fishy - A Framework for Implementing Filesystem-Based Data Hiding Techniques

Thomas Göbel(✉) and Harald Baier

da/sec - Biometrics and Internet Security Research Group,
Hochschule Darmstadt, Darmstadt, Germany
{thomas.goebel,harald.baier}@h-da.de

Abstract. The term anti-forensics refers to any attempt to hinder or even prevent the digital forensics process. Common attempts are to hide, delete or alter digital information and thereby threaten the forensic investigation. A prominent anti-forensic paradigm is hiding data on different abstraction layers, e.g., the filesystem layer. In modern filesystems, private data can be hidden in many places, taking advantage of the structural and conceptual characteristics of each filesystem. In most cases, however, the source code and the theoretical approach of a particular hiding technique is not accessible and thus maintainability and reproducibility of the anti-forensic tool is not guaranteed. In this paper, we present *fishy*, a framework designed to implement and analyze different filesystem-based data hiding techniques. *fishy* is implemented in Python and collects various common exploitation methods that make use of existing data structures on the filesystem layer. Currently, the framework is able to hide data within ext4, FAT and NTFS filesystems using different hiding techniques and thus serves as a toolkit of established anti-forensic methods on the filesystem layer. *fishy* was built to support the exploration and collection of various hiding techniques and ensure the reproducibility and expandability with its publicly available source code. The construction of a modular framework played an important role in the design phase. In addition to the description of the actual framework, its current state, its use, and its easy expandability, we also present some hiding techniques for various filesystems and discuss possible future extensions of our framework.

Keywords: Anti-forensics · Anti-anti-forensics · Digital forensics
Data hiding · File system analysis · ext4 · NTFS · FAT

1 Introduction

Since there are new cyber attacks and threats on a daily basis, the topic *anti-forensics* is now an important part of digital forensics. As cybercrime increases and attacks become more frequent and sophisticated, research in the field of anti-forensics is becoming increasingly relevant. In order to gather reliable evidence

F. Breitinger and I. Baggili (Eds.): ICDF2C 2018, LNICST 259, pp. 23–42, 2019.
https://doi.org/10.1007/978-3-030-05487-8_2

during a digital forensic investigation, it is important to know about appropriate countermeasures and mitigation strategies against anti-forensic methods, techniques and tools. Current data hiding approaches vary from encryption techniques on different abstraction layers, various steganography types, data contraception methods, hard disk and filesystem manipulation, to network-based data hiding [1].

In recent years, various taxonomies have been presented that categorize and summarize known anti-forensic techniques and methods [2–4]. The most recent taxonomy was proposed by Conlan et al. [1]. Their extended taxonomy distinguishes between hiding techniques on the hard disk layer and hiding techniques on the filesystem layer. In fact, a prominent anti-forensic paradigm is hiding data on different abstraction layers, e.g., the filesystem layer. As an example of potential hiding places in the filesystem, the authors mention alternate data streams and slack spaces.

Various proof of concept implementations and tools for different hiding techniques have been released in the past. These are known to the forensic community[1]. However, in most cases the authorship of the previously published tools is unknown and the source code or the documentation of the tools is inaccessible. For this reason, the maintenance of existing anti-forensic projects and the reproducibility of the source code of available tools are not guaranteed. Moreover, the theoretical approaches of potential hiding techniques in the filesystems are often not published. In addition, there is usually a separate tool for each hiding technique that can only be used with a specific filesystem (e.g., the anti-forensic tool called *slacker* can only conceal data within NTFS slack space). A common anti-forensic toolkit that collects various hiding techniques is missing so far. Furthermore, many of the existing tools are obsolete as they relate to old filesystems, such as ext2.

The aforementioned problems make research in the field of anti-forensics considerably more difficult. The inconsistent provision of the tools not only prevents the validation, reproducibility and further development of the hiding approaches themselves, but also the possible evaluation of existing forensic suites. The importance of digital forensic tool testing in order to get authentic, accurate and reliable evidence is obvious and well known to the members of the digital forensic community [4]. If the forensic software is unable to detect hidden data on the medium for some reason, this could cause courts to render judgments based on inaccurate or incomplete evidence [5]. This scenario gives an attacker the opportunity to exploit the weakness of forensic software to hide relevant evidence [6].

1.1 Contribution

The practical, uniform and publicly available implementation of multiple hiding techniques is advantageous for a later evaluation of forensic software components.

[1] http://www.forensicswiki.org/wiki/Anti-forensic_techniques#Generic_Data_Hiding (last accessed 2018-05-10).

Therefore, we want to introduce our Python-implemented toolkit and framework called *fishy* - filesystem hiding techniques in Python. While [7] presents the current state of the framework and already implemented hiding techniques, this paper focuses on the framework architecture, the usage of the framework, and instructions on how to add further hiding techniques. In this paper, we also publish the link that refers to our GitHub repository, which includes the full source code of the framework[2].

The toolkit is intended to introduce people to the concept of established anti-forensic methods associated with data hiding on the filesystem layer. *fishy* is able to obfuscate arbitrary data in a way that conventional file access methods cannot recognize the concealed content. The new framework allows the digital forensic community to implement, categorize and test filesystem-based data hiding techniques. Since the framework is open source, it can be extended by anyone and at any time with additional hiding techniques. The modular structure of the framework allows us (i) to easily implement further custom hiding techniques, (ii) to manipulate multiple filesystem types with one and the same framework, and (iii) to add compatibility for additional modern filesystems (e.g., APFS, Btrfs, ReFS, or XFS) in the future without having to change the basic functions of the framework. Currently, *fishy* supports the most popular filesystems: ext4, FAT and NTFS. The toolkit provides a command line interface that can be used to hide data. Furthermore, the already implemented hiding techniques can be used in other projects by importing *fishy* as a library, e.g., to analyze whether existing hiding techniques are recognized by common forensic software.

To the best of our knowledge, there is currently no actively maintained toolkit for filesystem-based data hiding techniques, other than *fishy*. As *fishy* aims to provide an easy to use framework for creating new hiding techniques, this project might be useful for security researchers and members of the digital forensic community who are concerned with security issues and data hiding attacks.

In addition to a detailed description of the actual framework architecture, we present some exemplary hiding techniques and demonstrate in detail how the framework can be used and how additional hiding techniques can be integrated.

1.2 Structure of the Paper

The rest of this paper is organized as follows: Sect. 2 presents some related work, especially relevant information about anti-digital forensics and previously published tools that allow to manipulate or hide data within filesystems. In Sect. 3, we explain in detail the core design principles and the modular structure of our framework. Section 4 gives some background information about the manipulated filesystem data structures and a brief explanation of all the hiding techniques that have been implemented so far. In addition, other hiding techniques, which are currently under development, are mentioned. Section 5 shows how to use the framework by explaining the typical command structure and the integration of new hiding techniques into the existing framework. Further technical details

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

can be found within the module reference in our GitHub repository. It documents the most important modules and classes, which you might use, if you want to integrate *fishy* into your own projects. In Sect. 6, we evaluate the current state of our implementation and present limitations of the framework. Section 7 concludes the paper and presents further tasks that supplement the work done so far.

2 Related Work

In this section, we review related work with respect to anti-forensics in general and previously published data hiding methods and tools for various filesystems in particular.

2.1 Anti-Forensics

Well known definitions for the term anti-forensics were proposed by Rogers [2] and Harris [3]. The most recent contribution in the field of anti-forensics is a paper provided by Conlan et al. [1]. The authors summarized previous definitions and defined the term anti-forensics as "any attempts to alter, disrupt, negate, or in any way interfere with scientifically valid forensic investigations". To better distinguish different anti-forensic techniques, Rogers [2] subdivided them into the following four categories: (i) data hiding, (ii) artifact wiping, (iii) trail obfuscation, and (iv) attacks against the computer forensic process and tools. Conlan et al. [1] expand Rogers' widely accepted taxonomy by a fifth category: (v) possible indications of anti-digital forensic activity, and came up with a more comprehensive and up-to-date taxonomy. Now that a variety of anti-forensic techniques are known, the authors additionally subdivide the five categories into several subcategories. This allows a more precise categorization of new anti-forensics techniques. Hiding techniques, as presented in this paper, can be assigned to the *data hiding* category and, in particular, to its subcategory *filesystem manipulation*.

2.2 Hiding Data in Filesystems

Hiding data in filesystem metadata was carried out by Anderson et al. [8] along with the development of a steganographic filesystem. This resulted in StegFS [9], a steganographic filesystem based on ext2, which allowed people to deny the existence of hidden data. Meanwhile, many other hiding places are known which make it possible to conceal sensitive data without affecting the actual function of the filesystem. In addition to the well-known hiding places, such as file slack or alternate data streams [10], hiding places in less popular filesystem structures can be used to hide data, such as the slack space of several block bitmaps or inode bitmaps [11]. Likewise, existing data structures, such as timestamps [12,23], or reserved areas for future filesystem extensions, such as reserved group descriptor table blocks [13], may be used to hide arbitrary data.

In digital forensic investigations, the knowledge of these, at first glance inconspicuous, hiding places definitely plays a decisive role in securing all incriminating digital traces. There are a few papers that deal with different filesystem hiding methods and countermeasures. Our list of existing publications indicates which filesystem is discussed in the respective paper. Eckstein et al. [14] show how to hide data in file and directory slack space or reserved inodes in ext3 and in alternate data streams in NTFS. Piper et al. [15] show how to hide data in the partition boot sector, in reserved portions of the superblock and in redundant superblocks of ext2/ext3 filesystems. Grugq [16] demonstrates further hiding places in ext2, such as the manipulation of directory entries and the utilization of reserved space in superblocks, group descriptors and inodes. Huebner et al. [17] present various approaches to hide data in metadata files of NTFS, e.g., in the $BadClus file, the $DATA attribute and the $Boot file. Krenhuber et al. [18] show how to hide data in NTFS using file slack, faked bad clusters, additional cluster allocation and alternate data streams. Other methods for ext filesystems, such as file slack, the mount procedure and extended file attributes, are shown. Berghel et al. [19] present approaches to conceal data using the slack space of various ext2/ext3 data structures, such as superblocks, group descriptor tables and directory entries. The authors also discuss some hiding places for NTFS, for instance the $Bitmap file, additional $FILE attributes or MFT entries.

It should be mentioned here that many of the existing publications on this topic are outdated and there are only a few recent contributions. A slightly newer work analyzes the most important filesystem data structures of ext4 in a digital forensic way [13]. Several potential data hiding places in ext4 are mentioned (e.g., group descriptor growth blocks, data structures in uninitialized block groups and Htree nodes), but not explored any further. The most recent publication in this field presents, analyzes and evaluates various anti-forensic techniques for ext4 and verifies whether previously mentioned hiding techniques still work in ext4 since most of them were developed for ext2/ext3 [11]. Unfortunately, this paper only discusses ext4 and does not focus on NTFS.

2.3 Available Data Hiding Tools

In our literature research on existing tools for filesystem-based data hiding techniques we found only a few working tools. None of these provide a consistent interface with support for multiple filesystems and various hiding techniques. For most of them it seems that development has been stopped. It is also noticeable that the data set provided by Conlan et al. [1], which collects 308 anti-forensic tools in total, only includes two tools (*bmap* and *slacker*) in its subcategory *filesystem manipulation*.

Three filesystem-based data hiding tools, that seem to be used more often, are *bmap*, *slacker.exe* and *FragFS* [21,22]. *bmap* is a Linux tool for hiding data in the NTFS slack space. However, this project seems to have no official website or any trustworthy repository, i.e., development of the tool probably stopped. Sources can only be found on some questionable websites[3]. *slacker.exe* is also a tool for hiding data in NTFS slack space, but is built for Windows. It was developed by Bishop Fox and integrated into the Metasploit framework under the name *Metasploit Anti-Forensic Investigation Arsenal (MAFIA)* along with some other anti-forensic tools (e.g., *Timestomp* - a tool to manipule NTFS timestamps)[4]. However, the tool is currently not available for download. *FragFS* is able to hide data within the last 8 bytes of an MFT entry in the NTFS Master File Table [20]. Available download links were not found here either.

In addition, several anti-forensics tools have been developed in the past for the ext filesystem [16]. Most of them, however, are outdated because they were originally developed for ext2 instead of the current version ext4. First, *Rune FS*[5] was able to hide data by taking advantage of a bug in the Coroner's Toolkit. Second, *Waffen FS* adds an ext3 journal to an ext2 filesystem and conceals up to 32 MB of data. Third, *KY FS* manipulates a directory entry as if the entry is not used to, subsequently, hide data in the directory entry. Fourth, *Data Mule FS* is able to conceal about 1 MiB data in a 1 GiB large ext2 image using all reserved areas in superblocks, group descriptors and inodes.

Two newer tools can hide data within timestamps. *Timestamp-Magic* [23] conceals data within the nanoseconds part of multiple timestamps in the inode table of an ext4 filesystem. *TOMS* [12] demonstrates a similar approach using the filename attribute of MFT entries in NTFS.

When reviewing existing tools it is noticeable that, with a few exceptions, the source code of the tools is often no longer publicly available. In addition, many of the tools were originally developed for earlier filesystem versions and development has been discontinued in the meantime. *fishy* allows a consistent implementation of hiding techniques that will facilitate the reproducibility and evaluation of forensic software in the future.

3 Framework Architecture

The following section explains in detail the basic architecture and the modular structure of the *fishy* framework. The depicted architecture overview gives an introduction to *fishys*' core design principles and structures. The key modules and classes someone might use to integrate *fishy* into their own projects are mentioned here. The command structure and its use are described in Sect. 5.

In the design process, we made sure that the architecture is as modular as possible. Different layers allow to encapsulate functionalities. The flowchart diagram in Fig. 1 represents the logical procedure of using a hiding technique. What we can see in the flowchart diagram is that the `Command Line Interface (CLI)` parses the given command line parameters and calls the appropriate `Hiding Technique Wrapper`. The `Hiding Technique Wrapper` then checks

[3] https://packetstormsecurity.com/files/17642/bmap-1.0.17.tar.gz.html (last accessed 2018-05-10).

[4] http://www.bishopfox.com/resources/tools/other-free-tools/mafia/ (last accessed 2018-05-10).

[5] http://index-of.es/Linux/R/runefs.tar.gz (last accessed 2018-05-10).

for the filesystem type of the given filesystem image and calls the respective `Hiding Technique` implementation for this filesystem. The modular structure of the framework is clearly visible here by treating the three different filesystems already implemented (ext4, FAT and NTFS) separately. In case of calling the write method, the `Hiding Technique` implementation returns metadata needed to restore the hidden data later. Currently, this metadata is written to the disk, using a simple JSON data structure. As future work it is considered to hide the metadata itself in the filesystem with a suitable hiding technique.

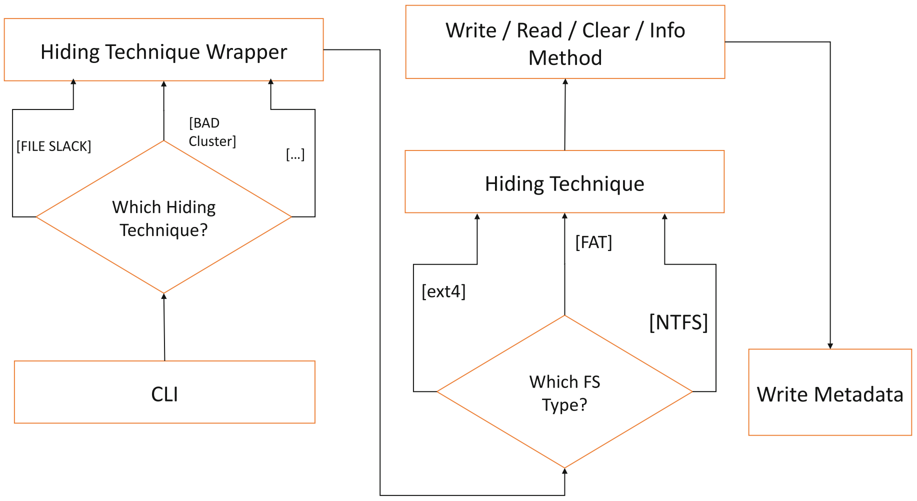

Fig. 1. Overview of the modular framework structure of *fishy*.

The command line argument parsing part is implemented in the *cli.py* module. `Hiding Techniques Wrapper` are located in the root module. They adopt converting input data into streams, casting/reading/writing hiding technique specific metadata and calling the appropriate methods of those hiding technique specific implementations. To detect the filesystem type of a given image, the `Hiding Technique Wrapper` use the *filesystem_detector* function which uses filesystem detection methods implemented in the particular filesystem module. Several filesystem specific `Hiding Technique` implementations provide at least a write, read and clear method to (i) hide data in the filesystem, (ii) to read or restore hidden content, and (iii) to delete previously hidden data. `Hiding Technique` implementations use either *pytsk3*[6] to gather information of the given filesystem or use custom filesystem parsers which are then located within the particular filesystem package.

[6] Python bindings for The Sleuth Kit: https://github.com/py4n6/pytsk (last accessed 2018-05-14).

3.1 Command Line Interface

As depicted in Fig. 1, the `CLI` module takes care of parsing the command line arguments and, depending on the given subcommand, calls the appropriate `Hiding Technique Wrapper`. The `CLI` thus forms the user interface of our toolkit.

Each hiding technique is accessible with a special subcommand, which itself defines further options. The `CLI` must be able to read data, that the user wants to hide, either from stdin or from a file. Previously hidden data that the user wants to recover is returned to either stdout or a specified file. If reading data from a file, the `CLI` is in charge of turning the content into a buffered stream, on which the hiding technique operates in the subsequent process.

3.2 Hiding Technique Wrapper

Each type of hiding technique has its own wrapper. This `Hiding Technique Wrapper` gets called by the `CLI` and subsequently calls the filesystem specific `Hiding Technique`, based on the filesystem type. As previously mentioned, to detect the filesystem type, the *filesystem_detector* function is called.

In order to find the correct offset of hidden data, read and clear methods of the hiding techniques require some metadata, which is gathered during a write operation. For this reason, the `Hiding Technique Wrapper` is responsible for reading and writing metadata files and providing hiding technique specific metadata objects for read and write methods. If the user wants to restore previously hidden data and store the information in a file instead of putting it to stdout, the `Hiding Technique Wrapper` is responsible for writing that file.

3.3 Hiding Technique

Multiple `Hiding Technique` implementations do the real work of this toolkit. Every `Hiding Technique` must at least offer a write, read and clear method in order to conceal new data and to restore or to delete hidden content. These methods must operate on streams only to ensure high reusability and reduce boilerplate code. All hiding techniques in the framework are called by the `Hiding Technique Wrapper`. The clear method must overwrite all hidden data with zeros and leave the filesystem in a consistent state. If an error occurs during the write process, i.e., while the private data is hidden, already written data must be deleted before exiting the program.

To get the required information about the current filesystem, hiding techniques use either the *pytsk3* library or a filesystem parser implementation located in the appropriate filesystem package.

If a hiding technique relies on some specific metadata (e.g., the exact offset of a MFT entry in NTFS) to restore hidden data, it must implement a hiding technique specific metadata class. This specific metadata class is used during the write process to store all information that is necessary for the recovery of the hidden data. The write method must return this metadata instance, so that the *Hiding Technique Wrapper* can serialize it and pass it to the read and clear methods.

Hiding techniques may implement further methods that are relevant for the hiding procedure, such as sub-methods that are relevant to actually hide the data in the right place or to split larger data into smaller parts to hide them in multiple data structures.

3.4 Filesystem Detector

The *filesystem_detector* is a simple wrapper function to unify calls to the filesystem specific detection functions, which are implemented in the corresponding filesystem package.

3.5 Metadata Handling

To be able to restore hidden data, most hiding techniques need some additional information. These information are stored in a JSON metadata file. The main-metadata class called *fishy.metadata* provides functions to read and write metadata files and to automatically decrypt or encrypt the metadata if a password is provided. The main purpose of this class is to ensure that all metadata files have a similar reasonable data structure. The program can thus recognize at an early point that, for example, the user is using a wrong hiding technique to restore hidden data.

When implementing a new hiding technique, this technique must also implement its own hiding technique-specific metadata class. The hiding technique itself therefore defines which data will be stored later during the hiding process. The write method then returns this hiding technique-specific metadata class which then gets serialized and stored in the main-metadata class.

4 Implemented Hiding Techniques and Current Work

An overview of all currently implemented hiding techniques in the framework, and hiding techniques that are stilfHides arbitrary data in l in development or are about to be added in the near future, is given in Table 1. Already finished hiding techniques are marked with the ✓ symbol, hiding techniques that are still in progress are marked with the ✗ symbol. The - symbol addresses hiding techniques which are filesystem specific, i.e., the respective data structure is missing in other filesystems. The table also includes a brief explanation of each implemented hiding technique. Moreover, all hiding techniques are briefly evaluated for their gained capacity, stability and their detection probability. *Capacity* addresses, how much data can be hidden by using the respective hiding technique, since an attacker is interested in storing a reasonable amount of data. *Detection probability* means the difficulty of finding the concealed artifacts, i.e., the likelihood that a forensic investigator discovers hidden data. For example, the default filesystem check or the standard GUI filesystem interface should not

Table 1. Overview of currently implemented **Hiding Techniques** (function call based on the respective subcommand) and work in progress of the framework *fishy* for different filesystems (**FS**) and our rating according to available *Capacity* (**C**), *Detection probability* (**D**) and *Stability* (**S**) (○=low; ◐=medium; ●=high).

Hiding Technique (Subcommand)	**FS**	**FS**	**FS**	**Description**	**C**	**D**	**S**
	FAT	NTFS	EXT4				
fileslack	✓	✓	✓	Exploitation of File Slack	●	◐	○
mftslack	-	✓	-	Exploitation of MFT Entry Slack	●	●	○
ads	-	✗	-	Use of Alternate Data Streams	●	●	◐
addcluster	✓	✓	✗	Additional Cluster/Block Allocation	●	◐	○
badcluster	✓	✓	✗	Bad Cluster/Block Allocation	●	◐	●
reserved_gdt_blocks	-	-	✓	Exploitation of reserved GDT Blocks	●	●	◐
superblock_slack	-	-	✓	Exploitation of Superblock Slack	◐	●	●
superblock_reserved	-	-	✗	Use of reserved space in Superblocks	○	◐	●
superblock_backups	-	-	✗	Exploitation of Superblock Backups	◐	◐	●
osd2	-	-	✓	Use of unused Inode Field osd2	○	◐	●
obso_faddr	-	-	✓	Use of unused Inode Field obso_faddr	○	◐	●
nanoseconds	-	✗	✓	Use of Nanoseconds Timestamp Part	◐	○	●
bootsector	✗	✗	✗	Exploitation of Partition Bootsector	○	●	●
null_dir_entries	-	-	✗	Exploitation of Directory Entries	●	◐	○
gdt_slack	-	-	✗	Exploitation of GDT Slack Space	◐	●	◐
groupdescr_reserved	-	-	✗	Use of reserved space in Group-Desc.	○	◐	●
gdt_backups	-	-	✗	Exploitation of GDT Backup Copies	●	◐	◐
blockbitmap_slack	-	-	✗	Exploitation of Block Bitmap Slack	○	●	●
inodebitmap_slack	-	-	✗	Exploitation of Inode Bitmap Slack	○	●	●
inode_slack	-	-	✗	Exploitation of Inode Record Slack	●	◐	◐
inode_reserved	-	-	✗	Use of reserved space in Inode Struct	●	◐	●
uninit_datastructure	-	-	✗	Exploitation of Data Structures in Uninitialized Block Groups	●	◐	○

disclose hidden data. *Stability* describes whether hidden data remains in the filesystem without complications, i.e., the possibility of data being overwritten is low.

As shown in Table 1, the development of many other hiding techniques is still in progress right now, for example the transfer of the nanosecond hiding

approach to the NTFS filesystem [12]. Besides, a major focus is on the integration of additional ext4 hiding techniques, as a recent work by Göbel and Baier [11] reveals many interesting hiding techniques for this filesystem. Nevertheless, the integration of the upcoming Windows filesystem ReFS and the upcoming Linux filesystem Btrfs, as well as some completely new hiding techniques are considered as future tasks. Members of the forensic community are also invited to add further filesystems and new hiding techniques to our framework.
We now present the manipulated filesystem data structures and give a brief explanation of each hiding technique that has been implemented so far.

4.1 File Slack

If a file is smaller than the cluster or block size of the filesystem, writing this file will result in some unusable space, which starts at the end of the file and ends at the end of the cluster/block. The remaining space can be used to hide data and is in general called File Slack [24, p. 187].

Most filesystem implementations of FAT and NTFS pad the RAM Slack with zeros, nowadays. This padding behavior must be honored by our implementation, as non-zero values in this area would be suspicious to any observer.

In case of ext4 filesystems, most implementations pad the complete File Slack with zeros, making the distinction between RAM and Drive Slack unnecessary but also making the detection of hidden data more likely. Our implementation for ext4 therefore calculates the end of a file on the filesystem and writes data into the following File Slack until no data is left or the end of the current block is reached.

4.2 MFT Entry Slack

The Master File Table contains the necessary metadata for every file and directory stored in a NTFS partition. An MFT entry does not have to fill up all of its allocated bytes, which often leads to some unused space at the end of an entry. The MFT entry slack is an suitable place to hide data inconspicuously.

NTFS uses a concept called Fixup [24, p. 253] for important data structures, such as the MFT, in order to detect damaged sectors and corrupt data structures. When an MFT entry is written to the disk, the last two bytes of each sector are replaced with a signature value. To avoid damaging the MFT, it is important to not overwrite the last two bytes of each sector when hiding data in the MFT entry slack. Besides this measure, the framework is able to write a copy of the hidden data in $MFT to corresponding entries in $MFTMirr to avoid detection by a simple `chkdsk` [24, p. 219].

4.3 Bad Cluster Allocation

If a sector or a cluster of sectors is damaged, read and write operations would lead to faulty data. Therefore the filesystem marks the affected area as bad clusters.

The filesystem saves the addresses for future reference and won't allocate them to a file or directory anymore. By marking some actually free cluster or blocks as faulty ones, we can reserve them to hide data in them.

In NTFS, affected areas are saved in an MFT file entry called $BadClus, the entries in this file will be ignored. In FAT, clusters are marked as bad in the File Allocation Table. In ext4, there is a list of bad blocks in the reserved inode 1 [24, p. 183, p. 225, p. 293].

4.4 Additional Cluster Allocation

Clusters are either unallocated or allocated to a file. By allocating an additional, actually unallocated, cluster to a file, the filesystem does not attempt to allocate or write data to that cluster, so that data can be hidden in that cluster [17].

If the file the cluster is allocated to grows in size and exceeds the boundary of its originally allocated clusters, the file will grow into the additionally allocated cluster and overwrite the hidden data. For this reason, a file that is unlikely to grow should be preferred as a carrier to hide data.

4.5 Reserved Group Descriptor Table Blocks

Reserved GDT blocks of an ext4 filesystem are not used until the filesystem is extended and group descriptors are written to this location. The reserved GDT blocks are located behind the group descriptors and are repeated in each backup copy. The number of copies varies depending on the sparse_super flag, which limits the copies of the reserved GDT blocks to block groups whose group number is either 0 or a power of 3, 5, or 7 [13].

Since the GDT growth blocks are really big reserved areas, this hiding method is quite obvious. Therefore our implementation skips the primary reserved GDT blocks before embedding some data. This prevents `e2fsck` from noticing these flaws in the filesystem. A big advantage of this technique is the high capacity. On the other hand, hidden data is overwritten in the case of a filesystem extension.

4.6 Superblock Slack

Depending on the block size, there is an acceptable amount of slack space behind the actual content of the Superblock which is repeated multiple times across the ext4 filesystem. The amount of copies of the Superblock depends on the sparse_super flag, i.e., less space to hide data if the flag is set [19].

The hiding technique collects all block numbers of the Superblock copies from each block group, taking the sparse_super flag into account. Data then gets written to the slack space of each of these blocks, considering the filesystem block size. This hiding technique benefits from the Superblock's characteristics, resulting in a safe storage because this data structure does not get overwritten. But as with all slack space hiding methods, hidden data is easy to find.

4.7 osd2, obso_faddr

The osd2 hiding technique uses the last two bytes of the 12 byte osd2 field, which is located at 0x74 in each inode of an ext4 filesystem. This field only uses 10 bytes at max, depending on the tag being either *linux2*, *hurd2* or *masix2* [25].

To hide data, the method writes data directly to the two bytes in the osd2 field in each inode, which address is taken from the inode table, until there is either no inode or no data left. Available space is small, but hidden data might be tough to find since data is distributed over several inodes across the filesystem.

The obso_faddr field in each inode at 0x70 is an obsolete fragment address field of 32bit length. This technique works accordingly to the osd2 technique, has the same advantages and flaws but can hide twice the data.

4.8 Nanosecond Timestamps

Modern filesystems like NTFS or ext4 support nanosecond precise timestamps. As shown in [12,23], hiding data in nanosecond timestamps is feasible. In ext4, four extra 32-bit fields `i_[c|m|a|cr]time_extra` were added to the existing inode structure. The lower 2 bits are used to extend the Unix epoch, the upper 30 bits are used for nanoseconds. Therefore, our implementation only uses the upper 30 bits of the nanosecond timestamps.

Data hidden by this technique is difficult to find. Common file explorers, as well as Autopsy, do not support nanosecond accuracy. Linux commands like `stat [file]` or `debugfs -R 'stat <inode>' [image]` are able to parse nanosecond timestamps, but this does not offer concrete information about hidden data. Furthermore, tests have shown that the Sleuthkit's `istat` command does not take the extra epoch bits into account and therefore timestamps beyond 2038 (Unix Epoch time overflow) are not decoded properly.

5 Use of the Framework

In this section we present how to use the framework and how to integrate new hiding techniques into the existing architecture. If the paper is accepted, theres is also an official module reference including the exact documentation of the most important modules and classes that might be helpful if *fishy* is integrated into other projects.

5.1 Using the CLI Interface

This section will give beginners a first introduction to the *fishy* command structure to better understand how to work with the toolkit. As we have already seen in Table 1, the `CLI` groups all hiding techniques into specific subcommands. Each subcommand provides specific information about how it is used via the `--help` switch. Additionally to the subcommands above, there are the following informational subcommands available:

fattools. Provides relevant meta information about a FAT filesystem, such as sector size, sectors per cluster or the offset to the data region. This command also shows the entries of the file allocation table or current files in a directory.

```
# Get some meta information about the FAT filesystem
$ fishy -d testfs-fat32.dd fattools -i
FAT Type:                                  FAT32
Sector Size:                               512
Sectors per Cluster:                       8
Sectors per FAT:                           3904
FAT Count:                                 2
Dataregion Start Byte:                     4014080
Free Data Clusters (FS Info):              499075
Recently Allocated Data Cluster (FS Info): 8
Root Directory Cluster:                    2
FAT Mirrored:                              False
Active FAT:                                0
Sector of Bootsector Copy:                 6
[...]
```

metadata. Displays the information stored in a metadata file which is created while writing information into the filesystem and are required to restore those information or to wipe hidden content. The first example shows the information that is stored when hiding data in the FAT file slack: (i) the cluster ID with hidden slack space data, (ii) the byte offset to the hidden content starting from cluster ID, and (iii) the length of the data which was written to file slack. The second example shows the information that is stored when hiding data in the MFT entry slack: (i) offset to the MFT slack space, (ii) length of data which was written to file slack, and (iii) sector address of MFT_Mirror when an additional copy of the hidden data is stored in the MFT_Mirror. In addition, the metadata file is password protected in the second case.

```
# Parse a given metadata file
$ fishy metadata -m metadata1.json
Version: 2
Module Identifier: fat-file-slack
Stored Files:
  File_ID: 0
  Filename: 0
  Associated File Metadata:
    {'clusters': [[12, 512, 11]]}

$ fishy -p password metadata -m metadata2.json
Version: 2
Module Identifier: ntfs-mft-slack
Stored Files:
  File_ID: 0
  Filename: secret.txt
  Associated File Metadata:
    {'addrs': [[16792, 11, 5116312]]}
```

The following subsections give some more examples how to use the subcommands of selected hiding techniques in the framework. Since space is limited here, not all techniques can be shown. However, the command structure of all hiding techniques behaves similarly.

fileslack. Hides arbitrary data in the file slack. Provides methods to read (-r), write (-w) and wipe (-c) the file slack of files and directories in ext4, FAT and NTFS filesystems. In addition, we can use the info switch (-i) to check the available slack space of a file in advance.

```
# Write data into slack space of a file (here: testfile.txt)
$ echo "TOP SECRET" | fishy -d testfs-fat32.dd fileslack -d testfile.txt
-m metadata.json -w

# Read hidden data from slack space
$ fishy -d testfs-fat32.dd fileslack -m metadata.json -r
TOP SECRET

# Wipe slack space
$ fishy -d testfs-fat32.dd fileslack -m metadata.json -c

# Show information about slack space of a file (size of testfile.txt: 5 Bytes)
$ fishy -d testfs-fat32.dd fileslack -d testfile.txt -i
File: testfile.txt
   Occupied in last cluster: 5
   Ram Slack:                507
   File Slack:               3584
```

addcluster. Additional cluster allocation where data can be hidden. Provides methods to read, write and wipe additional clusters for a file in FAT and NTFS.

```
# Allocate additional clusters for a file (myfile.txt) and hide data in it
$ echo "TOP SECRET" | fishy -d testfs-fat12.dd addcluster -d myfile.txt
-m metadata.json -w

# Read hidden data from additionally allocated clusters
$ fishy -d testfs-fat12.dd addcluster -m metadata.json -r
TOP SECRET

# Clean up additionally allocated clusters
$ fishy -d testfs-fat12.dd addcluster -m metadata.json -c
```

mftslack. Exploitation of MFT entry slack. Provides methods to read, write and wipe the MFT entries slack space in a NTFS filesystem. The info switch prints further information about the available slack space of each MFT entry (suppressed here because of its length).

```
# Writes the contents of secret.txt (TOP SECRET) into MFT slack space
# and additionally into MFT_Mirror slack space when --domirr is set
$ fishy -d testfs-ntfs.dd mftslack -m metadata.json --domirr -w secret.txt

# Read hidden data from MFT slack space
$ fishy -d testfs-ntfs.dd mftslack -m metadata.json -r
TOP SECRET

# Wipe MFT slack space
$ fishy -d testfs-ntfs.dd mftslack -m metadata.json -c

# Display information about the MFT slack
$ fishy -d testfs-ntfs.dd mftslack -i
```

reserved_gdt_blocks. Exploitation of reserved GDT blocks. Provides methods to read, write and wipe the space reserved for a future file system extension. The info switch summarizes all available bytes of different reserved GDT blocks in all block groups starting from block group 1 since hidden data in block group 0 causes file system check errors. Furthermore, already used space is shown.

```
# Writes the contents of secret.txt (TOP SECRET) into reserved GDT Blocks
$ fishy -d testfs-ext4.dd reserved_gdt_blocks -m metadata.json -w secret.txt

# Read hidden data from reserved GDT Blocks
$ fishy -d testfs-ext4.dd reserved_gdt_blocks -m metadata.json -r
TOP SECRET

# Clean up reserved GDT Blocks
$ fishy -d testfs-ext4.dd reserved_gdt_blocks -m metadata.json -c

# Show relevant information about reserved GDT blocks
$ fishy -d testfs-ext4.dd reserved_gdt_blocks -i
Block size: 1024
Total hiding space in reserved GDT blocks: 1048576 Bytes
Used: 1024 Bytes
```

As we can see, the framework understands the same command switches regardless of which hiding technique or filesystem is currently in use. For all other hiding techniques the same command structure applies, which is why we do not have to introduce all hiding techniques here. Achieving such behavior was one of the goals of the basic design principles of the framework.

5.2 Integration of New Hiding Techniques

This section shows how additional hiding techniques can be integrated into the existing project structure. This can also be understood as a call to the forensic community to add self-developed hiding techniques to the framework. In order to implement a new hiding technique, one can follow the following five steps:

1. Following the repository structure via the folder *'fishy'* to the folder *'wrapper'*, where we first create a new wrapper module for each hiding technique to be added. As already mentioned in Sect. 3, this wrapper module handles the filesystem specific hiding technique calls and fulfills the main-metadata handling. If, for example, another file system has already implemented this type of hiding technique, no new wrapper module needs to be created.
2. Only the CLI module knows about a new wrapper module, not the filesystem specific hiding technique module (bottom-up approach). Therefore, we first need to integrate the new hiding technique wrapper to the CLI module.
3. All currently implemented hiding techniques are located either in the `ext4`, `fat` or `ntfs` submodule. Please notice that further file systems can be added in the future. To add a new hiding technique implementation, we create a new file with an appropriate module name in the respective filesystem subfolder. A simple example would be `fishy/ext4/nanoseconds.py`. The filesystem specific implementation must then be added to the existing wrapper module.

4. A metadata class within the new hiding technique implementation is created. This class holds hiding technique-dependent metadata to correctly recover hidden data after write operations. The write method must return an instance of this class, which then will be written to the metadata file.
5. At least one `write`, `read` and `clear` method is implemented within the new hiding technique. The `info` method is optional. Additional internal helper methods can be implemented as needed. In order to keep the actual hiding technique implementation reusable and simple, we only operate on streams.

6 Evaluation

The practical usability of the framework has already been demonstrated in Sect. 5. Some of the results of our evaluation have already been included in Table 1. It offers an overview about the possible capacity gain of each hiding technique. It also contains a founded rating of its stability. Lastly, we evaluated the detection probability of each technique. The scenario we examined was whether a common filesystem check would detect inconsistencies and point to a modification. Accordingly, we used the standard OS filesystem check utilities to perform the evaluation: `fsck.ext4` for ext4, `fsck.fat` for FAT and `chkdsk` for NTFS.

To test the currently implemented hiding techniques several unit test have already been created. These can be found in the *'tests'* folder of our *fishy* repository. Existing unit tests can be executed by running `pytest`.

Using the `create_testfs.sh` script, it is possible to create prepared filesystem images for all supported filesystem types. These images already include files, which get copied from `utils/fs-files/`. The script has other options, which will be useful when writing unit tests. The created filesystems are intended to be used by unit tests and for developing a new hiding technique. To create a set of test images, we simply run `$ ./create_testfs.sh -t all`. The script is capable of handling branches to generate multiple images with a different file structure. These are especially useful for writing unit tests that expect a certain file structure on the tested filesystem.

6.1 Limitations

fishy still has limitations that are important to mention. First, *fishy* is currently only tested on Linux. Other operating systems may provide different functions to access low level devices.

Although it is possible to hide multiple different files on the filesystem, *fishy* is currently not capable of handling this situation. So, it is up to the user to avoid overwritten data. The CLI is limited to store only one file per call and does not consider other files already written to disk. A simple workaround would be to store multiple files in a zip file before embedding them into the filesystem. However, as a long-term solution, it is better to integrate the functionality of checking whether some data is already hidden and in case to add new content to the previously hidden data.

Currently, *fishy* does not encrypt the data it hides. If the user actually needs encryption, it is up to him to encrypt his data before using the framework. Since tracing unencrypted data with forensic tools is relatively easy, regardless of the hiding technique used (e.g., with file carving), the integration of an additional encyption layer will definitely be considered as a future task. The same applies to data integrity functionality. Since most hiding techniques are not stable, for example if data is hidden in slack spaces where the associated files might change at any time, some data integrity methods would be useful to at least detect whether the hidden data has been damaged in the meantime. Therefore, some redundant information could be added to the original file before the content is actually hidden in the data structure. Depending on the amount of redundant information, data can be recovered, even if some parts are missing.

Manipulating specific data structures in ext4 causes the original metadata checksums to no longer match the new content of the data structures. Therefore, we need to repair the metadata checksums to exactly match the new content, otherwise this could raise suspicions during a forensic investigation. In ext4, the filesystem check tool `e2fsck` can be used to repair inconsistencies, such as wrong metadata checksums. The filesystem check repairs all inconsistencies and another forced filesystem check does not give any further warnings afterwards. However, it would be more convenient to integrate the metadata checksum calculation into the framework to no longer rely on a workaround.

7 Conclusion and Future Work

This paper introduced a new framework called *fishy*, developed in Python. Currently, the implementation contains interfaces for three popular filesystems: ext4, FAT and NFTS. The current state of the framework provides enough functionality to hide data within the supported filesystems and to recover it afterwards. In contrast to previously released tools, we focused on the consistent implementation of different hiding techniques and their long-term reproducibility. The reproducibility and expandability of the framework as well as its modular structure allow the integration of additional hiding techniques and other modern filesystems in the near future (e.g., APFS, Btrfs, ReFS, or XFS).

However, the framework still has a lot of potential for future enhancements and improvements. This section gives a brief overview of some future tasks that complement the work done so far. The filesystem auto detection for FAT and NTFS is currently performed by checking an ASCII string in the boot sector. In order to increase the reliability of *fishy*, it could be reimplemented by using the detection methods that are already realized in regular filesystem implementations. Hidden data can be further obfuscated by filesystem independent approaches like data encryption and steganography. In its current state, *fishy* does not provide on the fly data encryption and has not implemented data integrity methods. Currently, *fishy* produces a metadata file for each hiding operation. Although the metadata file can be encrypted, it is visible through traditional data access methods and gives unwanted hints to hidden data. As a workaround, the metadata file itself could be hidden using an appropriate hiding technique. As a future work, we also consider to automate the evaluation of the hiding approaches against respective filesystem checks as well as the evaluation of forensic suites. Through the integration of *fishy* as an open source library into forensic suites we can analyze whether common filesystem checks recognize hidden data or not. Furthermore, the integration of the CRC32C algorithm into the framework is considered since it is used to calculate metadata checksums within kernel code of ext4 filesystems. Finally, the introduction of multi-data support is another future task. This would greatly enhance the regular use of this toolkit.

Acknowledgments. This work was supported by the German Federal Ministry of Education and Research (BMBF) within the funding program Forschung an Fachhochschulen (contract number: 13FH019IB6) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP (www.crisp-da.de). In addition, we would like to thank all participating students of the bachelor module Project System Development, who played a major role in the implementation of the framework.

References

1. Conlan, K., Baggili, I., Breitinger, F.: Anti-forensics: Furthering digital forensic science through a new extended, granular taxonomy. Digit. Investig. **18**, 66–75 (2016)
2. Rogers, M.: Anti-Forensics, presented at Lockheed Martin, San Diego, 15 September 2005. www.researchgate.net/profile/Marcus_Rogers/publication/268290676_Anti-Forensics_Anti-Forensics/links/575969a908aec91374a3656c.pdf. Accessed 12 May 2018
3. Harris, R.: Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem. Digit. Investig. **3**, 44–49 (2006)
4. Wundram, M., Freiling, F.C., Moch, C.: Anti-forensics: The next step in digital forensics tool testing. IT Security Incident Management and IT Forensics (IMF), pp. 83–97 (2013)
5. Ridder, C.K.: Evidentiary implications of potential security weaknesses in forensic software. Int. J. Digit. Crime Forensics (IJDCF) **1**(3), 80–91 (2009)
6. Newsham, T., Palmer, C., Stamos, A., Burns, J.: Breaking forensics software: weaknesses in critical evidence collection. In: Proceedings of the 2007 Black Hat Conference. Citeseer (2007)
7. Kailus, A.V., Hecht, C., Göbel, T., Liebler, L.: fishy - Ein Framework zur Umsetzung von Versecktechniken in Dateisystemen. D.A.CH Security 2018, syssec Verlag (2018)
8. Anderson, R., Needham, R., Shamir, A.: The steganographic file system. In: Aucsmith, D. (ed.) IH 1998. LNCS, vol. 1525, pp. 73–82. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49380-8_6
9. McDonald, A.D., Kuhn, M.G.: StegFS: a steganographic file system for Linux. In: Pfitzmann, A. (ed.) IH 1999. LNCS, vol. 1768, pp. 463–477. Springer, Heidelberg (2000). https://doi.org/10.1007/10719724_32

10. Piper, S., Davis, M., Shenoi, S.: Countering hostile forensic techniques. In: Olivier, M.S., Shenoi, S. (eds.) Advances in Digital Forensics II. IFIP AICT, vol. 222, pp. 79–90. Springer, Boston, MA (2006). https://doi.org/10.1007/0-387-36891-4_7
11. Göbel, Thomas, Baier, Harald: Anti-forensic capacity and detection rating of hidden data in the Ext4 filesystem. In: Peterson, G., Shenoi, S. (eds.) Advances in Digital Forensics XIV. IFIP AICT, vol. 532, pp. 87–110. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99277-8_6
12. Neuner, S., Voyiatzis, A.G., Schmiedecker, M., Brunthaler, S., Katzenbeisser, S., Weippl, E.R.: Time is on my side: steganography in filesystem metadata. Digit. Investig. **18**, 76–86 (2016)
13. Fairbanks, K.D.: An analysis of Ext4 for digital forensics. Digit. Investig. **9**, 118–130 (2012)
14. Eckstein, K., Jahnke, M.: Data hiding in journaling file systems. In: Proceedings of the 5th Annual Digital Forensic Research Workshop (DFRWS) (2005)
15. Piper, S., Davis, M., Manes, G., Shenoi, S.: Detecting Hidden Data in Ext2/Ext3 File Systems. In: Pollitt, M., Shenoi, S. (eds.) Advances in Digital Forensics. ITIFIP, vol. 194, pp. 245–256. Springer, Boston, MA (2006). https://doi.org/10.1007/0-387-31163-7_20
16. Grugq, T.: The art of defiling: defeating forensic analysis. In: Blackhat Briefings, Las Vegas, NV (2005)
17. Huebner, E., Bem, D., Wee, C.K.: Data hiding in the NTFS file system. Digit. Investig. **3**, 211–226 (2006)
18. Krenhuber, A., Niederschick, A.: Forensic and Anti-Forensic on modern Computer Systems. Johannes Kepler Universitaet, Linz (2007)
19. Berghel, H., Hoelzer, D., Sthultz, M.: Data hiding tactics for windows and unix file systems. In: Advances in Computers, vol. 74, pp. 1–17 (2008)
20. Thompson, I., Monroe, M.: FragFS: an advanced data hiding technique. In: BlackHat Federal, January 2018. http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Thompson/BH-Fed-06-Thompson-up.pdf. Accessed 12 May 2018
21. Forster, J.C., Liu, V.: catch me, if you can... In: BlackHat Briefings (2005). http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-foster-liu-update.pdf. Accessed 12 May 2018
22. Garfinkel, S.: Anti-forensics: techniques, detection and countermeasures. In: 2nd International Conference on i-Warfare and Security, pp. 77–84 (2007)
23. Göbel, T., Baier, H.: Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. Digit. Investig. **24**, 111–120 (2018)
24. Carrier, B.: File System Forensic Analysis. Addison-Wesley Professional, Boston (2005)
25. Wong, D.J.: Ext4 Disk Layout, Ext4 Wiki (2016). https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout. Accessed 12 May 2018