Data hiding in file systems: Current state, novel methods, and a standardized corpus

By:
Anton Schwietert, Jan-Niclas Hilgert

From the proceedings of
The Digital Forensic Research Conference
DFRWS APAC 2025
Nov 10-12, 2025

Forensic Science International: Digital Investigation 54 (2025) 301984

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

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

DFRWS APAC 2025 - Selected Papers from the 5th Annual Digital Forensics Research Conference APAC

Data hiding in file systems: Current state, novel methods, and a standardized corpus

Anton Schwietert, Jan-Niclas Hilgert *

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

ARTICLE INFO

Keywords:
Data hiding
Anti forensics
File system forensics
Data corpus

ABSTRACT

File systems are a fundamental component of virtually all modern computing devices. While their primary purpose is to manage and organize data on persistent storage, they also offer a range of opportunities for concealing information in unintended ways—a practice commonly referred to as data hiding. Given the challenges these techniques pose to forensic analysis, it becomes essential to understand where and how hidden data may reside within file system structures. In response, this paper systematically examines the current state of research on data hiding techniques in file systems, consolidating known methods across widely used file systems including NTFS, ext, and FAT. Building on this comprehensive survey, we explore how existing methods can be adapted or extended and identify previously unexamined data hiding opportunities, particularly in underexplored file systems. Furthermore, we propose and discuss novel data hiding techniques leveraging unique properties of contemporary file systems such as the misuse of snapshots. To support future research and evaluation, we apply a range of data hiding techniques across multiple file systems and present the first publicly available, scenario-based dataset dedicated to file system data hiding. As no comparable dataset currently exists, this contribution addresses a critical gap by supporting systematic evaluation and encouraging the development of effective detection methods.

1. Introduction

File systems are essential for organizing storage space and are therefore a fundamental component of every modern computing device. While recovering deleted files is a well-established aspect of file system analysis, prior research has also revealed the potential for concealing data within the file system itself, e.g. in slack space. Depending on the technique used, data hiding in file systems can offer a robust and high-capacity means of storing information covertly. Given the ubiquity of file systems, understanding both the methods for storing hidden data and the techniques for uncovering it is crucial for a file system forensic analysis. However, due to the diversity of file systems and the wide range of possible hiding techniques, it is often difficult to know which methods exist and which file systems they target.

In this work, we address this gap by first offering a comprehensive overview of data hiding techniques in file systems covering known methods. While a recent study conducted a similar survey, their work only included a subset of existing methods, as acknowledged by the authors themselves Toolan and Humphries (2025a). Building on our comprehensive survey, we further explore how established data hiding techniques can be adapted to lesser-studied, contemporary file systems and introduce novel strategies that leverage their unique features. To support the development and evaluation of detection techniques, we also present the first detailed and comprehensive dataset specifically dedicated to data hiding in file systems.

In summary, this paper makes the following contributions:

- Provide an overview of the current state of data hiding techniques in file systems.
- Discuss the potential for extending existing data hiding methods to different file systems or structures.
- Highlight novel possibilities for data hiding in contemporary file systems.
- Implement and share a corpus for evaluating data hiding detection methods.

The remainder of the paper is structured as follows: Section 2 reviews existing data hiding techniques and prior research. Section 3 extends

* Corresponding author.
E-mail addresses: anton.schwietert@fkie.fraunhofer.de (A. Schwietert), jan-niclas.hilgert@fkie.fraunhofer.de (J.-N. Hilgert).

https://doi.org/10.1016/j.fsidi.2025.301984

Available online 3 November 2025

these concepts to underexplored file systems and new hiding locations, while Section 4 presents novel techniques for contemporary file systems. Based on these results, Section 5 describes the standardized corpus containing representative examples of various data hiding strategies across multiple file systems. Section 6 concludes the paper and outlines directions for future research.

2. Current state of data hiding

Data hiding in file systems is a well-established area of research in digital forensics. To provide an overview of existing techniques, we reviewed 24 key scientific publications published between 2005 and 2025 that describe various methods for concealing data within file systems. In the sections that follow, we discuss these techniques using the commonly applied categorization based on the location of hidden data: *slack space*, *reserved space*, and *misuse of file system structures* Göbel and Baier (2018a). For a more detailed overview and to pinpoint exactly which components are exploited, we adopt the categorization proposed by Brian Carrier (2005), which divides file system structures into various data categories: *file system*, *content*, *metadata*, and *filename*. This fine-grained structure is reflected in Table 1, where each cell references the publication that describes a specific data hiding method in a given file system and location. The following sections provide a detailed overview of these existing data hiding techniques.

2.1. Slack space

One of the most common methods of hiding information in all file systems is the use of slack space. Slack space is the unused storage space in the last block[1] of a file that is created by the difference between the actual file size and the fixed block size. As these areas are not visible to standard users and leave the file size unchanged, they are particularly well-suited for data hiding. In addition, Srinivasan and Pieper also present a new steganographic method that leverages the file slack space to hide an entire filesystem volume Srinivasan and Pieper (2022).

Slack space can occur not only at the end of files, but also at the end of other file system structures. Examples include the superblock Piper et al. (2005); Göbel and Baier (2018a); Toolan and Humphries (2025b), block and inode bitmaps Göbel and Baier (2018a), inodes Göbel and Baier (2018a); Toolan and Humphries (2025b), the group descriptor table Piper et al. (2005); Göbel and Baier (2018a) and the boot sector Berghel et al. (2008); Piper et al. (2005); Göbel and Baier (2018a).

In recent years, research has increasingly turned to underexplored file systems such as APFS and Btrfs. In the case of APFS, slack space located at the end of key structures—including the container superblock, volume superblocks, and the object map—has been analyzed for its potential use as covert storage for hidden data Koolhaas and van Steenbergen (2020).

For Btrfs, prior work has shown that file slack can be leveraged to conceal data Göbel et al. (2024). However, Btrfs enforces data integrity through checksums on its data blocks, which must be updated when modifications are made, posing an additional challenge for data hiding. Beyond file slack, Btrfs node structures, including internal and leaf nodes, also provide hiding opportunities through their inherent slack space. This is particularly true for underutilized internal nodes. Overall, the available hiding capacity increases with the size and complexity of the file system Göbel et al. (2024).

A novel method recently introduced leverages symbolic links in to create slack space suitable for data hiding Toolan and Humphries (2025a). In most file systems, the target path of a symbolic link is stored directly within the metadata structure. However, if the path exceeds this available space, e.g. beyond 60 bytes in ext2, a separate data block is allocated to store the link target. This block is typically underutilized, leaving slack space at the end that can be exploited to conceal data. While the technique proved effective in most of the file systems analyzed, it did not succeed in APFS and Btrfs. Nonetheless, their

Table 1
Categorization of data hiding techniques across different file systems based on prior research.

	ext2/ext3	ext4	APFS	FAT	NTFS	XFS	exFAT	Btrfs
File System Category								
Superblock	2	9	22			7		20
Group Descriptor (Table)	2,16	9						
Boot Sector	2	9		1				
Content Category								
File Slack	6,17,19	8,24		5,8,14,18,19	3,8,17,18			20
Deleted Files				5	5			
Alternate Data Streams					1,3,5,16,17,18, 19			
Bad Blocks	1	8		1,5,8,14	1,3,8,17			
Block Bitmap		9						
Additional Data Units					17			
Metadata Category								
Metadata Entries	2,6	8,9	10, 22		3	7	4, 23	
MFT/FAT		8		8,13	1, 3			
$DATA Attribute					3			
Timestamps		8,9,11	10		8,12	7	4,23	20
Extended Attributes	17					17		
Allocation Bitmap		9						
Symbolic Link Slack	21	21			21	21		
File Name Category								
File Names	6,19				19			
Directory Entries		9		15				

1 Berghel et al. (2008), 2 Piper et al. (2005), 3 Huebner et al. (2006), 4 Heeger et al. (2022), 5 Davis et al. (2010), 6 Eckstein and Jahnke (2005), 7 Toolan and Humphries (2025b), 8 Göbel and Baier (2018b), 9 Göbel and Baier (2018a), 10 Göbel et al. (2019), 11 Göbel and Baier (2018a,b), 12 Neuner et al. (2016), 13) Khan et al. (2011), 14 Liu et al. (2009), 15 Kim et al. (2022), 16 Piper et al. (2006), 17 Krenhuber and Niederschick (2007), 18 Hassan and Hijazi (2016), 19 Dillon (2006), 20 Göbel et al. (2024), 21 Toolan and Humphries (2025a), 22 Koolhaas and van Steenbergen (2020), 23 Heeger et al. (2021), 24 Srinivasan and Pieper (2022)

[1] Block and cluster refer to the smallest I/O unit of a file system; hereafter, we use 'block' exclusively.

inclusion in the evaluation highlights the growing relevance of modern file systems in data hiding research.

The combination of potentially large amounts of slack space across multiple blocks and file system structures, together with the simplicity of the approach, makes slack space a promising target for data hiding. However, since slack space may be reused when files or structures are extended, any hidden data stored in these areas is at risk of being quickly overwritten.

2.2. Reserved space

The various reserved areas within a file system provide another way to hide data. These areas are usually reserved for future use or byte-alignment and not considered useable by the file system, thus minimizing the risk of data being overwritten in these areas. As a result, they serve as effective locations for concealing data.

Examples of using these areas to hide data are provided by Piper et al. (2005), who stored data in reserved areas of the superblock and its backup copies. Other reserved areas can be found in the boot sector Göbel and Baier (2018a), the group descriptor table Piper et al. (2005), and inodes Göbel and Baier (2018b). In inodes, data can be stored in reserved areas within the inodes as well as in entire reserved inodes Göbel and Baier (2018a); Toolan and Humphries (2025b). In APFS, up to 10 bytes of padding per inode can potentially be used to hide information Koolhaas and van Steenbergen (2020). In the ext2 file system, inodes 7 to 10 are reserved for future use, providing 512 bytes that can be repurposed for data hiding. In ext3, however, inodes 7 and 8 are used, reducing the available space Piper et al. (2005). In ext4, inodes 9 and 10 can be utilized to hide information without affecting the functionality of the file system Göbel and Baier (2018a). Additionally, the last two bytes of the 12-byte `osd2` field—located at offset `0x74` within each inode—can serve as a hiding space Göbel and Baier (2018b).

In the Btrfs file system, each superblock is preceded by a 64 KiB reserved region that remains unmodified during normal operation. This area can serve as a potential location for covert data storage Göbel et al. (2024).

In XFS, each allocation group includes a free list area containing four reserved blocks designated for future expansion of the inode B + Tree. These blocks, which collectively offer 64 KiB of space, can be repurposed for data hiding Toolan and Humphries (2025b).

2.3. Misuse of file system structures

Beyond slack and reserved space as methods for data hiding, existing file system structures themselves can also be misused to conceal data.

2.3.1. Timestamps

One notable example involves embedding data in the nanosecond portion of file system timestamps Göbel et al. (2019); Göbel and Baier (2018); Neuner et al. (2016). This technique has already been successfully demonstrated in several file systems. Detecting such hidden data is difficult because common file browsers do not usually display the nanosecond fields. Even when using terminal commands like `stat`, which reveal full timestamps, subtle variations in the nanosecond range are not easily noticeable. However, the method suffers from limited robustness. Timestamps are frequently overwritten during regular file system operations. To address this issue Neuner et al. (2016) incorporated error correction codes to enhance the reliability of this technique. The capacity of this approach depends on the level of nanosecond precision supported by the file system. For instance Göbel and Baier (2018) evaluated this technique on ext4 and successfully embedded up to 6.5 bytes per timestamp.

Another method demonstrates that Btrfs allows data hiding by exploiting the 4-byte sub-second portion of inode timestamps Göbel et al. (2024). This technique avoids modifying the main 8-byte timestamp value, thereby reducing the risk of detection through standard file system integrity checks. The process involves locating the relevant inode offsets, embedding the input data within the sub-second timestamp fields, and recalculating the inode's checksum to maintain file system integrity.

For exFAT, more advanced data hiding techniques that exploit timestamp metadata have been proposed Heeger et al. (2021, 2022). One such method conceals information in the file directory entry fields `Create10msIncrement` and `LastModified10msIncrement`, each 1 byte, by modifying only the 6 least significant bits. This technique supports data hiding in both existing and deleted files. A related method, known as exHide, targets only deleted files and embeds data into various metadata fields, including timestamp components, `Create10msIncrement`, `FirstCluster`, and file size. To maintain plausible values and reduce the likelihood of detection, it alters only the least significant bytes of these fields and distributes the encrypted, error-corrected embedded data across selected metadata using a pseudorandom scheme derived from a shared password.

2.3.2. Block allocation manipulation

Another data hiding method exploits bad block management in file systems. NTFS stores the addresses of unusable blocks in the `$BadClus` attribute of the eighth MFT record Berghel et al. (2008). FAT marks such blocks in the File Allocation Table, and ext file systems reserve the first inode for this purpose. By adding non-defective blocks to these lists, users can create storage areas that are ignored by the file system and could be used to conceal data Göbel and Baier (2018b); Davis et al. (2010); Huebner et al. (2006); Krenhuber and Niederschick (2007).

In contrast to methods that rely on bad block manipulation, Krenhuber and Niederschick (2007) describe a technique that alters the MFT or FAT to allocate additional, unrelated blocks to a file. This artificially increases the file's allocated space, creating slack space suitable for hiding data. A more file system–specific variant of this technique targets the `$DATA` attribute in NTFS Huebner et al. (2006). It allows hidden data to be appended to existing metadata files without affecting their intended functionality. Static files such as `$Boot`, `$Bitmap`, or `$UpCase` are especially suited for this, as their `$DATA` attributes are rarely updated. By manipulating block allocation, additional space can be reserved and covertly filled with hidden content.

2.3.3. Deleted files

In file systems such as FAT, deleting a file typically involves removing its references from the file allocation table, while the actual data remains on disk. To optimize performance, many file systems avoid immediately overwriting deleted content. Davis et al. (2010) highlight that this behavior can be exploited to hide data in deleted files. However, unless specific precautions are taken, such hidden content is at high risk of being overwritten by subsequent file system activity.

2.3.4. Hierarchy manipulation

Certain data hiding methods target the directory structures of file systems to obscure files without altering their content. In FAT file systems, for instance, directory entries store short file names and metadata. Kim et al. (2022) introduced a technique called *NULL byte injection*, where null bytes are inserted into the directory entry. As a result, affected files become effectively hidden. However, depending on the specific manipulation, the technique may lack robustness, as the file system may treat the altered directory entries as free space and overwrite them.

Similar manipulation is possible in ext file systems. One technique involves setting the `inode` and `name_len` fields of a directory entry to zero, causing the entry to appear unused. The `rec_len` field can then be extended to cover the entire block, allowing arbitrary data to be hidden in the name field Göbel and Baier (2018a).

A simple yet effective way to hide files is to use unconventional file names. By inserting special characters, such as carriage returns, into file names, certain tools may not parse or display them correctly, potentially skipping these files during analysis Eckstein and Jahnke (2005).

2.3.5. Alternate data streams

The NTFS file system supports multiple data streams for a single file, with the primary stream holding the visible content. Additional streams, known as Alternate Data Streams (ADS), can be added without affecting the file's apparent size. These streams are often not shown by standard file management tools and may therefore be easily overlooked. Multiple ADS can be associated with one file, and their storage capacity is generally constrained only by the available disk space on the NTFS volume Hassan and Hijazi (2016); Davis et al. (2010); Berghel et al. (2008); Huebner et al. (2006); Krenhuber and Niederschick (2007); Dillon (2006).

2.3.6. Extended Attributes

Linux supports a feature known as Extended Attributes, available across several file systems. These attributes are stored as name/value pairs associated with files or directories and allow metadata to be attached beyond the standard set of file system attributes. Although support must be enabled in the kernel configuration, it is widely available and easy to use once activated. Since these attributes are stored separately from file content and are not visible in standard file managers, they can be misused to conceal data Krenhuber and Niederschick (2007).

2.4. Tools for data hiding

In this section, we present tools that have been developed to hide data in file systems. During our research, we found that there are very few tools that are publicly available, trustworthy, and function as promised. The only one that stands out is the fishy framework, which we will discuss in greater detail at the end of this section.

slacker.exe[2] is a tool designed to hide data in NTFS slack space. Originally developed by James C. Foster and Vincent Liu, it was part of the Metasploit Anti-Forensics Project. However, it is no longer hosted on the project's official website.

bmap is a Linux-based tool that also targets slack space, specifically within NTFS file systems. It appears to be unmaintained and currently no reliable repositories or documentation are available.

FragFS, developed by Irby Thompson and Mathew Monroe, introduces a technique to hide data in the last 8 bytes of NTFS Master File Table (MFT) entries. Like the others, it seems to have been abandoned and is not available through verified distribution channels.

While the previously discussed tools focus primarily on the NTFS file system, a number of data hiding tools have also been developed for the ext family—though most target the now outdated *ext2* file system. As a result, many of these tools are no longer actively maintained or may not be compatible with modern file systems.

One example is *KY FS*, which manipulates directory entries to make them appear unused to the file system, allowing hidden data to be stored within them. Another is the *Waffen FS* toolkit, which extends an *ext2* file system by adding an *ext3*-style journal, creating up to 32 MB of useable space for data hiding. The *Data Mule FS* tool takes a different approach by utilizing reserved areas in key file system structures, such as superblocks, inodes, and group descriptor tables, to conceal data. On a 1 GB *ext2* file system image, this technique can offer up to 1 MB of covert storage.

More recent developments include *slack_hider*[3] as well as *timestamps-magic*, both designed for use with the *ext4* file system. *slack_hider* is a Python-based tool that stores and retrieves data from file slack space. It identifies all available slack regions and creates a Slack Allocation Table (SAT) to manage the location of hidden content. The SAT itself is stored in the first slack sector and is used for subsequent file retrieval. *timestamps-magic* takes a different approach by embedding data into the nanosecond fields of inode timestamps. This tool has since been integrated into the *fishy* framework.

The *fishy framework*[4] is a Python-based toolkit for implementing and analyzing data hiding techniques at the file system level. Developed as part of an academic research project, it is publicly available and well documented. Fishy supports multiple file systems, including FAT, NTFS, ext4, and more recently, APFS Göbel et al. (2019). The framework aggregates a variety of established data hiding methods that exploit file system structures, such as slack space, manipulation of bad block records, and alternate data streams. By offering a unified platform for experimentation and evaluation, fishy serves as a valuable resource for research and education in digital forensics and information hiding.

2.5. Summary

Our literature review demonstrates that a wide range of data hiding techniques for file systems have been developed and evaluated over the years. These techniques vary significantly in terms of maximum data capacity, robustness, and ease of use. While file slack space remains a well-known and general method that forensic investigators should be aware of, recent approaches increasingly exploit specific file system structures while also addressing the challenges of more advanced data hiding methods such as integrity checks. This diversity complicates the development of generic detection methods.

Moreover, prior research has primarily focused on established file systems such as ext, FAT, and NTFS. In contrast, data hiding techniques for more contemporary file systems—such as ZFS, APFS, and Btrfs—have received comparatively little attention. The following sections address this gap by examining underexplored data hiding techniques in modern file systems that should be considered in forensic analysis and tool development. Although not exhaustive, these insights contribute to the broader understanding of emerging anti-forensic strategies.

3. Extending existing data hiding techniques

In this section we identify additional hiding locations across both new and well-studied file systems based on existing data hiding methods.

3.1. Slack space

Although file slack is a well-established concept in data hiding, it gains renewed relevance in the context of contemporary file systems that employ dynamically sized extents, as noted by Beebe et al. (2009). Furthermore, the use of checksums in modern file systems can complicate this otherwise straightforward hiding technique and increase the likelihood of detection.

3.1.1. Content data (file slack)

Table 2 summarizes the minimum, maximum, and default block sizes for various file systems, along with their use of integrity checks to provide a better overview. Furthermore, the following section discusses the

[2] https://resources.bishopfox.com/resources/tools/other-free-tools/mafia/.

[3] https://github.com/exembly/slack_hider.

[4] https://github.com/dasec/fishy.

Table 2
Overview of block size configurations and checksum support across common file systems.

FS	Block Size			Checksum	
	default	min	max	meta	data
ext2/3	1/4 KiB	1 KiB	4 KiB[b]	×	×
ext4	4 KiB	1 KiB	64 KiB	×	×
XFS	4 KiB	512 B	64 KiB[c]	✓	×
NTFS	4 KiB	512 B	2048 KiB[d]	×	×
FAT	[a]	512 B	512 KiB[e]	×	×
exFAT	[a]	512 B	32 MB	✓	×
Btrfs	4 KiB	512 B	64 KiB	✓	✓
APFS	4 KiB	4 KiB	64 KiB	✓	×
ZFS	128 KB	512 B	16 MiB	✓	✓

[a] Default block size increases with partition size.
[b] 8 KiB on Alpha systems.
[c] Limited by the system's memory page size.
[d] Since version 1709. Before 64 KiB.
[e] Using the official maximum sector size of 4096 B.

challenges and intricacies of data hiding in file slack specifically for ZFS.

3.1.1.1. ZFS. Unlike many traditional file systems, where the block size is fixed at creation, ZFS uses a flexible scheme based on the `recordsize` property. This parameter defines the maximum block size used for storing file data and can be set during dataset creation (i.e., the creation of a ZFS file system). Although it can be modified later, the new value only affects files created thereafter.

Our experiments confirmed, that the `recordsize` serves as an upper limit. For files smaller than or equal to this value, ZFS allocates a single logical block whose size is rounded up to the next multiple of 512 bytes. For files larger than `recordsize`, ZFS splits the content into multiple blocks of exactly `recordsize`, with the final block padded to `recordsize`.

This padding behavior leads to slack space that can be exploited for data hiding. For instance, with a `recordsize` of 2 KiB (2048 bytes), a 1400-byte file would be stored in a single block rounded up to 1536 bytes, leaving 112 bytes of slack. In contrast, a 2100-byte file would be split into two full 2048-byte blocks (4096 bytes total), even though only 2100 bytes are needed—resulting in 1996 bytes of slack space in the second block. Given that `recordsize` can be configured up to 16 MiB, this mechanism enables the creation of substantial slack space, making ZFS an attractive target for data hiding.

ZFS also supports transparent data compression at the block level. Although this feature is available, it is disabled by default for newly created datasets. For the slack space–based data hiding method to function as intended, compression should remain disabled, as it could otherwise eliminate or alter slack space during storage.

In addition, ZFS ensures data integrity through mandatory checksums on all data blocks. If slack space is modified, these checksums must be recalculated to avoid detection. While it is technically possible to disable checksumming, this is strongly discouraged in ZFS documentation due to the risk of data corruption. However, from the perspective of an attacker, disabling checksums makes it significantly easier to hide data undetected. As such, the absence of checksums should be considered a strong forensic red flag.

As highlighted by Hilgert et al. (2017), the mapping of a logical file system address to the physical device and offset in ZFS depends on the specific pool configuration and involves additional steps. Consequently, this mapping is also essential for accurately locating slack space used for data hiding.

3.1.2. File system data

As discussed earlier, slack space can also exist at the end of various file system structures, including the superblock. This structure, which stores essential metadata such as file system size and block allocation, has been shown to allow data hiding in ext file systems. Other file systems rely on similar metadata structures, which may likewise contain slack space that can be exploited, as summarized in Table 3.

Table 3
Slack capacity details for FAT, exFAT, and ZFS.

FS	Location	Capacity	Quantity
FAT	FAT Table	sz – (FAT length mod sz)	2
exFAT	Boot Sector	sz – 512 B	2
	OEM Sector	sz – 480 B	2
	Alloc. Bitmap	sz – ((Block count/8) mod sz)	2
ZFS	Indirect Block	bs – (pointer count × 128 B)	per indirect block

sz: Sector size; bs: Block size.

3.1.2.1. exFAT. In the exFAT file system, the main boot region is located at the beginning of the volume and spans 12 sectors, starting with the main boot sector in the first sector. This boot sector contains the bootstrapping code and key exFAT parameters and has a defined size of 512 bytes. If the physical sector size exceeds 512 bytes, unused space remains within the sector. This slack space can be exploited for data hiding. The actual sector size is defined at offset 108 within the boot sector and can be up to 4 KiB.

Sector 9 of the main boot region is reserved for OEM parameters and typically occupies only 480 bytes, leaving unused space that can be repurposed for data hiding. Sector 11 serves as a checksum sector, containing checksums calculated over the entire boot region. To modify the boot region without detection, the corresponding checksums must be recalculated and Sector 11 must be updated accordingly.

Immediately following the main boot region is a backup boot region with an identical structure, which provides the same potential for data hiding.

3.1.3. Allocation structures

Some file systems use bitmaps to track the allocation status of data blocks, where each bit corresponds to a single block: a value of one indicates that the block is allocated, while zero denotes an unallocated block. Since the total number of data blocks does not always align perfectly with the number of bits that fit into a block-sized bitmap, unused bits may remain—creating slack space. This slack can be exploited for data hiding, as previously demonstrated for the ext4 file system Göbel and Baier (2018a).

3.1.3.1. FAT. The FAT file system does not rely on a bitmap to track allocation status. Instead, it uses the File Allocation Table, which records the usage of each cluster. Similar to other file system structures, slack space can occur at the end of the FAT, particularly when it does not align perfectly with the underlying sector size—i.e., when the table ends in the middle of a sector. Since the FAT file system does not implement checksums for integrity verification, no further modifications are required when hiding data in this slack space.

3.1.3.2. exFAT. The exFAT file system uses an allocation bitmap to track the allocation status of blocks. A bit value of 0 indicates that a block is free, whereas a value of 1 indicates that it is in use. This approach differs from traditional FAT file systems, which determine block allocation directly through the FAT table by setting an entry to zero for free blocks. In exFAT, the FAT is still present and used to maintain block chains for fragmented files, but the allocation bitmap serves as the primary mechanism for identifying free blocks.

The allocation bitmap is stored in the data area of the volume, and its metadata—such as the start blocks and total size—is recorded in a dedicated directory entry. The size of the bitmap can be retrieved from the `data_length` field at offset 24 of this entry. Slack space may occur if the size of the bitmap is not an exact multiple of the block size.

Since exFAT does not employ checksums or other integrity

Table 4
Reserved space locations and capacities.

FS	Location	Description	Capacity
FAT32	FSInfo[a]	FSI_Reserved{1,2}	492 B
exFAT	Boot Region[a]	10th sector	Sector Size
NTFS	MFT	Entries 12-15	4 × 1024 B
Btrfs	Superblock	Offset 0 × 23B and 0xDCB	805 B
	inode	Offset 0x50	inode count x 32 B
ZFS	vdev label	First 16 KB	4 × 16 KB
	vdev label	After second vdev label	3.5 MB
	dnode	dn_pad3	dnode count x 32 B

[a] Applies also to backup structure.

verifications for standard data blocks, this slack space may be exploited for hiding data without detection by the file system.

3.1.4. Metadata

3.1.4.1. ZFS. In ZFS, a *block pointer* is a 128-byte structure used to reference a data block. Each file is represented by a *dnode*, which holds up to three direct block pointers, allowing it to address data up to three times the configured block size. When additional space is required, ZFS employs a hierarchy of indirect blocks. For example, if a file exceeds the capacity provided by the three direct block pointers, a level-1 block pointer is used to reference an indirect block, which in turn contains level-0 block pointers to the actual data blocks.

If an indirect block is not fully populated with block pointers, the remaining slack space can be used for data hiding. However, if checksumming is enabled, the corresponding checksum must also be updated to avoid detection.

3.2. Reserved space

As highlighted in the previous section, various reserved areas in file systems—often intended for alignment or future use—have already been exploited to hide data. In this section, we extend the analysis by identifying additional reserved regions that have not yet been examined for data hiding, both in the file systems discussed earlier and in others not yet considered. These details are summarized in Table 4.

3.2.1. File system data

3.2.1.1. FAT. In the FAT file system, the BIOS Parameter Block (BPB) includes fields—specifically, `BPB_FATSz16` for FAT12 or FAT16 and `BPB_FATSz32` for FAT32—that define the number of sectors allocated to each FAT. These values are typically set during formatting based on the volume size and block configuration. However, it is possible to manually set these fields to values larger than necessary, resulting in unused sectors at the end of each FAT. These surplus sectors can potentially be used to hide data.

3.2.1.2. NTFS. NTFS manages metadata through the Master File Table (MFT), a sequence of fixed-size file records (typically 1024 bytes, as defined in the boot sector). Each file and directory is represented by one or more of these records, which are functionally similar to inodes in Unix-based systems. An MFT record consists of a header, a series of variable-length attributes, and an standard end marker.

While MFT entries 12–15 are reserved for future use, entries 16–23 are also initially marked as unused and could potentially be exploited for data hiding. However, since these entries are not officially documented, they may be allocated by the file system during normal operation.

3.2.1.3. exFAT. In exFAT, the 10th sector of both the main and backup boot regions is reserved and can range from 512 bytes to 4 KiB, depending on the sector size. When hiding data in this sector, the corresponding checksums in the subsequent checksum sectors must be updated to avoid simple detection.

3.2.1.4. Btrfs. As described by Göbel et al. Göbel et al. (2024), Btrfs reserves 64 KiB of space preceding each superblock, which can be used to hide data. Beyond that, the superblock itself also contains unused and reserved areas suitable for data hiding. Each Btrfs superblock is 4096 bytes in size. The last 565 bytes—starting at offset `0xDCB`—are currently unused, and an additional 240 bytes at offset `0x23B` are reserved for future extensions. Together, these regions provide 805 bytes per superblock that can potentially be repurposed for concealed data.

The primary superblock is located at offset `0x10000`, with additional mirror copies stored at predefined locations.[5] The superblock checksum is stored in its first 32 bytes and covers the content starting from offset `0x20`. As such, any modifications to the reserved areas must be followed by a checksum update to maintain integrity and avoid detection.

3.2.2. FSInfo

FAT32 features a `FSInfo` structure, which is referenced via the `BPB_FSInfo` field in the boot sector, which stores additional metadata about the file system. The primary `FSInfo` sector usually resides in logical sector 1, with an additional backup copy at sector 7. This structure includes two reserved fields: `FSI_Reserved1` (480 bytes) and `FSI_Reserved2` (12 bytes). Although these fields are usually zeroed during formatting, they are not validated and can be repurposed for hidden data. Furthermore, the `FSInfo` structure contains the `FSI_Free_Count` and `FSI_Nxt_Free` fields, which store the number of free blocks and the next free block. However, with a combined size of only 8 bytes, these fields offer limited capacity, even when considering the additional `FSInfo` backup structure, and may be modified by the file system driver during normal operation.

3.2.3. Vdev labels

ZFS storage pools consist of one or more virtual devices (vdevs), which typically correspond to physical storage devices such as disks. Each vdev contains four 256 KB structures known as vdev labels: two are located at the beginning of the device, and two at the end.

The first 8 KB of each vdev label is reserved for a potential Volume Table of Contents (VTOC) and is intentionally left blank. The following 8 KB is marked as reserved for future use. Together, these sections provide 16 KB of reserved space per label that can potentially be used to hide data.

Additionally, ZFS reserves a 3.5 MB region between the second and third vdev label for future use. This area begins at offset `0x8000`, directly following the second label, and offers a significantly larger opportunity for data hiding.

3.2.4. Metadata

3.2.4.1. ZFS. 24 bytes of a *block pointer* in ZFS are reserved as padding for future use. These bytes are unused by default and may be leveraged to hide data. The final 32 bytes of the block pointer store a checksum for verifying the integrity of the referenced data block. However, since ZFS allows checksums to be disabled, this region may no longer be validated. In such cases, the checksum field itself could potentially be repurposed for data hiding. In addition, the *dnode* structure includes several padding fields, most notably `dn_pad3`, located at offset `0x48` and spanning 32 bytes. As the number of dnodes scales with the number of files in the file system, this reserved area may offer substantial cumulative capacity for data hiding.

3.2.4.2. Btrfs. In Btrfs, *inodes* store metadata for files and directories.

[5] https://btrfs.readthedocs.io.

Each inode is represented by the `btrfs_inode_item` structure, which is 160 bytes in size. Within this structure, 32 bytes at offset `0x50` are reserved for future use making them a potential target for data hiding. Since the number of inodes scales with the number of files in the file system, this can result in a considerable amount of aggregate hiding space. Since Btrfs uses checksums to ensure the integrity of its metadata, any modifications to these reserved fields may require updating the associated checksums to avoid detection.

3.3. Misuse of file system structures

In addition to slack and reserved space, various file system structures can be misused to conceal data.

3.3.1. Bad blocks

As previously discussed, several file systems, such as ext, FAT, and NTFS, allow data to be hidden by marking blocks as bad, causing the file system to ignore them. This method is not applicable to XFS, APFS, Btrfs and ZFS which delegate bad block handling to lower layers like hardware or drivers. In the case of exFAT, however, a similar hiding technique is possible. If a block is marked as allocated in the bitmap but its corresponding FAT entry is set to `0xFFFFFFF7`, the file system interprets the block as defective and ignores it. Manually specifying bad blocks this way can be used to create additional space to hide data.

3.3.2. Timestamp hiding

As previously noted, the nanosecond portions of timestamps can be exploited to conceal data. In this section, we extend the analysis to file systems not previously examined, providing an overview of available timestamps, their locations, precision, and the presence of integrity mechanisms. These details are summarized in Table 5.

As shown, file systems such as FAT, exFAT, and HFS + do not support nanosecond-resolution timestamps, making them unsuitable for this data hiding technique. In contrast, Btrfs and ZFS use 64-bit timestamps with nanosecond precision, splitting seconds and nanoseconds into the upper and lower 32 bits, respectively. While this method has already been applied to Btrfs Göbel et al. (2024), it has not yet been examined for ZFS.

In ZFS, timestamps are stored within *dnodes*, which serve a similar role to inodes in ext-based file systems. Like Btrfs, ZFS protects its metadata and data blocks with checksums (see Table 2). However, ZFS offers the option to disable checksumming, which enables the modification of timestamp fields without triggering integrity violations. This makes the 32-bit nanosecond portion of each timestamp a viable candidate for data hiding.

As a well-established technique, timestamp-based hiding offers a capacity that scales with the number of files. However, due to the volatility of timestamp fields during regular file system activity, it is best suited for static files where metadata is unlikely to be modified.

Table 5
Comparison of file timestamp properties across different filesystems.

FS	Time-stamps	Size (bit)	Nano part	Location	Checksum
ext2/3	m,a,c	32	0	inode	×
ext4	m,a,c,cr	64	32	inode	✓
NTFS	m,a,c,cr	64	24	MFT record	×
exFAT	m,a,cr	32	0	Directory entry	×
FAT	m,a,cr[a]	32,16[a]	0	Directory entry	×
APFS	m,a,c,cr	64	32	inode	✓
XFS	m,a,c,cr	64	32	inode	✓
ZFS	m,a,c,cr	64	32	znode	✓
Btrfs	m,a,c,cr	64	32	inode	✓
HFS+	m,a,c,cr	32	0	Catalog file entry	×

m: last modification time; a: last access time; c: last metadaten modification time; cr: creation time.

[a] Creation timestamp only 16bit.

4. Novel techniques for data hiding

While the previous section examined how existing data hiding techniques can be adapted to file systems where they have not yet been applied—and identified additional potential hiding locations—this section presents novel techniques that, to the best of our knowledge, have not been previously described in the literature. These methods leverage architectural properties and features unique to contemporary file systems to conceal data in new and previously unexplored ways.

4.1. Snapshots

Snapshots are a feature of modern file systems such as ZFS and Btrfs allowing users to preserve and later revert to a consistent state of the file system at a specific point in time. This functionality is typically enabled through the Copy-on-Write (CoW) mechanism, where modified data is written to a new location rather than overwriting the original. As a result, snapshots can be maintained efficiently, since only modified data consumes additional space.

In order to hide data using this feature, a snapshot can be created while sensitive files are still present. Afterward, the files can be deleted or moved from the active file system. However, since the snapshot retains references to the original data blocks, the deleted files remain intact and are not overwritten. By reverting to the snapshot, the data becomes accessible again. This technique offers a simple yet effective method for concealing information within the file system.

However, this method remains effective only as long as an analyst is unaware of the presence or relevance of certain snapshots or of the snapshot feature itself. In ZFS, for example, snapshots are stored in a hidden directory at the root of the file system: `.zfs/snapshot`. Although this directory is not displayed by default in standard utilities, it can still be accessed and inspected. Moreover, investigators can use administrative tools to enumerate available snapshots. While basic obfuscation techniques, such as misleading naming, may be applied, fully concealing the existence of a snapshot typically requires more invasive manipulation.

To achieve this, file system structures must be directly altered—for example, by removing the snapshot from the list of active snapshots. However, this can lead to the snapshot and its associated data being overwritten, depending on the specific file system's behavior. Consequently, such an approach is highly dependent on the file system's internal implementation.

4.2. Lower file slack

While slack space is a well-established target for data hiding across various structures, modern file systems introduce a new variant: lower file slack Hilgert et al. (2024). This form of slack emerges in stacked file systems, which do not store data directly on volumes but instead rely on a lower file system to store their data. In this case, files from the upper (stacked) file system are stored as ordinary files—lower files—on the underlying file system.

These lower files are often allocated in fixed-size extents (e.g., 4 KiB). If the upper file's content does not fully utilize the last extent, unused space remains between the actual end of the file's content and the end of the lower file—the *lower file slack*. Moreover, because lower files behave like regular files, arbitrary data can easily be appended to them without affecting the upper file system's view. If the lower file has reached a predefined maximum size, this appended data may not even be overwritten, creating what is referred to as *extra lower file slack*.

In our experiments, we successfully applied this technique to stacked file systems such as MooseFS and GlusterFS. In particular, MooseFS—with its 64 MiB maximum size for lower files—results in extra lower

file slack that is not overwritten, making it a particularly interesting location for data hiding that forensic analysts should be aware of.

4.3. Volume management structures

Contemporary file systems such as ZFS and Btrfs offer integrated volume management capabilities, allowing multiple physical disks to be combined into logical storage pools using configurations like mirrors or striped arrays. While these features enhance flexibility and redundancy, they also introduce new possibilities for data hiding.

Even in traditional software or hardware RAID setups, combining disks of unequal size—such as using a larger disk in a mirrored configuration—can leave unused space on the larger device. This residual space is typically ignored by the RAID system and remains unallocated, making it a viable location for concealing data without affecting normal operation. File systems like ZFS and Btrfs, which include native volume management features, can also span multiple asymmetric disks. As a result, similar slack space may arise within these configurations, offering comparable opportunities for data hiding.

Another potential method for hiding data in volume-managed file systems like ZFS or Btrfs involves exploiting how data is distributed across multiple physical devices, similar to RAID configurations. In mirrored setups, data could be hidden by modifying only one of the mirrors, assuming the file system does not regularly cross-verify them. If reads consistently occur from a single mirror and no active integrity checks are triggered, concealed data may remain undetected. Likewise, in parity-based configurations such as RAID-Z or Btrfs RAID5/6, unused or infrequently verified parity areas could be repurposed for data hiding.

A more advanced technique can leverage the file system's internal mapping mechanism, which translates logical block addresses to physical offsets on specific devices in the storage pool. By manipulating these mapping structures, such as block pointers in ZFS, to always resolve to a single mirror member or specific disk, the file system can be tricked to ignore other devices. This effectively isolates those devices from regular access, allowing hidden data to persist without interference from normal file system operations. However, this method may be easily detected within file systems that enforce validation.

5. Standardized corpus

Data hiding in file systems is a well-established and extensively studied topic. Consequently, the reliable detection of such hidden data remains a critical task in forensic analysis. Effective detection methods must be thoroughly developed, adopted, and evaluated to ensure comprehensive identification of hidden data.

While the Cyber Forensics Lab at the University of New Haven[6] offers an overview of various forensic datasets, there appears to be no dedicated including file system images with data hiding techniques. However, a standardized corpus is crucial for the evaluation of detection approaches, enabling reproducibility and ensuring comparability of results across different tools and methodologies.

One notable attempt to synthesize such data is the *fishy* framework. However, its repository has not been maintained in recent years and lacks coverage of several techniques discussed in this work. To address this gap, we developed and publicly release a comprehensive corpus focused on file system-based data hiding techniques.

The dataset comprises file system images across multiple file systems, each embedding hidden data according to predefined scenarios derived from current techniques. Each image is accompanied by a ground truth file specifying the location (e.g., inode slack space), offset, length, and an extracted copy of the hidden content for validation. Image creation was performed using existing tools or frameworks where applicable, but primarily carried out manually.

Table 6
Overview of data hiding scenarios in the corpus.

Scenario	Description
Scenario 1	Data hidden in general file slack space.
Scenario 2	Data hidden in file timestamps.
Scenario 3	Data hidden in bad units.
Scenario 4	Data hidden in additional data units.
Scenario 5	Data hidden in slack space structures.
Scenario 6	Data hidden in reserved areas of structures.
Scenario 7	Data hidden in hidden snapshots.
Scenario 8	Data hidden in lower file slack.
Scenario 9	Data hidden in the slack space of physical members in pooled file systems.

The dataset currently includes the scenarios as shown in Table 6.

For some cases, we provide multiple images per scenario and file system to reflect different data hiding methods (e.g., data hidden in different file system structures). All such variations are documented in the corresponding ground truth files distributed with the dataset.[7]

6. Conclusion

In the field of anti-forensics, data hiding in file systems remains a well-established technique that has been the focus of ongoing research for many years, with new methods continually emerging to target both legacy and modern file systems. Some techniques, such as hiding data in file slack, are simple and easy to deploy, while others are more complex and difficult to detect. This dual nature reinforces the importance of understanding and addressing file system-based data hiding, making it a critical area of focus for forensic analysts and tool developers.

Our work provides a comprehensive overview of established data hiding methods and the file systems they target. By revisiting and extending these techniques, we demonstrate that the potential for file system-based data hiding is far from exhausted—even as modern features like checksumming introduce new challenges and increase the complexity of such methods. Beyond this, we present three novel approaches that exploit specific features of contemporary file systems, which underscores the ongoing innovation in this space.

Given the evolving landscape of data hiding methods, it is crucial to continuously develop, evaluate, and refine detection strategies to ensure comprehensive coverage. To support this effort, we present and release the first standardized corpus for data hiding techniques in file systems. Rather than offering a tool that could be misused, this corpus provides a controlled and transparent foundation for evaluating detection tools and techniques in a reproducible and comparable manner to foster research in this area.

References

Beebe, N., Mandes, S., Stuckey, D., 2009. Digital forensic implications of zfs. Digit. Invest. 6, S99–S107.
Berghel, H., Hoelzer, D., Sthultz, M., 2008. Data hiding tactics for windows and unix file systems. Adv. Comput. 74, 1–17.
Carrier, B., 2005. File System Forensic Analysis. Addison-Wesley Professional.
Davis, J., MacLean, J., Dampier, D., 2010. Methods of information hiding and detection in file systems. In: 2010 Fifth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering. IEEE, pp. 66–69.
Dillon, S., 2006. Hide and Seek: Concealing and Recovering Hard Disk Data, vol. 35. James Madison University Infosec Techreport, p. 17.
Eckstein, K., Jahnke, M., 2005. Data hiding in journaling file systems. In: DFRWS.
Göbel, T., Baier, H., 2018a. Anti-forensic capacity and detection rating of hidden data in the ext 4 filesystem. In: Peterson, G., Shenoi, S. (Eds.), Advances in Digital Forensics XIV. Springer International Publishing, Cham, pp. 87–110.
Göbel, T., Baier, H., 2018b. Fishy-a framework for implementing filesystem-based data hiding techniques. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 23–42.

[6] https://datasets.fbreitinger.de/datasets/.

[7] https://github.com/fkie-cad/hide-and-seek-dataset.

Göbel, T., Baier, H., Türr, J., 2024. Generating useable and assessable datasets containing anti-forensic traces at the filesystem level. In: IFIP International Conference on Digital Forensics. Springer, pp. 225–246.
Göbel, T., Türr, J., Baier, H., 2019. Revisiting data hiding techniques for apple file system. In: Proceedings of the 14th International Conference on Availability, Reliability and Security, pp. 1–10.
Göbel, T., Baier, H., 2018. Anti-forensics in ext4: on secrecy and usability of timestamp-based data hiding. Digit. Invest. 24, S111–S120. https://doi.org/10.1016/j.diin.2018.01.014. URL: https://www.sciencedirect.com/science/article/pii/S174228761830046X.
Hassan, N.A., Hijazi, R., 2016. Data Hiding Techniques in Windows OS: a Practical Approach to Investigation and Defense. Syngress.
Heeger, J., Yannikos, Y., Steinebach, M., 2021. Exhide: hiding data within the exfat file system. In: Proceedings of the 16th International Conference on Availability, Reliability and Security, pp. 1–8.
Heeger, J., Yannikos, Y., Steinebach, M., 2022. An introduction to the exfat file system and how to hide data within. J. Cyber Sec. Mob. 239–264.
Hilgert, J.N., Lambertz, M., Baier, D., 2024. Forensic implications of stacked file systems. Forensic Sci. Int.: Digit. Invest. 48, 301678.
Hilgert, J.N., Lambertz, M., Plohmann, D., 2017. Extending the sleuth kit and its underlying model for pooled storage file system forensic analysis. Digit. Invest. 22, S76–S85.
Huebner, E., Bem, D., Wee, C.K., 2006. Data hiding in the ntfs file system. Digit. Invest. 3, 211–226. https://doi.org/10.1016/j.diin.2006.10.005. https://www.sciencedirect.com/science/article/pii/S1742287606001265.
Khan, H., Javed, M., Khayam, S.A., Mirza, F., 2011. Designing a cluster-based covert channel to evade disk investigation and forensics. Comput. Secur. 30, 35–49. https://doi.org/10.1016/j.cose.2010.10.005. https://www.sciencedirect.com/science/article/pii/S016740481000088X.
Kim, D., Lee, Y.K., Jeong, J., 2022. Null byte injection: anti-forensic technique for data hiding in fat32 file system. In: Proceedings of the Twenty-Third International Symposium on Theory, Algorithmic Foundations, and Protocol Design for Mobile Networks and Mobile Computing, pp. 265–270.
Koolhaas, A., van Steenbergen, W., 2020. Apfs slack analysis and detection of hidden data. Secur. Netw. Eng. 2019–2020.
Krenhuber, A., Niederschick, A., 2007. Forensic and anti-forensic on modern computer systems. Johannes Kepler Universität Linz 11.
Liu, S.f., Pei, S., Huang, X.y., Tian, L., 2009. File hiding based on fat file system. In: 2009 IEEE International Symposium on IT in Medicine & Education, pp. 1198–1201. https://doi.org/10.1109/ITIME.2009.5236280.
Neuner, S., Voyiatzis, A.G., Schmiedecker, M., Brunthaler, S., Katzenbeisser, S., Weippl, E.R., 2016. Time is on my side: Steganography in filesystem metadata. Digit. Invest. 18, S76–S86. https://doi.org/10.1016/j.diin.2016.04.010. https://www.sciencedirect.com/science/article/pii/S1742287616300433.
Piper, S., Davis, M., Manes, G., Shenoi, S., 2005. Detecting hidden data in ext2/ext 3 file systems. In: Pollitt, M., Shenoi, S. (Eds.), Advances in Digital Forensics. Springer US, Boston, MA, pp. 245–256.
Piper, S., Davis, M., Shenoi, S., 2006. Countering hostile forensic techniques. In: Olivier, M.S., Shenoi, S. (Eds.), Advances in Digital Forensics II. Springer US, Boston, MA, pp. 79–90.
Srinivasan, A., Pieper, B., 2022. Steganography with filesystem-in-slackspace. In: 2022 IEEE International Conference on Networking, Architecture and Storage (NAS). IEEE, pp. 1–4.
Toolan, F., Humphries, G., 2025a. Data hiding in symbolic link slack space. Forensic Sci. Int.: Digit. Invest. 53, 301919. https://doi.org/10.1016/j.fsidi.2025.301919. https://www.sciencedirect.com/science/article/pii/S2666281725000587.
Toolan, F., Humphries, G., 2025b. Data hiding in the xfs file system. Forensic Sci. Int.: Digit. Invest. 52, 301884. https://doi.org/10.1016/j.fsidi.2025.301884. https://www.sciencedirect.com/science/article/pii/S266628172500023X.