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

A R T I C L E I N F O

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

A B S T R A C T

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.

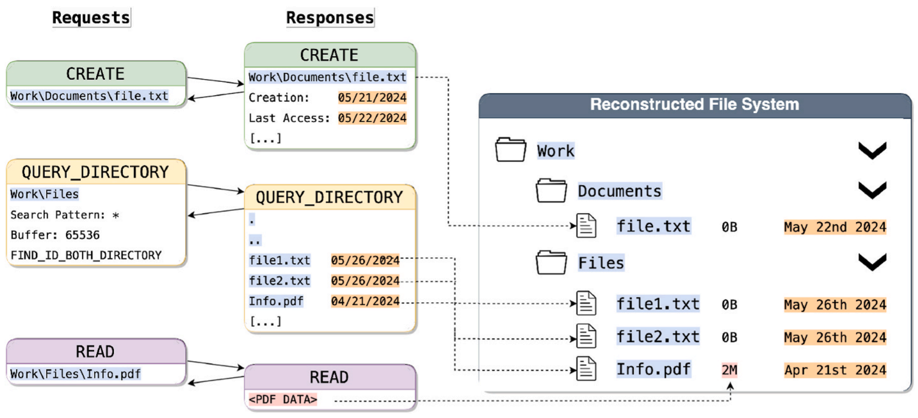


Fig. 1. Reconstructing file systems from SMB requests and responses.

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.

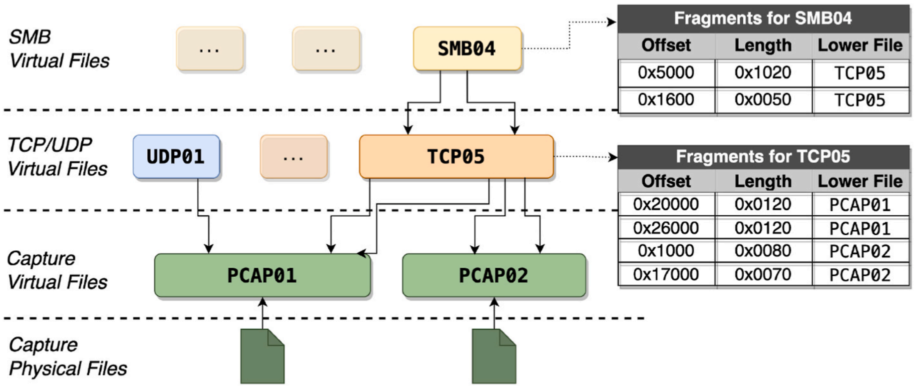
Similarly, for SMB files, pointers within the SMB file point to data in lower files, e.g., TCP files. Metadata for SMB files is extracted during an initial parsing step and then stored for each SMB file. Since this can be a time-consuming task, our implementation utilizes an *index file*, which stores 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

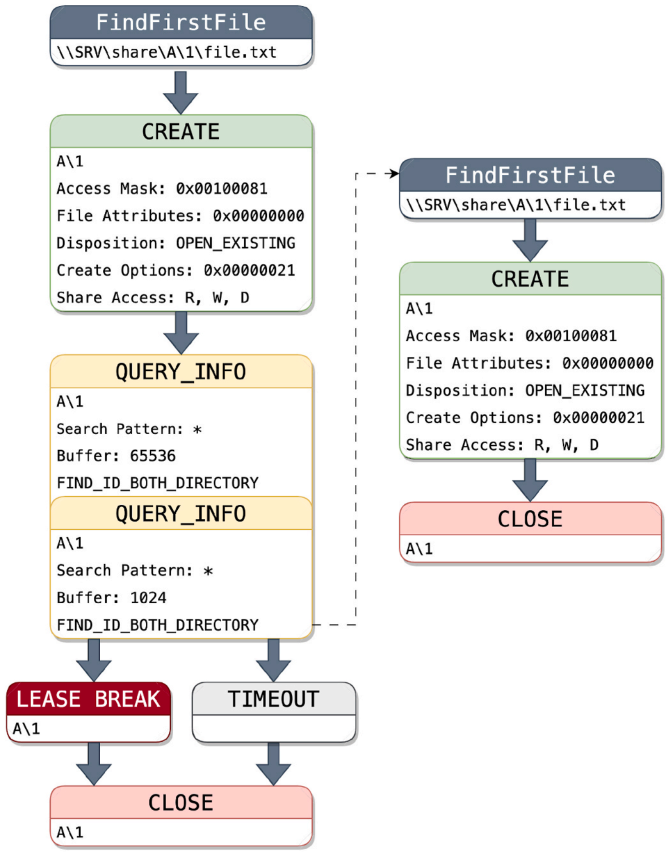


Fig. 3. SMB commands triggered by a `FindFirstFile` API call. The right side illustrates the outcomes when the call is made with the prior file handle remaining open.

[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

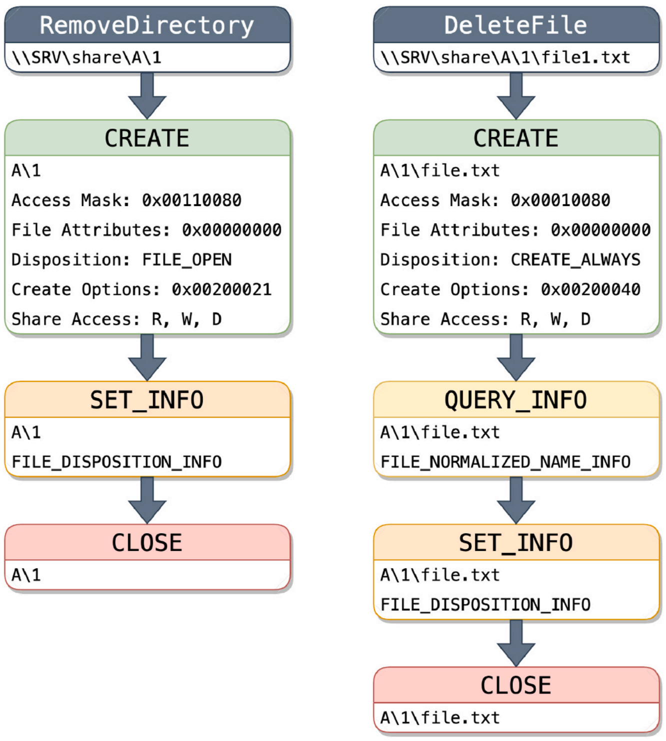


Fig. 4. SMB commands originating from a RemoveDirectory and DeleteFile API call.

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 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.

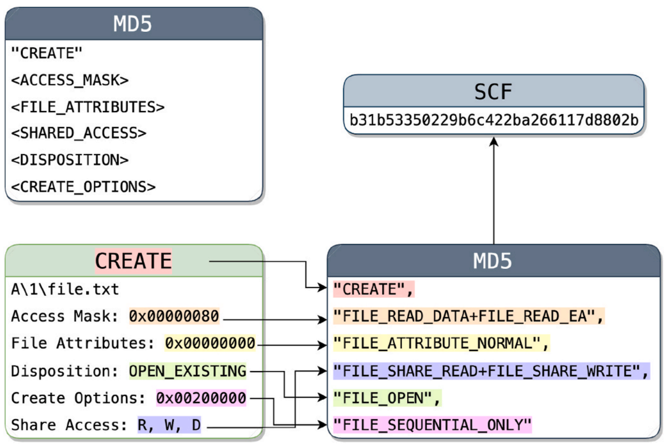


Fig. 5. Example calculation of an SMB2 Create Command Fingerprint.

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 e6f1d54f04f3f80e8c008be45ddb89f1	CREATE Compound Request QUERY DIRECTORY — QUERY DIRECTORY
GetFileAttributes	3df084742fb18607089dd93e01da07bb	CREATE

SetFileAttributes	ec10dfc12cab368e93459f451fd6b2dc
	56100944eac20b9e9e3229bee6916e1b
	5a4606aac7612839c39162746e8655a0
		CREATE
		QUERY INFO FileNormalizedNameInformation
		SET INFO FileBasicInformation

CreateDirectory	cb7e84430eef9f80aa038dfa3679fd91	CREATE

RemoveDirectory	26b162f9c78b0d1095d94d55dfb9bc69
	472410aa272671f7d8a103954703cc5a
		CREATE
		SET INFO FileDispositionInformation

DeleteFile	17221fcc0857e79e60545bb409f37497
	56100944eac20b9e9e3229bee6916e1b
	472410aa272671f7d8a103954703cc5a
		CREATE
		QUERY INFO FileNormalizedNameInformation
		SET INFO FileDispositionInformation

(*continued on next page*)

Table 2 (*continued*)

WinAPI	SCF	Description
CopyFile (to server)	901e16b20a7cf536d5279df8a06e2a0a	CREATE
	c10eadfd4fc2a82352888ea761f9ce54	Compound Request
		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
		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.