Generating Usable and Assessable Datasets Containing Anti-Forensic Traces at the Filesystem Level

Thomas Göbel, Harald Baier, and Jan Türr

Bundeswehr University, Munich, Germany
thomas.goebel@unibw.de

Abstract. Digital forensics and anti-forensics are essential to security because they provide vital information to institute preventive and reactive measures. Diverse and realistic datasets that reflect anti-forensic measures are needed to validate digital forensic tools and advance digital forensics education and research. However, datasets are increasingly created in a synthetic manner due to privacy and legal constraints.
The work described in this chapter contributes to improving the digital forensic process by assessing anti-forensic measures at the filesystem level and providing a means for synthesizing datasets containing anti-forensic artifacts. Specifically, it provides an in-depth analysis of anti-forensic data hiding techniques in the evolving Linux-based B-tree filesystem (Btrfs). Also, it presents a methodology for generating anti-forensic traces at the filesystem level in a *post mortem* storage device dataset. The methodology links the `ForTrace` data synthesis framework and `fishy` anti-forensic data hiding framework. A data synthesis tool is developed for generating anti-forensic data hiding traces for three common filesystems, NTFS, ext4 and Btrfs, and providing essential data synthesis functionality to simulate the expected behavior of the operating system. Additionally, a validation model comprising three complexity levels is presented for assessing the implemented anti-forensic data hiding techniques. Overall, the research provides a powerful approach for generating datasets that reflect anti-forensic artifacts potentially used by attackers.

Keywords: Data Synthesis · Data Generation · B-Tree Filesystem · Filesystem Forensics · Anti-Forensics · Data Hiding

1 Introduction

The increased reliance on Internet-connected devices and traditional computers in personal and business environments has led to significant increases in attack vectors, attacks and potential victims. Forensic analyses of attacked and manipulated systems are vital to implementing mitigations that minimize damage, attribute attacks and institute proactive measures to prevent future attacks. Education and research in digital forensics are therefore of high importance.

A key component for digital forensic education and research is the availability of diverse forensically-relevant data [30]. However, real data is often outdated, incomplete, unrealistic and/or not publicly accessible and, most importantly, often has missing labels [2,38]. Collecting and curating new and complete real-world datasets, especially for law enforcement purposes, is often not possible due to data protection laws [16]. To bypass privacy laws and other constraints on real-world data collection, several attempts have been made to generate synthetic forensic datasets [6,8,14,15]. In particular, efforts seek to facilitate data synthesis by simulating user interactions because manual data synthesis involves significant effort [8,14].

An accurate understanding of anti-forensic measures is important for training digital forensic practitioners and developing high-functionality forensic tools. Anti-forensic methods are designed to obscure, destroy and/or mislead digital forensic investigations [10,17]. A recent taxonomy by Conlan et al. [7] starts with the four original anti-forensic categories (data hiding, artifact wiping, trail obfuscation and attacks against computer forensic processes and tools) [32] and augments them with additional sub-categories of anti-forensic behavior. This work focuses on data hiding, one of the largest anti-forensic fields because it encompasses many types of manipulations with different levels of complexity [7, 24]. However, most data hiding algorithms and tools are often inaccessible due to missing source code and documentation or are focused on specific filesystems or data hiding techniques.

A promising approach is to deploy and maintain a modular and extensible framework capable of manipulating multiple filesystems, including a variety of data hiding techniques. Therefore, the work described in this chapter focuses on manipulating filesystem structures to hide data. Data hiding methods that leverage filesystem vulnerabilities attempt to obfuscate data using reserved, unchecked or unused parts of a filesystem without impairing filesystem functionality [3]. Sometimes, even data structures ordinarily used by a filesystem are abused to make it difficult to detect and reconstruct the hidden data. The goal is to develop an understanding of the extent to which attackers can conceal traces at the filesystem level.

This chapter provides an in-depth analysis of anti-forensic data hiding techniques in the evolving Linux-based B-tree filesystem (Btrfs). It presents a methodology for generating anti-forensic traces at the filesystem level by leveraging the `ForTrace` data synthesis framework and `fishy` anti-forensic data hiding framework. A data synthesis tool is developed for generating anti-forensic data hiding traces in three common filesystems, NTFS, ext4 and Btrfs, and providing essential data synthesis functionality to simulate the expected behavior of the operating system. Additionally, a validation model comprising three complexity levels, basic, specialist and white box levels, is presented for assessing the implemented anti-forensic data hiding techniques.

2 Related Work

This section discusses related work on anti-forensic dataset generation with a focus on data hiding and data synthesis tools.

2.1 Data Hiding Tools

Using filesystem structures to hide data is not a new idea. StegFS, a filesystem that employed steganographic channels to hide data, was introduced in 2000 [25]. In 2005, Eckstein and Jahnke [9] identified several methods for hiding data. Several early anti-forensic tools such as FragFS, MAFIA, `bmap` and RuneFS are either no longer accessible or poorly documented and only accessible via third-party publications [23].

Huebner et al. [20] have described methods for hiding data in NTFS filesystem structures. Berghel et al. [3] have investigated NTFS and Linux-based filesystems for potential hiding places. Göbel and Baier [12] and Neuner et al. [28] have discussed the data hiding potential of timestamps (especially nanosecond timestamps) in the ext4 and NTFS filesystems, respectively. Heeger et al. [18] have developed the exHide tool that implements multiple methods for hiding data in exFAT filesystem structures.

Abduhalil et al. [1] proposed a somewhat different variant of data hiding. Specifically, they duplicated files in NTFS using index entries. After deleting the files, the duplicate files remained in the filesystem.

Other researchers have analyzed the capabilities and vulnerabilities of Btrfs. Bhat and Wani [4] and Juch [22] partially base their analyses of Btrfs on the work of Rodeh et al. [31] and mention potential vulnerabilities and features of Btrfs. Wani et al. [37] provide an overview of potential data hiding methods for Btrfs. Gehrke [11] has studied Btrfs timestamps as a forensically-relevant structure. Schneider et al. [34] have shown that entire filesystems can be hidden in Btrfs.

2.2 Data Synthesis Tools

The early data synthesis tool Forensig^2 [26, 27] generates filesystem images by simulating user interactions via Python scripts. The EviPlant [33] framework is used to create forensically-relevant datasets. It produces a base system with a set amount of potential actions to create forensic artifacts. The differences between the base and manipulated systems are combined into an evidence package. The evidence package is then applied to the base system to create a forensically-relevant dataset. The VMPOP data synthesis framework employs virtual machines [29], which it populates with forensically-relevant artifacts by scripting interactions with the virtual machine interfaces.

The Python-based `hystck` framework [15] generates network and hard drive traces automatically using Python scripts or via YAML configuration files. Automated synthesis makes it possible to create a variety of traces with little effort

in virtual machines. In parallel to the synthesis process, a log file is created that contains and retains the ground truth of a scenario.

TraceGen [8] emulates user interactions to synthesize forensic artifacts. It employs scripts running on a virtual machine as well as scripts interacting with the virtual machine as in the case of VMPOP. Unfortunately, the source code of the TraceGen framework is not publicly available.

The Fade tool [6] synthesizes forensically-relevant Android device images. Specifically, it uses Android emulators to create forensic artifacts. Unfortunately, the Fade source code is also not available.

ForGeOSI (`github.com/maxfragg/ForGeOSI`) and ForGen (`github.com/Jjk422/ForGen`) are tools that also generate forensically-relevant images. ForGeOSI acts as a wrapper for the Python virtualization environment `pyvbox` and uses the connection to synthesize data whereas ForGen uses FTK Imager and VirtualBox to generate images.

Although there is consensus in the digital forensics community on the importance of synthetic datasets [2, 16, 38], combining a data synthesis approach with forensically-relevant anti-forensic techniques is relatively uncommon. This is curious because the resulting images would be good candidates for testing forensic tools and use in educational/training environments. Only the ForGe tool [35] is able to create NTFS images and enable data hiding in file slack or via steganographic channels. However, ForGe has not been updated since 2015.

3 Data Hiding in Btrfs

Btrfs is a filesystem that is gaining importance in Linux systems [5]. The origin of the acronym remains open; one is its B-tree directory representation whereas another is that it is simply a better filesystem. The following sections describe parsing Btrfs to find data structures that could hide data and discuss data hiding in Btrfs data structures.

3.1 Parsing Btrfs

As in the case of modern filesystems, it is necessary to find and understand the basic filesystem data structure of a Btrfs partition, i.e., superblock, to identify additional relevant filesystem structures.

Figure 1 presents the steps involved in parsing a Btrfs filesystem to find data structures that could be used to hide data. One component of interest is the filesystem interface that recognizes the filesystem and helps locate and interpret all the relevant filesystem structures. The second component comprises two chosen data hiding techniques, timestamp (nanoseconds) data hiding and file slack data hiding. These techniques are responsible for hiding, reading and removing hidden data as well as creating metadata objects and computing the checksums of filesystem structures changed by hiding or deleting data.

- **Step 0:** The filesystem of interest has to be identified as an instance of Btrfs. The filesystem type is checked by finding the expected superblock location

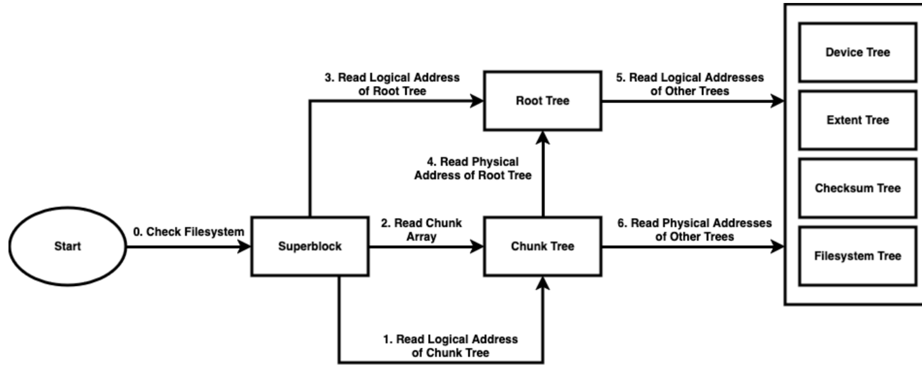

Fig. 1. Parsing Btrfs filesystem to find data structures for data hiding.

at offset 64 KiB in the partition and validating the superblock by checking its magic signature string `_BHRfS_M` at offset `0x40` of the superblock, i.e., at offset `0x10040` of the partition [5].

– **Step 1:** The superblock contains essential information about the filesystem, such as the size of the allocation units (blocks) and nodes that must be known in order to work with the filesystem. This is why the rest of the superblock is parsed. The chunk tree is an important data structure that is identified during this step. Btrfs mainly uses logical addresses and the chunk tree is used to translate the logical addresses to their physical addresses to access data structures. During Step 1, the logical address of the chunk tree at offset `0x58` of the Btrfs superblock is extracted.

```
1   POST http://android.clients.google.airpush.com/login
2   if logical_adr >= key_variable
3   AND logical_adr < key_variable + chunk_size:
4   physical_adr = stripe_adr
  + (logical_adr-key_variable)
```

Fig. 2. Pseudocode for the addressing process.

– **Step 2:** The chunk tree physical address is computed from its logical address and a partial copy of the chunk tree, which is stored at offset `0x32B` of the superblock. Figure 2 shows the pseudocode that translates logical addresses to physical addresses.

– **Step 3:** The logical address of the root tree is read from offset `0x50` of the superblock. The root tree is important because it contains the logical addresses of the entry points of all the other trees in Btrfs [19].

Table 1. Data fields of Btrfs inodes for hiding data [5].

Offset	Size (Bytes)	Description
`0x00`	8	Inode generation
`0x70`	8	Access timestamp
`0x78`	4	Access timestamp (nanoseconds)
`0x7c`	8	Change timestamp
`0x84`	4	Change timestamp (nanoseconds)
`0x90`	4	Modified timestamp (nanoseconds)
`0x9c`	4	Inode creation timestamp (nanoseconds)

- **Step 4:** Using the chunk tree located in Step 2, the logical address of the root tree is converted to its corresponding physical address, providing access to the root tree.
- **Step 5:** The root tree is analyzed and the logical addresses of all the other relevant trees are extracted.
- **Step 6:** As in Step 4, the physical addresses of the trees can be computed as needed. At this point, access is available to all the relevant Btrfs data structures.

3.2 Btrfs Data Hiding Techniques

This section analyzes the Btrfs filesystem data structures that can be leveraged to hide data. It begins with classic locations such as timestamps and slack areas and finally examines Btrfs-specific locations.

Btrfs Timestamp Hiding Technique. Inodes in Linux-based filesystems are the data structures that store metadata of filesystem objects such as directories and files. Table 1 shows the important data fields of Btrfs inodes for hiding data.

While the fine-grained structure of a Btrfs inode differs from the specifications of other filesystems, there are some similarities. A similarity with the ext4 filesystem is the presence of nanosecond timestamps. Unlike ext4 and APFS timestamps, which only allow 30 bits of the entire timestamp to be used to ensure the integrity of the rest of the timestamp [12], the Btrfs variant of the hiding technique can use the entire four-byte sub-second timestamp portion due to the separation of the timestamp parts shown in Table 1. However, manipulating the eight-byte main timestamp should be avoided because a simple filesystem check would detect a broken timestamp [11].

Figure 3 shows the workflow of the `write` function Btrfs timestamp hiding technique as implemented in the `fishy` anti-forensic framework. The filesystem is parsed as explained in Figure 1 to create a list of inode offsets. Additionally, the data input stream, whether from a file or the console, is read. For every entry in the list of inode addresses, 16 bytes of the input stream are read. If

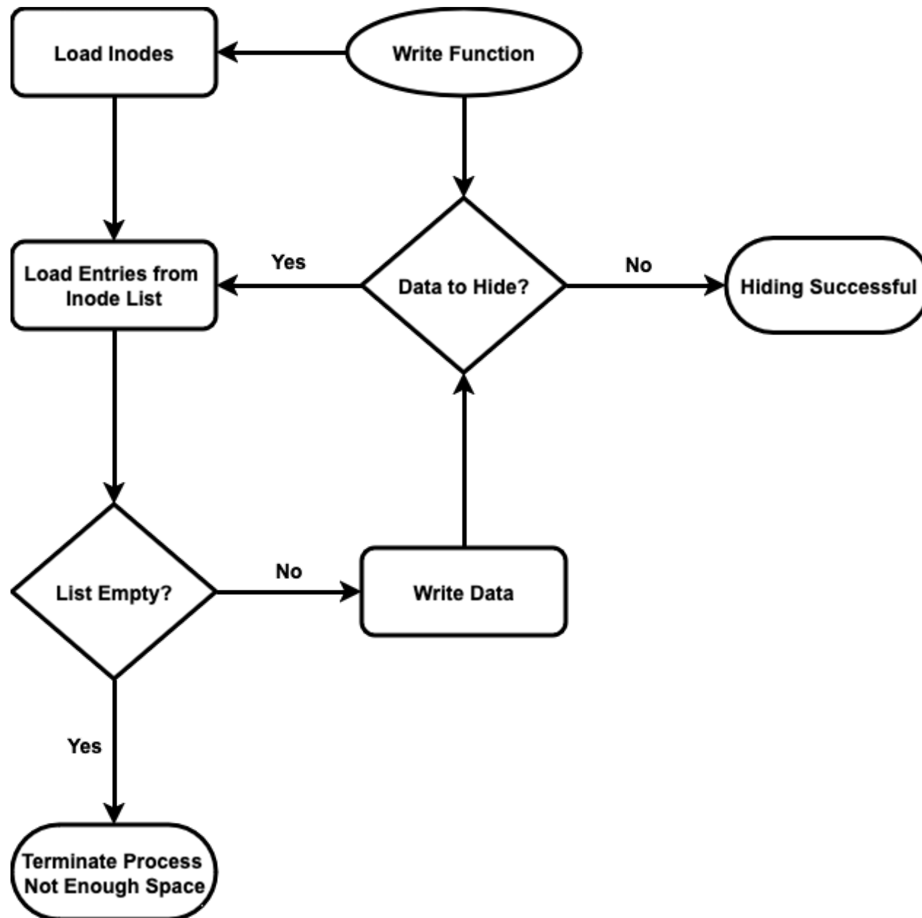

Fig. 3. Btrfs timestamp hiding technique workflow.

the list of inode addresses is empty, but the input stream has not yet ended, the process terminates due to the lack of hiding space. After writing data to the four timestamps of an inode, the new checksum corresponding to the inode is computed and replaced.

Btrfs File Slack Hiding Technique. A slack space is created when more space is allocated to an object than is needed. Different types of slack spaces arise in filesystems, for instance, between the end of the file content and its allocated space. Because the slack space does not store content, it may be used to hide data.

The workflow of the `write` function of the Btrfs file slack hiding technique is similar to that of the Btrfs timestamp hiding technique (Figure 3). The filesystem

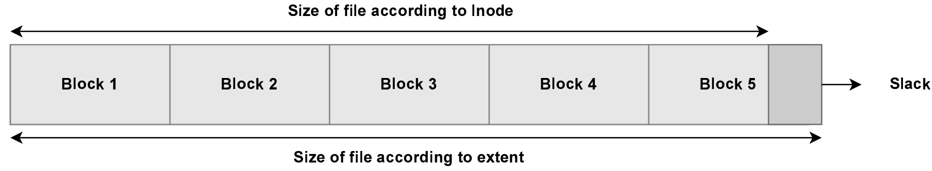

Fig. 4. Computation of Btrfs file slack space.

is parsed to create lists of extent and inode offsets. The lists are then used to find slack areas. If the lists are empty, but the input stream of data continues, the process terminates. Unlike the timestamp hiding technique, the potential hiding space existing in file slack space is more difficult to compute. The reason is that while timestamps offer a fixed amount of hiding space, file slack space is volatile and therefore has variable size.

Figure 4 illustrates the computation of Btrfs file slack space. The extent and inode structures tied to a file are required. The extent specifies the size of a file in terms of the number of allocated filesystem units (blocks). For example, the file in Figure 4 allocates five filesystem blocks of typical size 4 KiB = 4,096 bytes. Thus, the extent shows a file size of 5×4 KiB = 20 KiB = 20,480 bytes. However, as indicated by the respective inode, the actual file size may be any number of bytes between 16,385 and 20,480 bytes. The file slack size is simply the difference between 20,480 and 16,385 bytes. Similarly, the offset of the file slack location is computed by adding the file size found in the inode to the physical address of the file. The logical file address found in the inode is computed as described in Figure 2.

An additional challenge compared with timestamp-based hiding is updating the checksums in different Btrfs filesystem structures. First, the file content checksum in the checksum tree has to be recomputed and replaced. The entry is found using the starting position of file data located in the extent. Additionally, the file data checksum is contained in a node, similar to a node that contains multiple inodes. Therefore, after recomputing a file data checksum, the checksum of the corresponding node has to be computed and replaced as well.

Additional Btrfs Hiding Techniques. Ahead of every superblock in the filesystem is a 64 KiB unused space. This space is not traditional slack space, but it can be used in a similar manner. Even better, data hiding does not present the challenges posed by slack space hiding because the unused space never changes. However, the data hiding is easily detected. This is because of the limited number of superblocks in a filesystem and manual investigations commonly performed on important structures such as superblocks would detect the data hiding. In any case, since the area ahead of the superblock is usually empty, it could be paired with another hiding space [37].

Btrfs node structures also provide space for hiding data. Both types of nodes, internal nodes and leaf nodes, have default allocation sizes of 16 KiB and con-

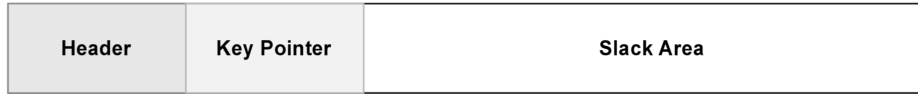

Fig. 5. Internal node slack area.

tain slack areas too. Internal nodes, shown in Figure 5, often only fill the first 1,000 bytes of the nodes to locate the node headers and the key pointers to child nodes, leaving up to 150,00 bytes of slack area if the node sizes are set to the default size of 16 KiB.

Fig. 6. Leaf node slack area.

Leaf nodes usually have smaller slack areas than internal nodes because they contain filesystem metadata. While the internal node slack is always located at the end of a node, the leaf node slack corresponds to the space between the node item metadata and the actual node item data (Figure 6).

The storage capacity of node-based hiding techniques increases with filesystem size – in depth (more files) as well as in width (more volumes) – because both options scale with the amount of metadata and, therefore, the number of nodes [36].

4 Dataset Generation with Anti-Forensic Traces

This section describes how anti-forensic data hiding practices are integrated in a data synthesis framework. The goal is to provide an easy-to-use method for generating datasets that contain hidden forensic traces and assessing the extent to which data hiding can actually be detected by forensic tools and digital forensic practitioners. The section describes the workflow and capabilities of the `fishy` filesystem anti-forensics framework [13] and the `ForTrace`data synthesis framework [14]. Following this, insights are provided into Btrfs data hiding techniques and the implementation of the interface between the two frameworks.

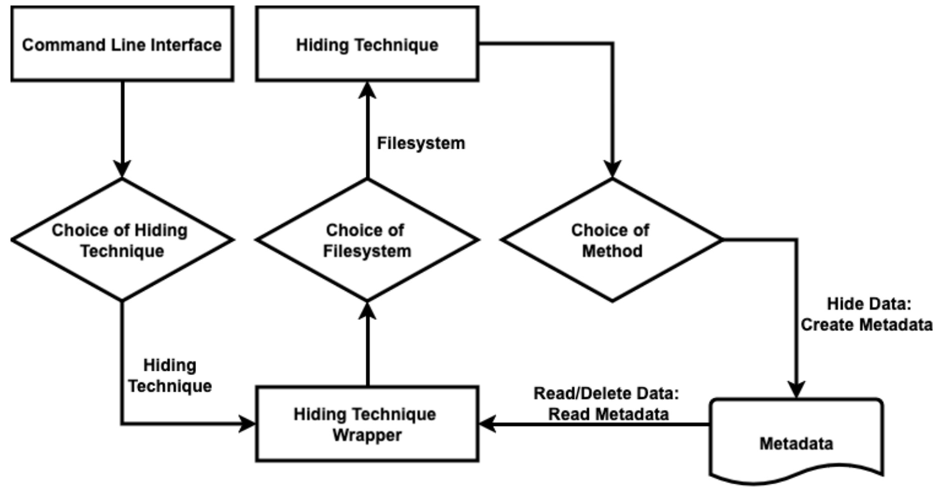

Fig. 7. `fishy` framework architecture.

4.1 `fishy` Framework

`fishy` [13] is a modular open-source framework that enables users to hide, read and delete data using reserved areas and other vulnerabilities in various filesystems. The framework was selected as the data hiding component for the data synthesis environment for several reasons. First, `fishy` is not restricted to specific anti-forensic methods or filesystems. In particular, the tool provides wrappers for common filesystems and also enables existing anti-forensic modules to be extended and new modules to be added (e.g., new filesystems and new hiding techniques). The framework is comprehensively documented and still maintained. Additionally, all the data hiding techniques offered by the framework have similar structures and are, therefore, easy to use. Also, the modular architecture of the framework facilitates the creation of a uniform interface.

Figure 7 shows a simplified version of the `fishy` framework architecture. The framework has three principal components that regulate the workflow:

- **Command Line Interface:** The command line interface parses user inputs and determines the target image, chosen techniques, methods and metadata options. As described later, when `fishy` is integrated with `ForTrace`, the command line interface is skipped because the new interface directly calls the second component, the hiding technique wrappers.
- **Hiding Technique Wrappers:** A wrapper exists for every implemented data hiding technique. The wrappers employ filesystem detectors to determine the implementations of the hiding techniques that need to be called. The wrappers also handle organizational tasks such as reading metadata files if they are needed by the called functions. Additionally, the wrappers handle input and output data streams.

- **Hiding Techniques:** Data hiding techniques are implemented in the corresponding filesystem modules. The filesystem modules parse filesystem structures and offer the requested information about them to the actual techniques to be used. Every hiding technique has three basic functions: `write` that inserts data to be hidden (Figure 3), `read` that recovers hidden data and `clear` that wipes hidden data using zero bytes.
  Additional functions can be implemented. Most modules also provide an `info` function that displays the corresponding filesystem and data hiding technique information.

Before the integration of the new Btrfs module, `fishy` had modules for the NTFS, ext4, FAT32, exFAT and APFS filesystems. Note that the FAT32, exFAT and APFS modules are not included in the data synthesis integration effort described in this chapter.

- **NTFS Module:** This module parses the common NTFS Windows filesystem. The NTFS module has four implemented hiding techniques. Two of the techniques, badcluster and addcluster, manipulate filesystem cluster management. The third technique computes the offsets and sizes of slack areas in master file table (MFT) records. The fourth technique hides data in the ordinary file slack area of a regular file (as in the case of Btrfs, this technique is present in all the `fishy` modules except for the APFS module). The filesystem interface and parser are realized by employing The Sleuth Kit and its Python bindings `pytsk3` (`pypi.org/project/pytsk3`) and `Construct` (`construct.readthedocs.io/en/latest`) to detect and model the filesystem structures.
- **ext4 Module:** This module parses the common ext4 Linux and Android filesystem. In addition to the almost universal file slack hiding technique, the ext4 module supports five hiding techniques. One uses nanosecond locations in the inode tables to hide data [12]. Other techniques leverage other areas in the inodes (such as the `osd2` and `obso_faddr` fields), backup copies of group descriptor tables and slack areas connected to filesystem superblocks. Unlike the NTFS module, the ext4 module employs an additional internal filesystem parser in combination with the `pytsk3` library.

All the `fishy` modules relevant to the interface implementation are presented in more detail in Section 4.3. Table 2 shows all the hiding techniques that can be used via the new interface as part of the data synthesis framework `ForTrace`.

4.2 `ForTrace` Framework

`ForTrace` [14] is a data synthesis framework based on `hystck` [15]. As shown in Figure 8, the architecture supports the simulation of human-computer interactions by leveraging a client-server architecture connected through a private network so that the collected network traffic is not affected by `ForTrace`-specific orchestration artifacts [14].

Table 2. Integrated `fishy` and `ForTrace` data hiding techniques.

Hiding Technique	NTFS	Btrfs	ext4	Description
`file slack`	✓	✓	✓	Uses file slack
`mftslack`	✓			Uses master file table entry slack
`addcluster`	✓			Adds clusters to files for hiding data
`badcluster`	✓			Marks bad clusters for hiding data
`reserved_gdt_blocks`			✓	Uses reserved group descriptor table blocks
`superblock_slack`			✓	Uses superblock slack
`osd2`			✓	Uses `osd2` entries in inodes
`obso_faddr`			✓	Uses `obso_faddr` entries in inodes
`nanoseconds`		✓	✓	Uses nanosecond parts of timestamps

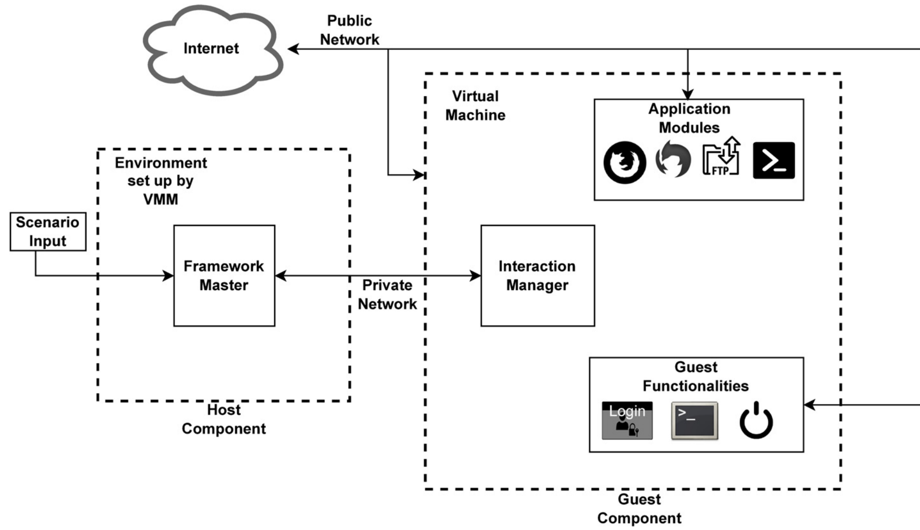

Fig. 8. `ForTrace` framework architecture [14].

The server-side components handle organizational tasks. The Virtual Machine Monitor (VMM) class generates the client-side components by cloning a previously installed and configured template. Additionally, sockets for communications are created and maintained by the class. The Guest class handles the actual communications between the server and client components. It transfers commands and parameters to the client. This component is called the framework master in Figure 8. The client side is a virtual machine cloned by the Virtual Machine Monitor class that is referred to as the interaction manager in Figure 8. It executes the client-side modules corresponding to the commands sent by the Guest class and returns status updates to the server components.

The `ForTrace` workflow is briefly summarized as follows. First, the Virtual Machine Monitor class clones a template. Next, the client and server components attempt to connect over the private network. Following this, the created scenario

	Slot	Start	End	Length	Description
000:	Meta	0000000000	0000000000	0000000001	Primary Table (#0)
001:	-------	0000000000	0000002047	0000002048	Unallocated
002:	000:000	0000002048	0000104447	0000102400	NTFS / exFAT (0x07)
003:	000:001	0000104448	0124781196	0124676749	NTFS / exFAT (0x07)
004:	-------	0124781197	0124782591	0000001395	Unallocated
005:	000:002	0124782592	0125825023	0001042432	Unknown Type (0x27)
006:	-------	0125825024	0125829119	0000004096	Unallocated

Fig. 9. Typical image partitioning created by `ForTrace`.

is communicated to the client. The client executes all the necessary actions. After completing the scenario, a network dump and an optional memory dump are created. A hard disk image with all the new traces is also created. Finally, the client is shut down. If specified, the client is deleted.

4.3 Data Synthesis Framework Integration

The integration of the two frameworks provides the ability to invoke `fishy` functions directly without having to invoke `fishy` from a different console while attempting to create a highly realistic image simultaneously using all the `ForTrace` data synthesis functions. Like the existing framework components, the new interface is also constructed in a modular manner, meaning that module extensions and new modules can be added easily using the new specifications. The usual first step of interaction with `fishy` is skipped because the interface directly invokes the hiding technique wrappers with the appropriate parameters.

The new interface is designed to fulfil three goals:

- The principal functionality of the new interface is that `ForTrace` can invoke the `fishy` hiding techniques. `fishy` could have been implemented like other `ForTrace` modules, but this would leave additional framework artifacts on the guest machine, which must always be avoided in a data simulation environment. Instead, the new interface is implemented as a exclusive server-side utility that can be used like regular modules during data synthesis.
- Another key functionality is that service on the synthesized virtual machine need not be interrupted. This is important because disruption of service due to the use of data hiding techniques would constitute high detectability of manipulation. To achieve this goal, changes were made in `fishy` to allow for specific starting point offsets, which are also included in every function in the filesystem parsers and hiding techniques. Additionally, checks are implemented to see how running guest machines react to the use of hiding techniques.
- Compatible hiding techniques must work despite the presence of additional partitions. By default, `fishy` only works with single filesystem partitions. However, virtual machines created by `ForTrace` have additional partitions such as the additional default Windows reserved partition (Figure 9). The hiding techniques would attempt their workflows starting at the first marked partition, resulting in immediate failure. Therefore, the actual offset of the

correct partition is added to every offset computation in `fishy` to find the correct partition and filesystem structure.

The interface implementation involved three steps. First, `fishy` was adapted so that the potential additional offset of a specific partition in an image could be interpreted and added to all the necessary functions. This change was realized by adding an additional parameter to all the functions that compute offsets in the filesystem.

Second, `fishy` was integrated with `ForTrace` by implementing a utility class that allows direct invocations of hiding technique wrappers to support running filesystems, i.e., the operations of a manipulated virtual machine are not disturbed by hiding data. When called, the class has to be initialized with the path to the synthesized image and the partition offset containing the filesystem the user intends to manipulate. Third, an additional change to `ForTrace` was an optional second template and client-side component creation process. `ForTrace` previously only created `qcow2` images. However, to use `fishy`, a raw image file is needed. Therefore, an installation script for raw templates was added along with optional `create_guest_raw` and `create_raw` functions to create and start raw guest images, respectively.

5 Validation Model and Evaluation

This section evaluates the data hiding techniques. It introduces the validation model for assessing the difficulty of detecting anti-forensic data hiding techniques. Next, the model is employed to evaluate Btrfs data hiding techniques. Finally, the framework interface functionality is assessed.

5.1 Validation Model

The validation model for evaluating the difficulty of discovering Btrfs hiding techniques has three levels of complexity:

- **Basic Level:** The first level provides an entry barrier for hiding techniques and is, therefore, the weakest validation level. A hiding technique that fails to succeed at the basic level is disqualified from hiding data and would only be usable in another context such as creating falsified evidence [7]. The basic level simulates an uneducated user without forensic knowledge and tools. The only available knowledge is that a hiding incident has occurred and the only tool available at the basic level is a filesystem check. The goal is to confirm the existence of a data hiding incident.
- **Specialist Level:** The second level is equivalent in difficulty to a cryptanalytic chosen plaintext attack [39]. The specialist level assumes that the digital forensic pratitioner knows that hidden data is in the system. However, the practitioner does not know the actual hiding technique used. In addition to knowledge about the incident, additional tools such as The Sleuth Kit are available. Due to the availability of knowledge and tools, the goals at this

level are increased. In addition to confirming the data hiding incident, it is necessary to find and clear the system of hidden data.
- **White Box Level:** The third level is comparable in difficulty to a cryptanalytic white box attack [21, 40]. The level assumes knowledge about the hiding technique as well as access to all possible tools, including Btrfs forensic tools. The only limitations are the lack of knowledge about the size or type of hidden data and no direct access to the specific hiding technique. Due to the increased capabilities at this level, the goals increase in complexity. In addition to confirming the data hiding incident and locating the hidden data, it is required to perform a complete or at least partial reconstruction of the hidden data. However, it is assumed that the hidden data is not encrypted.

5.2 Btrfs Hiding Technique Evaluation

This section evaluates data hiding in the Btrfs file slack and timestamp data structures. It also evaluates the new interface that integrates the `fishy` anti-forensic framework with the `ForTrace` data synthesis framework.

Btrfs File Slack Hiding. Typically, file slack hiding techniques offer the greatest potential data hiding capacity [23]. While this is also true for the Btrfs implementation, there is a caveat regarding reduced storage capacity. Files that are small enough to fit within the extent items are marked as inline and leave no usable slack because they are integrated in metadata structures [37]. Additionally, the stability of this data hiding technique is low because file sizes change. Therefore, it is possible that growing files may overwrite hidden data. Methods for increasing stability include filling slack areas backwards and hiding data in files that are rarely used. However, both methods would reduce the storage capacity.

The basic validation level can only employ filesystem checks. Since the data hiding technique computes and replaces the checksums of used file data blocks, it is not possible to confirm the data hiding incident at the basic level.

The specialist validation level has a much better chance at finding hidden data. While there is no direct way to find the hidden data, sampling file contents using the `icat` tool can lead to discovery. Figure 10 shows that hidden text is easily detected, confirming the data hiding incident and enabling the removal of the hidden data.

In the case of less recognizable data, such as binary image data in Figure 11, achieving the specialist level goal is more complex. Additional steps would have to be taken to confirm hidden data. This could be accomplished in the same way as the data hiding technique. Specifically, `istat` would be used to obtain the file size to be compared with the `icat` file size that includes the slack space. Hidden data is confirmed when the `icat` output is larger than the `istat` output.

Since the data hidden in file slack can be discovered at the specialist level, the same process can be applied at the more powerful white box level. Since the white box level has no restrictions, a data reconstruction is also possible using similar methods as at the specialist level. However, manipulation that is harder

```
Ǝ���p��U�^9tB�d�LI��8>������M�r�R§~�H����}��m�mw�\��G��Cw.������F��Y&��d����H[�o��\����I�6{�tH��w�%-���;�J{
ia�����43<��!S�E�,rY/��`��,�7�$�Ri�i�B���N=�L��u.X^�;��z3�������!�T�pÃ�;Zm[��    m����w�U5b)ia�i�VV1U�9���I��
<c\ъ$��{w#U_>uV��{�}Gxl���#sU�o�)��-�ε�Vu|ƶ�_PK
J       �R`��j�,        MalwareBot-master/MalwareBot/packages.configUTm`�`{�{��}En�BYjQqf~�������Bj^r~Jf^��R
J       �RúY�  '        MalwareBot-master/MalwareBot/upload.txtUTm`�`Dies ist eine hochgeladene DateiPK
J       �R      MalwareBot-master/UTm`�`PK
J       �R��n��         9MalwareBot-master/.gitattributesUTm`�`PK
J       �R��
q       4MalwareBot-master/.gitignoreUTm`�`PK
MalwareBot-master/CommandTest/UTm`�`PK
MalwareBot-master/CommandTest/CommandLineParser.cppUTm`�`PK
J       �R��3�- ZMalwareBot-master/CommandTest/CommandTest.cppUTm`�`PK
J       �R$6C�5h$1       oMalwareBot-master/CommandTest/CommandTest.vcxprojUTm`�`PK
J       �R�cb:�9         �MalwareBot-master/CommandTest/CommandTest.vcxproj.filtersUTm`�`PK
J       �Rů     �y�,     rMalwareBot-master/CommandTest/CryptoTest.cppUTm`�`PK
J       �R7XO�7,         >MalwareBot-master/CommandTest/ParserTest.cppUTm`�`PK
J       �R��"��%         tMalwareBot-master/CommandTest/pch.cppUTm`�`PK
J       �R�&����#        �MalwareBot-master/CommandTest/pch.hUTm`�`PK
J       �Re��Q-          �!MalwareBot-master/MalwareBot.slnUTm`�`PK
J       �R      �$MalwareBot-master/MalwareBot/UTm`�`PK
J       �R���>
�$      ^$MalwareBot-master/MalwareBot/Bot.cppUTm`�`PK
J       �R����           .MalwareBot-master/MalwareBot/Command.cppUTm`�`PK
J       �RI�8�'&         2MalwareBot-master/MalwareBot/Command.hUTm`�`PK
2       N4MalwareBot-master/MalwareBot/CommandLineParser.cppUTm`�`PK
J       �R�q�
             $S0         �8MalwareBot-master/MalwareBot/CommandLineParser.hUTm`�`PK
J       �R�#l`f8['       (:MalwareBot-master/MalwareBot/Crypto.cppUTm`�`PK
J       �R3�#�,%         �IMalwareBot-master/MalwareBot/Crypto.hUTm`�`PK
J       �R����X+         �OMalwareBot-master/MalwareBot/Executions.cppUTm`�`PK
J       �R��t�� G )      �gMalwareBot-master/MalwareBot/Executions.hUTm`�`PK
J       �R%6Z�C/         ~qMalwareBot-master/MalwareBot/InternetHelper.cppUTm`�`PK
J       �R;���H-         �MalwareBot-master/MalwareBot/InternetHelper.hUTm`�`PK
J       �R����U%/        ɑMalwareBot-master/MalwareBot/MalwareBot.vcxprojUTm`�`PK
J       �RK     ��r     7       F�MalwareBot-master/MalwareBot/MalwareBot.vcxproj.filtersUTm`�`PK
J       �R��Cc  '        �MalwareBot-master/MalwareBot/Parser.cppUTm`�`PK
J       �Ru)�`��%        ��MalwareBot-master/MalwareBot/Parser.hUTm`�`PK
J       �R��� ��(        ±MalwareBot-master/MalwareBot/constants.hUTm`�`PK
J       �R`��j�,         ��MalwareBot-master/MalwareBot/packages.configUTm`�`PK
J       �RúY�  '        l�MalwareBot-master/MalwareBot/upload.txtUTm`�`PK  �
                                                                           6685)ٕcc0659050001aa1d4fbc26f3a9
Thou art more lovely and more temperate:
Rough winds do shake the darling buds of May,
And summer's lease hath all too short a date;
Sometime too hot the eye of heaven shines,
And often is his gold complexion dimm'd;
And every fair from fair sometime declines,
By chance or nature's changing course untrimm'd;
But thy eternal summer shall not fade,
Nor lose possession of that fair thou ow'st;
Nor shall death brag thou wander'st in his shade,
When in eternal lines to time thou grow'st:
   So long as men can breathe or eyes can see,
   So long lives this, and this gives life to thee.
```

Fig. 10. `icat` output of a hidden text file in Btrfs file slack.

to detect involves hidden data in slack space that has been deleted. Since file slack areas filled with zeros are not out of the ordinary, it would impossible to determine if hidden data existed in the past at all three validation levels.

Btrfs Timestamp Hiding. The Btrfs timestamp hiding technique is more powerful than the comparable methods for the ext4 and APFS filesystems because all four Btrfs nanosecond bytes can be used to hide data. Of course, the storage capacity of this technique is significantly lower than the file slack hiding technique. Each timestamp has only four bytes available and all four timestamps of the inode can be employed to hide data. Thus, there is a maximum of 16 bytes of available data hiding space per inode. The stability of the timestamp hiding

```
J       �R      �$MalwareBot-master/MalwareBot/UTm`�`PK
J       �R���>
�$      ^$MalwareBot-master/MalwareBot/Bot.cppUTm`�`PK
J       �R����          .MalwareBot-master/MalwareBot/Command.cppUTm`�`PK
J       �RI�8�'&        2MalwareBot-master/MalwareBot/Command.hUTm`�`PK
2       N4MalwareBot-master/MalwareBot/CommandLineParser.cppUTm`�`PK
J       �R�q�
             $S0        �8MalwareBot-master/MalwareBot/CommandLineParser.hUTm`�`PK
J       �R�#l`f8['       (:MalwareBot-master/MalwareBot/Crypto.cppUTm`�`PK
J       �R3�#�,%        �IMalwareBot-master/MalwareBot/Crypto.hUTm`�`PK
J       �R����X+        �OMalwareBot-master/MalwareBot/Executions.cppUTm`�`PK
J       �R��t�� G )     �gMalwareBot-master/MalwareBot/Executions.hUTm`�`PK
J       �R%6Z�C/        ~qMalwareBot-master/MalwareBot/InternetHelper.cppUTm`�`PK
J       �R;���H-        �MalwareBot-master/MalwareBot/InternetHelper.hUTm`�`PK
J       �R����U%/       ⅎMalwareBot-master/MalwareBot/MalwareBot.vcxprojUTm`�`PK
J       �RK     ��r     7       F�MalwareBot-master/MalwareBot/MalwareBot.vcxproj.filtersUTm`�`PK
J       �R��Cc  '       �MalwareBot-master/MalwareBot/Parser.cppUTm`�`PK
J       �Ru)�`��%       ��MalwareBot-master/MalwareBot/Parser.hUTm`�`PK
J       �R��� ��(       ⍊MalwareBot-master/MalwareBot/constants.hUTm`�`PK
J       �R`��j�,        ��MalwareBot-master/MalwareBot/packages.configUTm`�`PK
J       �RúY�  '        l�MalwareBot-master/MalwareBot/upload.txtUTm`�`PK  �
                                                                        6685)ر cc0659050001aa1d4fbc26f3a9001fca11a2����JFIF���

,$&1'-=-157:::#+?D?8C49:7
```


Fig. 11. `icat` output of a hidden image file in Btrfs file slack.

```
Inode (virtual): 1
Subvolume: 0x5
Inode (real): 257
Allocated: yes
Compressed: no
Generation: 8
UID / GID: 0 / 0
Mode: drwxr-xr-x
Size: 0
Num of Links: 1

Flags:

Inode Times:
Accessed:       2023-01-14 19:23:21.1701144692 (UTC)
Created:        2023-01-14 19:23:25.544175136 (UTC)
Modified:       2023-01-14 19:23:25.1970479201 (UTC)

Extended attributes:
```

Fig. 12. `istat` output of inode timestamps affected by `write`.

technique can be improved by reducing the number of timestamps used because some timestamps are changed more frequently than others. But this would significantly decrease the storage capacity of the timestamp technique.

The base validation level would fail to find hidden data in the timestamps because the checksums would have been modified after the data was hidden. Therefore, the data hiding incident could not be confirmed at the base level.

```
Inode (virtual): 1
Subvolume: 0x5
Inode (real): 257
Allocated: yes
Compressed: no
Generation: 8
UID / GID: 0 / 0
Mode: drwxr-xr-x
Size: 0
Num of Links: 1

Flags:

Inode Times:
Accessed:       2023-01-14 19:23:21.000000000 (UTC)
Created:        2023-01-14 19:23:25.000000000 (UTC)
Modified:       2023-01-14 19:23:25.000000000 (UTC)

Extended attributes:
```

Fig. 13. `istat` output of inode timestamps affected by `clear`.

At the specialist level, single inodes with hidden data could be discovered. While the hidden data shown in Figure 12 is difficult to distinguish, the presence of deleted data shown in Figure 13 would be discovered because it would be very unusual for multiple timestamps to have their nanosecond parts filled with zeros.

```
Inode (virtual): 1
Subvolume: 0x5
Inode (real): 257
Allocated: yes
Compressed: no
Generation: 8
UID / GID: 0 / 0
Mode: drwxr-xr-x
Size: 0
Num of Links: 1

Flags:

Inode Times:
Accessed:       2023-01-14 19:23:21.908000000 (UTC)
Created:        2023-01-14 19:23:25.748000000 (UTC)
Modified:       2023-01-14 19:23:25.748000000 (UTC)

Extended attributes:
```

Fig. 14. `istat` output of unaffected inode timestamps.

However, an anomaly would be identified at the specialist level if the unaffected timestamps in Figure 14 were to be compared against the affected timestamps in Figure 12. Therefore, the data hiding incident could be confirmed at the specialist level, but it would not be possible to find all the hidden data. Nevertheless, at the specialist level it would be possible to find all the former hiding places in the case of deleted data.

At the white box level, the first goal of confirming the data hiding incident would be accomplished in the same way as at the specialist level. Similarly, all the locations with deleted hidden data would be identified. However, reconstructing the data would be complicated. If the hidden data was simple text, manually investigating all the timestamps using a hex or disk editor could restore the hidden data. However, if hidden binary data or encrypted data was deleted, this process would not work because the recovered hidden data would be indistinguishable from legitimate timestamps.

5.3 Framework Interface Functionality

The correct functionality of the new interface between the `fishy` and `ForTrace` frameworks is validated by showing that the three goals stated above are accomplished:

- `ForTrace` is able to use the hiding techniques provided by `fishy`. This fundamental functionality is realized by the newly-implemented DataHiding utility class.
- Service on the manipulated virtual machine must not be interrupted. All the hiding techniques presented in Table 2 can be used to hide data on running machines and shutdown machines without interrupting running machines or preventing shutdown machines from being started. This is accomplished by incorporating the filesystem partition offset as a parameter of the interface function that invokes the hiding technique wrappers.
- Compatible hiding techniques must work despite the presence of additional partitions. The changes that address the previous goal also help accomplish this goal. A new parameter `-area` has been added to account for different starting offsets and multiple filesystems available in the partitioning. The separate use of `fishy` in the traditional manner is not affected because the new parameter is initialized to zero by default.

6 Conclusions

This work has attempted to improve the digital forensic process by assessing anti-forensic measures at the filesystem level and providing a means for synthesizing datasets containing anti-forensic artifacts. Additionally, three validation models for assessing anti-forensic data hiding techniques have been developed.

The research has demonstrated that the Btrfs filesystem is resilient to some data hiding methods due to its robust checksums and filesystem protections such as its logical addressing structure and reducing the number of files capable of producing slack space. However, Btrfs is still susceptible to data hiding methods. The new interface that integrates the `fishy` anti-forensic framework and the `ForTrace` data synthesis framework makes it possible to evaluate data hiding techniques for Btrfs as well as for the popular ext4 and NTFS filesystems. Due to its modular design, the interface component can be extended to include additional anti-forensic data hiding techniques and other filesystems. Furthermore,

`ForTrace` can be used to generate complex scenarios involving hidden data at the filesystem level over and above its existing data synthesis functions.

Future research will attempt to enhance compatibility by synthesizing datasets using additional `fishy`-compatible filesystems such as APFS, FAT32 and exFAT. A viable APFS interface requires a `ForTrace` macOS implementation that is missing, but introducing FAT32 and exFAT compatibility is relatively simple. This would permit the simulation of anti-forensic manipulations of USB devices during data synthesis.

Future research will also focus on the Btrfs filesystem. Certain other slack and reserved areas are known to exist in Btrfs, some of which have been considered unusable due to checksums and other filesystem protections [36, 37]. However, experiments conducted during this research revealed that some of the reserved areas in the filesystem could be used to hide data. Key features that have not been implemented for Btrfs with regard to this research are using multiple hard drives and addressing multiple subvolumes. Structures and parsing functions needed to address these issues are present in the current Btrfs module. Future research will attempt to implement this functionality.

References

1. G. Abduhalil, M. Obid and Y. Rakhmatulla, Algorithm for steganographic hiding of information using a filesystem, *Proceedings of the International Conference on Information Science and Communications Technologies*, 2021.
2. S. Abt and H. Baier, Are we missing labels? A study of the availability of ground truth in network security research, *Proceedings of the Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, pp. 40–55, 2014.
3. H. Berghel, D. Hoelzer and M. Sthultz, Data hiding tactics for Windows and Unix filesystems, *Advances in Computers*, vol. 74, pp. 1–17, 2008.
4. W. Bhat and M. Wani, Forensic analysis of the B-tree filesystem (Btrfs), *Digital Investigation*, vol. 27, pp. 57–70, 2018.
5. Btrfs Contributors, *Wikipedia: The Free Encyclopedia* (`en.wikipedia.org/wiki/Btrfs`), 2024.
6. A. Ceballos Delgado, W. Glisson, G. Grispos and K. Choo, Fade: A forensic image generator for Android device education, *Wiley Interdisciplinary Reviews: Forensic Science*, vol. 4(2), article no. e1432, 2022.
7. K. Conlan, I. Baggili and F. Breitinger, Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy, *Digital Investigation*, vol. 18(S), pp. S66–S75, 2016.
8. X. Du, C. Hargreaves, J. Sheppard and M. Scanlon, TraceGen: User activity emulation for digital forensic test image generation, *Digital Investigation*, vol. 38(S), article no. 301133, 2021.
9. K. Eckstein and M. Jahnke, Data hiding in journaling filesystems, presented at the *Digital Forensic Research Workshop*, 2005.
10. D. Forte, Dealing with forensic software vulnerabilities: Is anti-forensics a real danger? *Network Security*, vol. 2008(12), pp. 18–20, 2008.

11. A. Gehrke, Forensic Analysis of Timestamps in Selected Filesystems (in German), Bachelor of Engineering Thesis, IT Forensics, Hochschule Wismar, University of Applied Sciences: Technology, Business and Design, Wismar, Germany (`it-forensik.fiw.hs-wismar.de/images/9/95/BT_AGehrke.pdf`), 2020.
12. T. Göbel and H. Baier, Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding, *Digital Investigation*, vol. 24(S), pp. S111–S120, 2018.
13. T. Göbel and H. Baier, `fishy` – A framework for implementing filesystem-based data hiding techniques, in *Digital Forensics and Cyber Crime*, F. Breitinger and I. Baggili (Eds.), Springer, Cham, Switzerland, pp. 23–42, 2019.
14. T. Göbel, S. Maltan, J. Türr, H. Baier and F. Mann, `ForTrace` – A holistic forensic dataset synthesis framework, *Forensic Science International: Digital Investigation*, vol. 40(S), article no. 301344, 2022.
15. T. Göbel, T. Schäfer, J. Hachenberger, J. Türr and H. Baier, A novel approach for generating synthetic datasets for digital forensics, in *Advances in Digital Forensics XVI*, G. Peterson and S. Shenoi (Eds.), Springer, Cham, Switzerland, pp. 73–93, 2020.
16. C. Grajeda, F. Breitinger and I. Baggili, Availability of datasets for digital forensics – And what is missing, *Digital Investigation*, vol. 22(S), pp. S94–S105, 2017.
17. R. Harris, Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem, *Digital Investigation*, vol 3(S), S44–S49, 2006.
18. J. Heeger, Y. Yannikos and M. Steinebach, Exhide: Hiding data within the exFAT filesystem, *Proceedings of the Sixteenth International Conference on Availability, Reliability and Security*, article no. 77, 2021.
19. J. Hilgert, M. Lambertz and S. Yang, Forensic analysis of multiple device Btrfs configurations using The Sleuth Kit, *Digital Investigation*, vol. 26(S), pp. S21–S29, 2018.
20. E. Huebner, D. Bem and C. Wee, Data hiding in the NTFS filesystem, *Digital Investigation*, vol. 3(4), pp. 211–226, 2006.
21. M. Joye, On white-box cryptography, in *Security of Information and Networks*, A. Elci, S. Ors and B. Preneel (Eds.), Trafford Publishing, Victoria, Canada, pp. 7–12, 2008.
22. A. Juch, Btrfs Filesystem Forensics, Master's Thesis, Software Engineering/Internet Program, Faculty of Informatics, Vienna University of Technology, Vienna, Austria, 2014.
23. A. Kailus, C. Hecht and T. Göbel, `fishy` – A framework for implementing hiding techniques in filesystems, *Proceedings of the D-A-CH Security Conference*, 2018.
24. X. Lin, *Introductory Computer Forensics: A Hands-On Practical Approach*, Springer, Cham, Switzerland, 2018.
25. A. McDonald and M. Kuhn, StegFS: A steganographic filesystem for Linux, *Proceedings of the Third International Workshop on Information Hiding*, pp. 463–477, 1999.
26. C. Moch and F. Freiling, The Forensic Image Generator Generator (Forensig^2), *Proceedings of the Fifth International Conference on IT Security Incident Management and IT Forensics*, pp. 78–93, 2009.
27. C. Moch and F. Freiling, Evaluating the Forensic Image Generator Generator, *Proceedings of the International Conference on Digital Forensics and Cyber Crime*, pp. 238–252, 2011.
28. S. Neuner, A. Voyiatzis, M. Schmiedecker, S. Brunthaler, S. Katzenbeisser and E. Weippl, Time is on my side: Steganography in filesystem metadata, *Digital Investigation*, vol. 18(S), pp. S76–S86, 2016.

29. J. Park, TREDE and VMPOP: Cultivating multi-purpose datasets for digital forensics – A windows registry corpus as an example, *Digital Investigation*, vol. 26, pp. 3–18, 2018.
30. A. Qadir and A. Varol, The role of machine learning in digital forensics, *Proceedings of the Eighth International Symposium on Digital Forensics and Security*, 2020.
31. O. Rodeh, J. Bacik and C. Mason, BTRFS: The Linux B-tree filesystem, *ACM Transactions on Storage*, vol. 9(3), article no. 9, 2013.
32. M. Rogers and M. Lockheed, Anti-Forensics, Center for Education and Research in Information Assurance and Security, Department of Information and Computer Technology, Purdue University, West Lafayette, Indiana, 2005.
33. M. Scanlon, X. Du and D. Lillis, EviPlant: An efficient digital forensics challenge creation, manipulation and distribution solution, *Digital Investigation*, vol. 20(S), pp. S29–S36, 2017.
34. J. Schneider, M. Eichhorn and F. Freiling, Ambiguous filesystem partitions, *Forensic Science International: Digital Investigation*, vol. 42(S), article no. 301399, 2022.
35. H. Visti, S. Tohill and P. Douglas, Automatic creation of computer forensic test images, in *Computational Forensics*, U. Garain and F. Shafait (Eds.), Springer, Cham, Switzerland, pp. 163–175, 2015.
36. M. Wani, A. AlZahrani and W. Bhat, Filesystem anti-forensics – Types, techniques and tools, *Computer Fraud and Security*, vol. 2020(3), pp. 14–19, 2020.
37. M. Wani, W. Bhat and A. Dehghantanha, An analysis of anti-forensic capabilities of the B-tree filesystem (Btrfs), *Australian Journal of Forensic Sciences*, vol. 52(4), pp. 371–386, 2020.
38. K. Woods, C. Lee, S. Garfinkel, D. Dittrich, A. Russell and K. Kearton, Creating realistic corpora for security and forensic education, *Proceedings of the Sixth Annual Conference on Digital Forensics, Security and Law*, 2011.
39. U. Wurst, Use of Cryptographic Methods on Highly Resource-Limited Devices (in German), Ph.D. Thesis, Faculty of Computer Science, University of Stuttgart, Stuttgart, Germany (`d-nb.info/1042941734/34`), 2003.
40. B. Wyseur, W. Michiels, P. Gorissen and B. Preneel, Cryptanalysis of white-box DES implementations with arbitrary external encodings, *Proceedings of the Fourteenth International Workshop on Selected Areas in Cryptography*, pp. 264–277, 2007.