., 1995. Operating System and File System Monitoring: A Comparison of Passive Network Monitoring with Full Kernel Instrumentation Techniques. Ph.D. Thesis. Monash University.

Taylor, V.F., Spolaor, R., Conti, M., Martinovic, I., 2016. Appscanner: automatic fingerprinting of smartphone apps from encrypted network traffic. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, pp. 439–454.

# References

[Car05] Carrier, Brian: *File System Forensic Analysis*. 2005.

[HLB24] Hilgert, Jan-Niclas; Lambertz, Martin & Baier, Daniel: "Forensic implications of stacked file systems". In: *Forensic Science International: Digital Investigation* 48, 2024, p. 301678.

[HLP17] Hilgert, Jan-Niclas; Lambertz, Martin & Plohmann, Daniel: "Extending The Sleuth Kit and its underlying model for pooled storage file system forensic analysis". In: *Digital Investigation* 22, 2017, S76–S85.

[HLY18] Hilgert, Jan-Niclas; Lambertz, Martin & Yang, Shujian: "Forensic analysis of multiple device BTRFS configurations using The Sleuth Kit". In: *Digital Investigation* 26, 2018, S21–S29.

[HML24] Hilgert, Jan-Niclas; Mahr, Axel & Lambertz, Martin: "Mount SMB.pcap: Reconstructing file systems and file operations from network traffic". In: *Forensic Science International: Digital Investigation*, 2024.

[HP94] Heidemann, John S & Popek, Gerald J: "File-system development with stackable layers". In: *ACM Transactions on Computer Systems (TOCS)* 12.1, 1994, pp. 58–89.

[Pal+01] Palmer, Gary et al.: "A road map for digital forensic research". In: *First digital forensic research workshop, utica, new york*. 2001, pp. 27–30.

# List of Figures