Universiteit Leiden
The Netherlands

Opleiding Informatica

Implementation and Evaluation of a Detection Tool for Data Hiding Techniques in EXT4 File Systems

Rajeck Massa

Supervisors:
Dr. K. F. D. Rietveld & Dr. O. Gadyatskaya

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

28/07/2023

Abstract

Cyber criminals can use computers to store malicious data, for example a virus or some other criminal information. To ensure that not everyone can find this data, they might want to hide this, which is possible in the computer's file system. Several techniques for data hiding on EXT4 file systems have been proposed in prior work. However, all the research currently available is on devising hiding techniques and no tool to reliably detect hidden data using these techniques exists. In this thesis, a hide and detection tool to hide as well as detect hidden data in the EXT4 file system is proposed. Additionally, a benchmark for detection tools is proposed, which facilitates the evaluation of detection tools on a diverse set of file system images and hiding techniques. The benchmark is built in a modular fashion, which means that it can be extended with new detection tools and hiding techniques. Evaluation of the proposed detection tool and a number of existing tools using this benchmark showed that the proposed detection tool successfully detects the implemented data hiding techniques, while the existing tools can not, or detect the data hiding techniques very unreliably. These tools are intended to help computer forensic researchers to speed up their work and make it easier to find any anomalies hidden in a file system, and thus make it easier to catch cyber criminals.

Contents

1 Introduction

In June 2020, Europol shut down a famous encrypted text service, called EncroChat [23]. EncroChat was widely used by criminals to send encrypted messages to each other. They thought their messages and activities were hidden and law enforcement could not decipher them. As of 2023, the shutdown and decryption of EncroChat messages led to 6,500 arrests [22].

Unfortunately, encrypting messages is not the only way cyber criminals could mask their activities and hide any data. One method to hide data is to apply hiding techniques on a file system. This type of data hiding is not as well known as other techniques and could possibly be widely used by cyber criminals. Any type of data could be hidden, such as a coordinates, pictures or harmful files to a device, such as a virus or malware. At this point, a forensic researcher needs to examine the full file system by hand, by using a hex viewer or some other type of tool. This can take a lot of time and is more error-prone, since this has to be done by humans.

To make it easier to detect hidden data, this thesis proposes an automatic detection tool for data hiding techniques in the EXT4 file system. To be able to detect hidden data, the data needs to hidden first. To achieve this, this project also includes a Hide tool. Furthermore, the effectiveness of the detection tool is benchmarked and compared to other tools currently available. To ensure the detection tool is effective, it is necessary that it can be proved that the detection tool can detect the hidden data. Currently, there are no file system images available where data is already hidden nor is there a benchmark available to test such detection tools. To address this issue, a way to easily generate these images and benchmark detection tools is proposed in this thesis, which can be used to prove the effectiveness of such detection tools. Using this benchmark, the detection tool proposed in this thesis is also compared with other computer forensic tools publicly available.

This thesis makes the following contributions to the computer forensics field:

- Implementation of a detection tool for data hiding techniques in the EXT4 file system
- Proposal and implementation of a benchmark to detect hidden data in the EXT4 file system
- Comparison of effectiveness of different tools to detect hidden data in the EXT4 file system
- The detection tool, hide tool and benchmark are released as open source software on GitHub, which can be found here: `https://github.com/RajeckMassa/DHEXT4`

This thesis is organized as follows. In Chapter 2, a concise explanation of the EXT4 file system is given. Furthermore, all the relevant data hiding techniques, the current tools available and their capabilities are discussed. In Chapter 3, the design of the tools is explained. Chapter 4 is centred on the implementation of the code, including a motivation of the choice to program the tools in Python. There is also an explanation included on how the right offsets for each data hiding technique is calculated. In Chapter 5, the benchmark is explained and the results of the benchmark are discussed. Chapter 6 concludes the thesis and gives recommendations for further research.

2 Background

In this Chapter, the background of the EXT4 file system is discussed. The reader is provided with an explanation of the EXT4 file system, which is necessary to understand the discussed hiding techniques. Furthermore, all the data hiding techniques which are considered in the proposed tools are briefly explained. At last, prior art and related work is discussed, where the contributions of this thesis are explained.

2.1 Data hiding in the file system

Data hiding is, as the name suggests, the act of hiding data that someone does not want to be found. For example, cyber criminals can use computers to store malicious data. To make sure that law enforcement cannot find this data, they can decide to hide the data. One widely known method to store data is steganography, which is the "technique of hiding secret data within an ordinary, non-secret, file or message in order to avoid detection" [25]. Steganography can be used to, for example, store text data in a image.

A not as well known hiding technique is hiding data in a file system, such as EXT4. Data hiding in the file system takes advantages of the way how the file system works, for example by exploiting a slack space to fill up a block. However, since it is not as well known, data can slip more easily through an forensic research.

This thesis is focused on the EXT4 file system, and not on the FAT, NTFS, APFS or any other file system. The decision to focus on the EXT4 file system has been made on the fact that Android phones support the use of EXT4 [13], and therefore many devices rely on this file system. The second reason why this thesis is only focused on EXT4 is due to time constraints; it would take too much time for this thesis to focus on more available file systems. However, some of the techniques and methods described in this thesis can be transferred to other file systems.

2.2 Overall structure of EXT4

In this section, the overall structure of the EXT4 file system is discussed. All the information about the EXT4 file system is obtained from the EXT4 wiki. For more details, we refer the reader to [24]. The EXT4 file system consists of blocks, where each block has the same size, the so called block size. A block can store an inode, (a copy of) a group descriptor, (a copy of) a superblock, an inode bitmap, a block bitmap or data.

Superblock
The superblock is the 'main information block' of the file system which contains the most important information about the whole system. Information stored in the superblock is for example the block size, the total number of inodes and the total number of block groups.

Inode
An inode contains information about a file or directory, such as the size, the name and where the

datablocks are stored [24]. All the inodes are stored in the inode table.

Group descriptor
All the blocks in a file system are divided in so called block groups. Each block group has its own inode and block bitmap. If the option sparse_super is enabled, every block group which is a power of 3, 5 or 7 contains a copy of the superblock, a copy of the group descriptors and growth blocks. A group descriptor contains information about the group, such as the number of blocks in the group.

Inode bitmap
An inode bitmap is used to keep track of all the used and free inodes in a block group. An 0 means that the inode is free, and a 1 means that the inode is used. If, for example, the 456th bit in the inode bitmap is a 0, this means that the 456th inode of that block group is free and can be used.

Block bitmap
A block bitmap works the same as the inode bitmap, but instead of keeping track of inodes, a block bitmap keeps track of all the used and free blocks in a block group.

Data blocks
Data blocks are, as the name suggests, blocks that store the data.

Fields
The superblock and group descriptor consist of different fields which store the information. For example, the field `s_inodes_count` in the superblock contains the number of all the inodes in the file system. Some of these fields have a name which ends with `lo` or `hi`, where only the `lo` field is used if the integer stored requires 64 bits.

2.3 Data Hiding Techniques

There are different data hiding techniques for EXT4 identified by T. Göbel and H. Beier in their paper 'Anti-forensic capacity and detection ratings of hidden data in the EXT4 file system' [14]. A data hiding technique is used to describe a location where data in the EXT4 file system can be hidden.

Of the 19 possible data hiding techniques identified, there are 12 considered in this thesis. In the next subchapters, we will discuss the techniques from this paper that will be considered in this thesis. All the information about the data hiding techniques in the next subchapter is obtained from the paper of T. Göbel and H. Beier, in particular the calculations of the number of bytes which can be hidden per data hiding technique. For more details, we refer the reader to [14].

2.3.1 Slack methods

A slack space is a term for allocated space which is not used by the assigned purpose, and cannot be allocated for another purpose. This is due the fact that the file system allocates space in blocks

and there can be some unused room between the end of the needed space (logical size) and the end of the block (physical space). In the EXT4 file system, slack spaces exist at a number of locations as described in [14]. The slack spaces which are discussed in this thesis are concisely explained below.

Superblock slack
Each superblock in the EXT4 file system is 1,024 bytes long. This means that if there is a block size which is larger than 1,024 bytes, there is room to hide data. The only exception is in the first block: the first 1,024 bytes of an EXT4 image are reserved for the partition boot sector, which means that the superblock starts at a offset of 1,024 bytes. The data that can be hidden in the first blockgroup is thus $2,048 - block\_size$ bytes. Each superblock has a copy in an group where the group number is a power of 3, 5 or 7, where the partition boot sector is not included. The amount of data that can be hidden there is $1,024 - blocksize\_bytes$.

Block bitmap
By default, there is no usable space at the end of the block bitmap, since the standard option for EXT4 is that the number of blocks corresponds to the number of bits in a block. However, there is an option to change the number of blocks per group, which can result in $BlockSize - (blocks\_per\_group/8)$ [14] bytes to hide data.

Inode bitmap
There is some usable space at the end of the inode bitmap if the number or inodes is less than the number of bits in one block, which is usually the case. This results in $BlockSize - (inodes\_per\_group/8)$ [14] bytes to hide data, for each inode bitmap per group.

File slack
When a file ends in a block and does not uses all the space in the last allocated block, there exists some space between the end of the file and the end of the block, which can be used to hide data. The size of data which can be hidden differs per image.

2.3.2 inode methods

The following data hiding techniques are performed on inodes.

OSD2
Each inode in the EXT4 file system has an OSD2 field which is 4 bytes [24]. However, this field is obsolete and thus not used, which means that it could be used to hide data [15]. Since each inode has this field, this data hiding technique has $4 * n\_inodes$ bytes of capacity [15].

Extended attributes
Each inode in the EXT4 file system is usually 156 bytes and an EXT4 inode record is 256 bytes long [24]. To expand the inode, it is possible to use the remaining space and use extended attributes. However, when the extended attributes are not used, this space can be used to hide data. This can result in $InodeSize - (128 + i\_extra\_isize)) * inodes$ bytes.

Reserved inodes
The EXT4 file system comes standard with 10 reserved inodes, where inode 8 and 9 are not used for any application [24]. Therefore, these two inodes can be used to hide data. It is also possible to assign more reserved inodes to the file system at initialization, which means that there are more reserved inodes to hide data in. This result in $2 * InodeSize$ bytes.

Reserved space in inodes
Each inode in the EXT4 file system has two bytes available in the `l_i_reserved` field, which is not used. This field can be used in any inode, minus inodes 1-8. Therefore, this results in $2 * (n\_inodes - 8)$ bytes to hide data.

2.3.3 Other methods

The following data hiding techniques does not fit in any particularly group, and are therefore placed under 'other methods'.

Superblock: Backup copies
There are backups of superblocks stored in block groups where the block group number is a power of 3, 5 or 7 [24]. These copies can be used to hide data, with the only risk that the primary superblock cannot be restored using the backup, if the primary superblock is corrupted. This results in the number of block groups which are a power of 3, 5 or 7 $*1024$ bytes to hide data.

Reserved Group Descriptor Table Growth Blocks
In EXT4, reserved group descriptor table growth blocks exists to enable the file system to expand after creation [24]. However, when these growth blocks are not yet used, they can be used to hide data. The first growth block should be left untouched. This results in $s\_reserved\_gdt\_blocks * block\_size * (n\_special\_block\_groups)$ usable bytes, where n_special_block_groups are the block groups which number is a power of 3, 5 or 7.

Partition boot sector
The first 1,024 bytes are used as a partition boot sector in a EXT4 file system [24]. This space can be used to hide data, and thus can hold up to 1,024 bytes of data. It should be noted that it is possible that there is actual data here which is used by an bootloader, but this is a rarity [24].

Block Group Descriptor: Reserved Space
Each block group descriptor in the EXT4 file system has 4 bytes which is used as 'padding', to make the group descriptor 64 bytes long. This padding space can be used to hide data, and therefore results in $4 * BlockGroups$ bytes to hide data.

2.4 Prior Art

There are currently a couple tools that are being used in data forensics. The tools Fishy, Autopsy / The Sleuth Kit, FTK Imager and E2FSCK are widely available. Other tools, such as EnCase [3], are only available for law enforcement and therefore not taken into account.

2.4.1 Fishy

Fishy can be used to hide data using a number of techniques in the EXT4, APFS, FAT and NTFS file systems [15]. For EXT4, the hiding techniques 'Superblock slack', 'reserved GDT blocks', 'file slack', 'osd2' and 'obso_faddr' are implemented [15]. This tool uses a metadata file in the JSON-format to keep track in which location the data is hidden. Using this metadata file, the tool can read the data from the hidden location.
This tool has no option to scan the file system for any suspicious data.

2.4.2 Autopsy / The Sleuth Kit

Autopsy is a program created by BasicTech, focused on digital forensics [17]. This tool is mainly created to find deleted files and to secure data from a hard drive. This tool has one option to detect hidden data, which is searching for a string. However, when a string is encrypted, this function cannot find the encrypted data. Furthermore, the search option does not search in every file system feature [18]. Furthermore, Autopsy implemented a HEX viewer to examine the contents of different files manually. However, there is no option to scan the file system or search for an unknown string and it is not possible to examine different file system features, such as the inode table.
The Sleuth Kit is an CLI program, created by the same creators of Autopsy and also focussed on digital forensics and securing evidence [19]. This tool has some more advance options than Autopsy, such as examing the contents of an inode. However, just like Autopsy, The Sleuth Kit has no option to scan a file system for hidden data or search for an unknown string.

2.4.3 FTK Imager

FTK Imager is a program created by Exterro and also focussed on digital forensics [8]. This tool is also created to find deleted files and to secure data from a hard drive. The main difference between FTK Imager and Autopsy is that FTK Imager gives an user the option to examine various system files, such as the inode table. However, FTK Imager does not provide an option to automatically scan the file system and has no option to search for a string.

2.4.4 E2FSCK

E2FSCK is a tool which is part of E2fsprogs and created by Theodore Ts'o [28]. This tool is used to check the consistency of a file system, where it usually checks the checksum, which is calculated of the metadata from the file system. This checksum is different when there is data hidden and E2FSCK can return an error if it notices this. However, this can easily be fixed by recalculating the checksum, which E2FSCK can do for you.

Furthermore, E2FSCK can report various errors about backup copies. For example, E2FSCK can report errors in the first backup and restore them [14].

2.5 Related work

There are currently several papers based on data hiding. One notable contribution is from Thomas Göbel and Harald Baier, with their paper 'Anti-Forensic Capacity and Detection Rating of Hidden

Data in the Ext4 Filesystem' [14]. This paper is based on finding data hiding techniques in the EXT4 file system, with a rating on the capacity of data they can hold and how easy they are to detect. The same authors created a data hiding framework for multiple file systems, including EXT4, which can be used to hide data in a file system [15]. At last, the same authors published a paper about a data hiding technique called 'Timestomping' [26]. This technique takes advantage of several fields which are reserved for time, but barely used. All the data hiding techniques discussed in this paper are obtained of the research from Thomas Göbel and Harald Baier.

Furthermore, there is a paper from Jeremy Davis, Joe MacLean and David Dampier which analyzes some data hiding techniques for different file systems [16]. Two of these techniques, file slack space and slack space in general, can be used for the EXT4 file system and are discussed in this thesis. The other techniques can be used on other file systems.

There are also some papers based on hiding techniques for the NTFS and APFS file system. One of these papers, written by Thomas Göbel, Harald Baier and Jan Türr [27] has researched several data hiding techniques for the APFS file system. In this paper, they checked if several known data hiding techniques are applicable on the APFS file system. Furthermore, they discussed any newly found data hiding techniques for the APFS file system.

Furthermore, Ewa Huebner, Derek Bem and Cheong Kai Wee have written a paper about data hiding techniques in the NTFS files system [6]. Numerous options to hide data in the NTFS file system were discussed, where the theory of slack methods could be used on EXT4 file systems. Another paper based on NTFS is written by Mamoun Alazab, Sitalakshimi Venkatraman and Paul Watters [20]. This paper concentrates on one particular data hiding technique in NTFS.

The benchmark proposed in this paper could also, with minimal changes, be used to check any detection tools written for the NTFS and APFS file system.

All the papers discussed in this Chapter have in common that they are focused on hiding the data itself, with no option to automatically find and detect any hidden data. Only one tool, Fishy, has the option to read hidden data if the exact location is specified by using a JSON-file. However, if the exact location is unknown, it is impossible to automatically detect that there is data hidden in the file system.

The detection tool proposed in this thesis provides a solution for this problem. Using the detection tool, a computer forensic researcher can automatically find any data hidden in the file system which is hidden using one of the supported data hiding techniques and therefore speeds up the process. Additionally, it makes the work less prone to error. The alternative for now is manually examining a file system using a HEX editor, which can be very confusing and makes it easier to make mistakes.

Furthermore, it was currently very impractical to hide any data using different data hiding techniques in a file system. For the data hiding techniques Superblock Slack, reserved GDT blocks, File Slack, osd2 and obso_faddr, the toolkit Fishy [15] [1] could be used. However, for any other data hiding technique, this process was not automated and one must manually calculate the right offset and hide the data. To address this issue, a hiding tool is included in this thesis, which can hide the

data using one of the implemented data hiding techniques. Using this tool, it is easier and faster to hide data in a file system, and therefore easier to validate the detection tool.

At last, to simplify the process of creating detection tools for new data hiding techniques, a way to easily benchmark the effectiveness of such detection tools is proposed in this thesis. There are currently no test images publicly available, which makes testing a detection tool rather hard. This benchmark includes a way to quickly generate a image with various file system options. This generator can be used to test the detection tool on a diverse set of images. Using this benchmark, one can simply show that their detection tool is working, which minimizes the amount of manual testing and therefore speed up the process. This benchmark is fully automatic.

3 Design

In this Chapter, the design of the hiding tool, detection tool and benchmark is discussed. There were several choices made while designing the various tools, and all these choices are explained in this Chapter.

As stated in Chapter 2, it was necessary to design our own data hiding tool. It was also an option to extend the Fishy framework, as is described on the wiki on the Fishy GitHub page [5]. However, we haven chosen not to do this and create our own tool. Implementing in Fishy also meant that we had to implement additional features of Fishy, such as the JSON files discussed in Chapter 2.4. These features were not needed for our tool, but still take time to implement. This also means that our Hide tool does not keep track where the data is hidden. However, it is possible to add this feature later on. In this Chapter, the other design choices for the Hiding tool are discussed.

By designing the detection tool, the focus laid on automatic detection without manual interference, to solve the problem stated in Chapter 2.4; manually checking the file system takes a lot of time and effort. By creating a automatic tool, a computer forensic researcher does not have to check the whole file system by itself, which saves said time and effort. In this Chapter, the design of the hiding tool is discussed.

At last, the design choices of the benchmark are explained. Currently, there is no benchmark available which can be used to test a data hiding detection tool. To prove the reliability of our detection tool, we are proposing a new benchmark, which can easily be used to create images and run the benchmark on new data hiding detection tools. This benchmark can also be used for new detection tools. In addition, we created a way to simply create images with different file system parameters. This tool is also included in the benchmark.

The hiding and detection tools are both created using their own class, so they can store various information about the image. Each data hiding technique has it's own function: one to hide the data (in the 'Hide'-class) and one to detect data (in the 'Detect'-class).

3.1 Assumptions

For this thesis, all the tools assume that the spare_super flag is enabled in all the images. Furthermore, the tools assume that all the images are 64 bit, since some of the data hiding methods require a 64 bit image.

3.2 Design hiding tool

The data hiding tool is designed as an Command Line Interface (CLI) utility, which can be used from the command line. There are various options implemented in the hiding tools, which can be toggled using a command line argument. The following command line arguments are implemented:

- -f –filename: Filename of the EXT4 image where the data will be hidden. Required.
- -d –data: The data (text string) which needs to be hidden. Required.
- -t –technique: The hiding technique which needs to be used. Required.
- –log/–no-log: –log when logging is required, –no-log to run in 'silenced' mode. Required.
- -i –inode: Specify an inode if the data needs to be hidden in a specific inode. Optional.
- -g –group: Specify an block group if the data needs to be hidden in a specific block group. Optional

It is also possible to use the program in a script, and therefore not use the command line arguments. An script can do this by making an instance of the Hide class.
If the logging function is enabled, the hiding tool will log all the data hiding techniques applied on the image with the right block number to the console.

3.3 Design detection tool

Similar to the hiding tool, the detection tool is a CLI utility. The detection tool has two options: one to report if an particular, specified string is found, and one to report any suspicious data found in the image. This option can be toggled by a command line parameter.

- -f –filename: The name of the EXT4 image which needs to be searched. Required
- -s –string: Can be used to search for a specific string. Optional.
- –log/–no-log: Log the outcome to the console. Required.

When the detection tool finds any suspicious data, it creates an incident. If the 'specific string' option is enabled, the detection tool will only create an incident if the specified string is present in the found data. An incident is a message which includes information about the data found, which is at the end printed at the console if the program is used as an CLI tool. If the program is used in another script, the function `check_all()` returns a Python list of the he names of all the data hiding techniques found.

3.4 Design benchmark

In this subchapter, the design of the benchmark is discussed. With the use of the benchmark it is possible to prove that a detection tool can find all the possible data hiding techniques. We want to use a benchmark which consists of multiple configurations of file systems, to eliminate the chance that a detection tool only works on one type of file system. Furthermore, we want to do this efficiently, so that the benchmark takes as little time as possible

The offsets of multiple factors in the file system, such as the offset of the superblock, can differ in images with different blocksizes. Therefore the benchmark runs the tests with images with different block sizes, to see if this has an effect on the detection tool. Furthermore, by using different image sizes, we also obtain some images where there are too few groups to apply some data hiding techniques which require multiple block groups. Using these, we can test if the hiding tool successfully recognizes this and does not attempt to hide the data in a block group which does not exist. The following list contains all the different parameters used to create the file systems:

- Block size with sizes 1024 KB, 2048 KB and 4096 KB
- Inode size with sizes 256 KB and 512 KB
- Image sizes with approximate 17, 35 and 45 MB.

These configurations lead to ($n\_block\_sizes * n\_inode\_sizes * n\_image\_sizes = 3 * 2 * 3 = 18$) possible images. To extensively test our data hiding techniques, the benchmark is created in a way that every possible combination of data hiding techniques is applied on each possible image configuration. This way, the benchmark ensures that the data hiding techniques do not interfere with each other and that, no matter how many data hiding techniques there are applied on the file system, every one of them can be found.

To test all the data hiding techniques, we want to test them in every possible combination with each other. This results in a total of ($2^{n\_hiding\_techniques} = 2^{12} = 4096$) different combinations of data hiding techniques which are theoretically possible. Since every combination of data hiding technique is applied on the images, the number of images needed for the benchmark results in $4096 * 18 = 73.728$ images.
It should be noted that not every hiding technique is always possible, as explained in Chapter 2.3. Taking the impossible combinations of hiding techniques and file systems in account, there can be a total of 61.422 different, usable images generated.

Of the 61.422 images, 20.474 are the size of approximate 17 MB, 20.474 are the size of approximate 35 MB and 20.474 are the size of approximate 45 MB. This result in a total size of $(20.474 * 17) + (20.474 * 35) + (20.474 * 45) = 348.058 + 716.590 + 921.330 = 1.985.978\ MB$, which is 1985.98 GB. It is ineffective to create all the images at once and store them all together, since this would take up too much space. Since 1985 GB could theoretically be stored on a large hard drive, though impractical, storing all the images at once could be impossible if the benchmark would ever be extended to include extra data hiding techniques.

To solve this issue, the benchmark generates images on demand. To do so, two XML catalogs [2] are used: one with all the possible images and one with all the possible hiding techniques. By using these XML catalogs, a configuration can be made 'on the spot' using a parser. After the creation of the image, an combination of data hiding techniques can be applied on the file system by choosing an entry from the data hiding catalog. This way, only one image is stored at a time, which takes up at a maximum of 45 MB (the largest image size). One downside of this approach is that there are a lot of I/O operations, which could increase the SSD wear if the benchmark is ran on a SSD. Therefore, it is recommended that the benchmark is applied on a RAMDISK.

Using an generator program, all the possible combinations of parameters are created and stored in the catalog, which consists of entries. A parser is used to parse the two catalogs, and respectively create the right image or apply the right combination of data hiding techniques on a file system. An example of an entry of the image catalog is as follows:

```
<image id="1">
    <info>
        <block_size>1024</block_size>
        <inode_size>256</inode_size>
        <size>2048</size>
    </info>
</image>
```

The second catalog is used for the data hiding techniques. Each data hiding technique has the name of the technique, the data which needs to be hidden and the location where the data needs to be hidden. The location is necessary, since each image needs to be exactly the same for the benchmark, so the benchmark can be repeated. An example of an entry in the data hiding catalog is as follows:

```
<image id="40">
    <techniques>
        <technique>
            <name>gd_reserved</name>
            <data>gd</data>
            <inode>-1</inode>
            <group>3</group>
        </technique>
        <technique>
            <name>osd2</name>
            <data>os</data>
            <inode>22</inode>
            <group>-1</group>
        </technique>
    </techniques>
</image>
```

As shown in the example, an inode or group can have the value of '-1'. This means that an inode or group number is not necessary for this hiding technique, and can thus be ignored.

Not every entry in the data hiding catalog is used. When, for example, the data hiding technique 'Superblock slack' is combined with 'osd2' (ID 77) and superblock slack is not possible on the image, this whole entry is skipped and the tool does not check if OSD2 can be found for this entry. Since this would result in an image with only the OSD2 technique performed, this is exactly the same image as ID 10, and therefore unnecessary to check again.
Furthermore, the benchmark tool accepts the following command line arguments:

- –search/no-search - Run the benchmark with respectively to search for the string or search for the unknown
- -i/–imagepath - Path to the image catalog
- -t/–techniquepath - Path to the technique catalog

The benchmark tool included in the public GitHub repository bundles the parser and the detection tool together. The benchmark tool itself loops over all the possible combinations, which will be applied on all the possible images. By using the parser, the image with the right configuration is created, where the right combination of data hiding techniques is applied to. After this process, the detection tool is called, which returns a list of found data hiding techniques. At last, this list is compared to the entry in the data hiding technique, and is determined if the detection tool has found all the possible data hiding techniques. All of this is done automatically when the Benchmark program is ran.

Since the benchmark consists of multiple parts and is modularly constructed, it is possible to use the benchmark for different, new created detection tools. The only part that needs to be changed to work with a new detection tool is the part where the detection tool is called. Furthermore, it is possible to use the benchmark for other file system. To do this, it is required to change a couple of file system specific things, such as the 'Generator' program where an EXT4 file system is created. However, it is possible to use this benchmark with minimal changes for other file systems.

4 Implementation

In this Chapter, the implementation details of the Hide tool, Detection tool and Benchmark are discussed. Furthermore, a explanation is given about how the right location of the data hiding techniques are found and checked.

The programming language Python [10] is used to create all the tools from this thesis. Python is chosen for this project due the fact that it has a large collection of external libraries available, which can be used to work with EXT4 file systems. Furthermore, it is rather straight forward to do I/O operations with Python with the use of the OS library.
The hiding and detection tool uses an open source EXT4 library [4]. This library enables the program to work with EXT4 images and extract various data, such as the offset of inodes.

4.1 Hiding methods

A global explanation on how the correct location to hide and detect the data is calculated is described below. Each inode or group used is specified by the user of the program. If there is no inode or group specified, the program obtains a random inode and set the block group number to 3.

4.1.1 Slack methods

A global overview of the implementation of the slack methods is provided below.

Superblock slack
As stated in Chapter 2,this data hiding technique requires a block size of at least 2048 bytes. If this is not the case, the program raises an Exception and quits. If the block size is 2048 bytes and there is only one block group, there is also no room to hide the data and the program also throws an Exception and quits.
If the file system meets the requirements, the program goes to the first block of the specified group. If there is no specified group, the hiding program will always choose the fourth group (group 3), which contains the second backup.

If there is no data here, the bytes in this space should be equal to 0x00. The detection program checks here if all the bytes are equal to 0x00's, and if not, creates an incident.

Block bitmap
As stated in Chapter 2, it is necessary that the block size in bits is larger than the number of available blocks. If this it not the case, the program throws an error and quits.
If the file system meets these requirements, the program obtains the corresponding block bitmap of the specified group. This is done by reading the `bg_block_bitmap_lo/hi` field of the group descriptor. If there is no group specified, the first group is used. The position in bytes of the start of the slack space is calculated using the following formula: $offset = (block\_of\_bitmap * block\_size) + (blocks\_per\_group/8)$ and the size of the slack space is calculated by $size = starting\_point - block\_end\_bitmap$. The first part is used to calculate the offset of the block bitmap in bytes, and the second part is to skip the occupied room by the blocks. Since one bit is used for each block, it

is necessary to skip $blocks\_per\_group/8$) bytes to go to the start of the slack space.

If there is no data in the slack space, the bytes in this space should be equal to 0xff. The detection program checks if all the bytes in the slack space are equal to 0xff, and if not, creates an incident. It is possible that this space is filled with 0x00's, if the flag 'block_uninit' is enabled. Therefore, the detection tool also checks if the bytes in this space are equal to all 0x00's.

Inode bitmap
The block of the inode bitmap is obtained by reading the `bg_inode_bitmap_lo/hi` field of the specified group descriptor. The offset of the slack space is calculated using the following formula: $offset = ((block\_bitmap * block\_size) + (inodes\_per\_group/8)$. The first part is to calculate the offset of the bitmap in bytes and the second part is to skip to the slack space. Since one bit is used for each inode, it is necessary to skip $inodes\_per\_group/8$ bytes to go to the start of the slack space

If there is no data in the slack space, the space should be filled with all 0xff. The detection program checks if all the bytes in the slack space are equal 0xff, and if not, creates an incident. It is possible that this space is filled with 0x00's, if the flag 'inode_uninit' is enabled. Therefore, the detection tool also checks if the bytes in this space are equal to all 0x00's.

File slack
To calculate the right location of any file slack, the space between the logical size and the physical size is calculated. This is done by calculating how much bytes of the block is used up, and that number is subtracted of the block size. Using this method, it is possible to calculate the offset of the end of the used space and calculate the size of the slack space.

To detect any data in the slack space, it is checked if the bytes in the slack space are not equal to 0x00's, and if there is, the detection tool creates an incident. It should be noted that there could be data from a previous file, which is deleted, but the block is not yet cleared. Therefore, a forensic researcher should still manually check out the data stored in the file slack.

4.1.2 inode methods

All methods discussed in this section operate on inodes in the file system. The external EXT4 library [4] is used to calculate the offsets of the inode.

OSD2
Since the OSD2 field is always at a fixed offset, 0x7E (126 in decimal), the formula to calculate the offset to hide the data is $location\_inode\_byte + 0x7E$.

If there is no data here, the OSD2 field should be filled with all 0x00's. The detection program checks if all the bytes in the OSD2 field are equal to 0x00's, and if not, creates an incident.

Extended attributes
To hide data in the extended attributes, the program skips to the end of the 'standard' inode

without the extended attributes, to the place where the extended attributes start. During testing of the program, it seemed like that the first three bytes of the extended attributes are often used, and these are thus also skipped. So, the formula to calculate the starting point to hide data is $location\_inode\_bytes + 159$

Since there can be actual data in the extended attributes, detection of this data hiding method is a bit more complex. The detection method is designed to obtain the original size of the extended attributes by reading the i_extra_isize field of the inode, and it checks the 'slack space', if there is any data which should not be there. The 'slack space' should be filled with 0x00's, so the detection tool checks if that is the case. If not, the detection tool creates an incident.

Reserved inodes
Since inode 9 and 10 are almost never used [14], they can be used to hide data. The offsets of these inodes are obtained using the EXT4 library.

If there is no data, the inodes should be completely filled with 0x00's - with the exception of the checksum, which is stored at offset 0x7C in the inode. The detection program checks if the remainder of the inodes are filled with 0x00's, and if not, creates an incident.

Reserved space in inodes
Since the `l_i_reserved` field is always at the same offset (0x7A), it is rather straight forward to calculate the location.

The detection tool loops over all the available inodes and checks if the data in the reserved field is filled with 0x00's. If not, the tool creates an incident.

4.1.3 Other methods

The following methods do not fit in a particular group, and are therefore placed in 'Other methods'.

Superblock: Backup copies
The offset to hide the data in the backup copies are calculated using the following formula: $offset = ((blocks\_per\_group * group) + l\_offset) * block\_size$. l_offset can be a 0 or a 1: if the block_size is ≤ 1024, the first block is occupied for the partition boot sector, which means that every block is one 'shifted' to the right.
To detect any hidden data, the backup in the other block groups are compared with the first backup. Since the first backup will always be restored by E2FSCK using a file system check [14], this backup cannot be touched. Therefore, by comparing each backup to the first backup, any hidden data will be found. If any hidden data is found, the detection tool creates an incident.

Reserved Group Descriptor Table Growth Blocks
The growth blocks always start at the block after the group descriptors and the number of growth blocks is stored in the superblock. To calculate the offset of the growth blocks, the following formula

is used: $offset = l\_offset + skip\_group\_descriptors + (group * blocks\_per\_group)$. l_offset can be a 1 or a 2, and is calculated in the same way as the Superblock: Backup Copies. Skip_group_descriptors is the number of blocks occupied to store the group descriptors, and is calculated as follows: $skip\_group\_descriptors = ((n\_group\_descriptirs * 64)/block\_size) + 1$, where the + 1 is used to round up the number.
The detection tool checks if all the data in the growth blocks contains 0x00's, and if not, creates an incident.

Partition boot sector
The partition boot sector is always in the first 1024 bytes, so to hiding and detection data is rather straight forward. The detection tool checks if all the bytes in the PBS are 0x00's, and if not, creates an incident.
It should be noted that it is still possible that the PBS is filled with actual data, so that the detection tool wrongfully reports any data listed here. There is currently no way implemented to check if the data in the PBS stored is actual data used by a bootloader, or hided data.

Block Group Descriptor: Reserved Space
Every block group descriptor in a 64 bit file system has a padding of 4 bytes, which has an offset of 0x3C from the start of the group descriptor. There can be $block\_size/64$ group descriptors stored in each block. For the hiding tool, the data will always be hidden in the first group descriptor from the block. To calculate the offset of the start of the group descriptors, the following formula is used: $offset = (((group * blocks\_per\_group) + l\_offset) * size) + reserved\_offset$. l_offset can be a 1 or a 2, depending on the block_size, by the same condition described by Superblock Copy.

To detect any data in the padding, the detection tool iterates over all the group descriptors and checks if the data is filled with 0x00's. It calculates the group descriptor using the same formula as hiding the data. The detection tool adds 64 bytes for as long as there are group descriptors in the block, to check all the group descriptors stored in that block.

4.2 Implementation of the hiding and detection tool

The hiding and detection tool both uses the OS-library [12], and in particular the `os.pread(fd, offset)` and `os.pwrite(fd, offset, data)` functions, to respectively read and write data to various offsets in the file system. The hide-class needs an specific inode and group number, which can be provided by the command line parameter. If there is no inode or group specified, an random inode is used and the group is set to 3. All the hiding techniques are implemented as a method of the hiding class.
The right offset to hide the data is calculated by the program. A small example of a method of the hide class is explained below:

```
def osd2(self, data: str):
    """
    Responsible for hiding data in the OSD2 field.
    Returns:
        Number of bytes written
```

```
        Location where the data is written to
    """
    offset: Final = 0x7E
    size, data_bytes = self.check_all(2, data)
    return self.write_inodes(self.inode, offset, data_bytes)
```

The `check_all(size, data)` function checks if the data is not larger than the maximum bytes which can be stored in the particular space. If this is the case, the program raises an Exception and quits. The `self.write_inodes(inode, offset, data_bytes)` function writes the data to the offset to that specific inode.
Some hiding techniques require more logic. All the code which is not self-explaining is provided with commentary, which should help understand the code.

The right offset to search for hidden data is calculated using the same way as the hidden data, with the difference that the program now loops over all the existing inodes and/or block groups. A small example of a function to detect hidden data is explained below:

```
def check_osd2(self):
    """
    Checks if there is data in the OSD2-field
    Returns:
        Number of incidents where there is data in the OSD2 field
    """
    count = 0
    osd2_offset: Final = 0x7E
    n_inodes = getattr(self.superblock, "s_inodes_count")
    # Loop through all the inodes
    for n_inode in range(n_inodes):
        # Obtain the inode
        inode = self.volume.get_inode(n_inode)
        offset = inode.offset + osd2_offset
        # Obtain the data and make sure it is all 0's
        data = os.pread(self.fd, 2, offset)
        if data != b"\x00\x00":
            count += 1
            self.handle_found_data(n_inode, data,
                                   "OSD2 is not empty.", "osd2")
    return count
```

`n_inodes` obtains the number of all inodes, where the function loops through. For each inode, the data stored in the OSD2 field is checked and if it is not full of 0x00's, an incident is created.

4.3 Implementation of the benchmark tool

As stated in Chapter 3, the benchmark tool uses an XML Catalog to create the images and apply the right data hiding technique(s) on them. To generate and parse these images, the

`xml.etree.ElementTree` [9] library is used. This library is used due the fact that the creation of XML-trees is fairly simple by using this library. The benchmark is subdivided into two parts.

4.3.1 Generating image catalog

The generator program contains a list of all the image parameters used to generate different images, as discussed in Chapter 3.4. The program uses the `itertools.product()` function to calculate all the possible combinations. For each combination, an entry in the XML catalog is created and added to the tree.

4.3.2 Generating technique catalog

The generator program also contains a list of all the possible data hiding techniques. The data hiding techniques have a fixed string and fixed location assigned to it. This is to assure that every time the exact same image will be created when the catalog is used. If this is not the case, data could be stored in a different place each time the benchmark is run, which means that the results are not reproducible.
To create all the different combinations of the data hiding techniques, the function `itertools.combinations()` [11] is used.

4.4 Parser

The parser is responsible for parsing both the image and technique catalog. To create an image, an ID from the Image Catalog is provided to the parser, which will obtain all parameters connected to this ID. By using `os.system()`, the program `mkfs.ext4` is called to create the image with the desired parameters.

To hide data in an image, an ID from the Technique Catalog and a path to is provided to the parser, which will obtain all the different data hiding techniques, data to be hidden and the location of where the data needs to be hidden. The parser uses the Hide-program to hide the data and stores them in the image which is provided.

4.5 Benchmark execution

After calling the Benchmark script, the tool is fully automatic. The script consists of two loops: the 'inner loop' loops over all the possible images, where the 'outer loop' loops over all the possible combinations of data hiding techniques. On each image, the combination of data hiding techniques are applied using the Hide-class. When these techniques are applied successfully, the instance of the Detection-class is created and the function `check_all()` is called. As explained in Chapter 3, this function returns a list of all the found techniques. This list is compared with the list obtained from the Technique Catalog and any differences are stored, which are printed at the end of the benchmark. This tool can also be found in the public GitHub repository.

To evaluate the effectiveness of the detection tool, the benchmark keeps track of an 'success'. An success is defined if the detection tool can detect all the hiding techniques performed on the image

and there are no 'false positives' reported by the detection tool. The benchmark keeps track of all the missed and false positives, including the data hiding technique which is wrongly detected or missed and the image parameters on where the hiding technique is performed.

It is possible to create a different Hide and Detect class and use this in the benchmark. This way, it is possible to create a new hide and detection tool and test it with the proposed in this thesis. For more information about how to implement a different Hide and Detection class to work with the Benchmark, we refer the reader to the public GitHub repository [21].

4.6 Testing

During testing of the benchmark, multiple bugs in the hiding and detection code were found and fixed. An example is that some of the data hiding methods only worked with an block size of 1024. The results presented in the Chapter 5.3 are of the latest benchmark, with all previous found bugs fixed.

5 Experimental Evaluation

In this Chapter, we conduct a number of experiments to evaluate the effectiveness of the detection tool proposed in this thesis is compared to other forensic tools publicly available. The whole benchmark is also ran at the detection tool, to made sure that the detection tool can catch every combination of data hiding techniques possible. Furthermore, a 'cross-check' is performed with the tool Fishy, which had implemented a small set of data hiding techniques. By checking the detection tool with Fishy, we made sure that the detection tool can detect data hidden by previously proposed methods. At last, a runtime experiment is ran between E2FSCK and the detection tool proposed in this thesis.

5.1 Set up

All the experiments are performed on a WSL2 machine running Ubuntu 20.04.6 LTS. All the images are created on a RAMDISK to reduce SSD wear. Furthermore, Autopsy version 4.20.0, FTK Imager 4.7.1.2 and E2FSCK v1.47.0 [28] are used for these experiments.

5.2 Comparison between tools

Not every tool offers the same tool set, and thus is not every tool available to use in the experiments. A clear overview is shown in Table 1.

Tool / technique	Searching for a string	Searching for the 'unknown'
FTK Imager	✗	✗
The Sleuth Kit	✗	✗
Autopsy	✓	✗
E2FSCK	✗	✓
Tool proposed in this thesis	✓	✓

Table 1: Comparison between options of tools.

Since FTK Imager and The Sleuth Kit both have no option to search for a string or search for the 'unknown', these tools are not included in these experiments. Since Autopsy is a GUI Tool based on The Sleuth Kit, this tool is still mentioned.
It should be noted that there is an extended version of FTK Imager, called FTK Forensic Toolkit. However, since this tool is not publicly available, this tool is not taken into account [7].

Not every data hiding technique is found using the available tools (Autopsy and E2FSCK). Tables 2, 3 and 4 shows a comparison between which data hiding techniques were found by all the different tools. A ✓ is used if a tool can detect the hiding technique on every image that the hiding technique is performed. A ✗ is used if a tool can not detect the hiding technique. A '!' is used if a tool can detect the hiding technique some of the time, and thus is unreliable. For the tool proposed in this thesis, a ✓ or ✗ means that the tool can or can not detect the method using both the 'search for a string' and 'search for the unknown' option.

Technique / tool	Autopsy	E2FSCK	Tool proposed in this thesis
Superblock slack	!	✗	✓
Block bitmap	✗	✓	✓
Inode bitmap	✗	✓	✓
File slack	✓	✗	✓

Table 2: Comparison of detection capability for slack methods.

As can be seen in Table 2, Autopsy can detect the 'File slack' in every file system that the technique was performed on. Furthermore, Autopsy can sometimes detect the Superblock slack hiding technique. The hidden data is not found in every image, and thus Autopsy is for this hiding technique unreliable.
E2FSCK can detect the block and inode bitmap space all of the time.
At last, the tool proposed in this thesis can detect all the slack based hiding techniques.

Technique / tool	Autopsy	E2FSCK	Tool proposed in this thesis
OSD2	✗	✗	✓
Extended attributes	✗	✗	✓
Reserved inodes	✗	✓	✓
Reserved space in inodes	✗	✗	✓

Table 3: Comparison of detection capability for inode methods.

As can be seen in Table 3, Autopsy can not detect any inode based hiding technique.
E2FSCK can only detect the 'Reserved inode' hiding method.
The tool proposed in this thesis can detect all the inode based hiding techniques.

Technique / tool	Autopsy	E2FSCK	Tool proposed in this thesis
Superblock: Backup copies	!	✗	✓
Growth Blocks	!	✗	✓
Partition Boot Sector	✗	✗	✓
Block Group Descriptor: Reserved Space	✗	✗	✓

Table 4: Comparison of detection capability for all the other methods.

As can be seen in Table 4, Autopsy can sometimes detect the 'Superblock: Backup copies' and 'Growth Blocks'. The hidden data is not found in every image, and thus unreliable.
E2FSCK can detect none of the other hiding methods.
The tool proposed in this thesis can detect all of the other hiding methods.

5.3 Results

Since Autopsy and E2FSCK both have no option to automatically test a set of images, every image needed to be loaded individually, which takes a lot of time. Therefore, not all the 61.422

different images described in Chapter 3 are tested with Autopsy and E2FSCK. For these tools, we decided to apply each hiding technique individually on all the possible images, so there are no combinations of data hiding techniques tested with Autopsy and E2FSCK. This resulted in a total of $(11 * 18) + (1 * 12) = 210$ images which needed to be tested by hand. Since the hiding technique 'Superblock: Slack Space' required a block sice of at least 1024 bytes, there were 12 possible images for this hiding technique from the image catalog. For all the other hiding techniques, all the 18 images from the image catalog were used.

5.3.1 Searching for a string

As can be seen in Table 1, Autopsy and the tool proposed in this thesis has the option to search for a string. In Table 5, a comparison between these two tools is given. While performing this experiment, the 'Searching for a string'-option in the tool proposed in this thesis is used.

	Autopsy	Proposed tool	Times applied
Superblock slack	4 (33%)	12 (100%)	12
Block bitmap slack	0 (0%)	18 (100%)	18
Inode bitmap slack	0 (0%)	18 (100%)	18
File slack	18 (100%)	18 (100%)	18
OSD2	0 (0%)	18 (100%)	18
Extended attributes	0 (0%)	18 (100%)	18
Reserved inodes	0 (0%)	18 (100%)	18
Reserved space in inode	0 (0%)	18 (100%)	18
Superblock: backup copies	8 (44%)	18 (100%)	18
Reserved Group Descriptor Table Growth Blocks	12 (66%)	18 (100%)	18
Partition Boot Sector	0 (0%)	18 (100%)	18
Block Group Descriptor: Reserved space	0 (0%)	18 (100%)	18

Table 5: Comparison of detected hiding techniques between Autopsy and the tool proposed in this thesis.

Autopsy is only capable of detecting the File Slack hiding method 100% of the time, and thus only reliable for detecting the File Slack. The tool proposed in this thesis can detect all the hiding techniques all the time.

5.3.2 Searching for the unknown

As can be seen in Table 1, E2FSCK and the tool proposed in this thesis have the option to 'scan' a file system. In Table 6, a comparison between these two tools is given. While performing this experiment, the 'Searching for the unknown'-option in the tool proposed in this thesis is used.

	E2FSCK	Proposed tool	Times applied
Superblock slack	0 (0%)	12 (100%)	12
Block bitmap slack	18 (100%)	18 (100%)	18
Inode bitmap slack	18 (100%)	18 (100%)	18
File slack	0 (0%)	18 (100%)	18
OSD2	0 (0%)	18 (100%)	18
Extended attributes	0 (0%)	18 (100%)	18
Reserved inodes	18 (100%)	18 (100%)	18
Reserved space in inode	0 (0%)	18 (100%)	18
Superblock: backup copies	0 (0%)	18 (100%)	18
Growth Blocks	0 (0%)	18 (100%)	18
Partition Boot Sector	0 (0%)	18 (100%)	18
BGD: Reserved space	0 (0%)	18 (100%)	18

Table 6: Comparison of detected hiding techniques between between E2FSCK and the tool proposed in this thesis.

E2FSCK is only capable of detecting hidden data in the block bitmap slack, inode bitmap slack and in reserved inode. E2FSCK returns a 'Padding not set' error for the block and inode bitmap slack, and a 'Checksum error' for the reserved inode.

It should be noted that it is possible to fix the 'Checksum' errors. E2FSCK gives the option to recalculate the checksum. After this, the error is gone, and the data is still hidden.
Furthermore, the 'Padding not set' error is only available if respectively the inode_uninit and block_uninit flag is not set. If this is the case, the slack space is filled with 0xff's. However, if these flags are set, the slack space is filled with 0x00's, and E2FSCK does not produce errors.

5.3.3 Full benchmark on proposed tool

Since the proposed tool is capable of full automatic testing, it was possible to test all the possible combinations. This experiment is used to check if there is a combination of data hiding techniques and image parameters possible where the detection tool can not find all the techniques. As can be seen in Table 7, there is no combination possible where the detection tool can not find any of the applied data hiding techniques.

Technique	Times found	Times applied
Slack space: Inode bitmap	27648 (100%)	27648
Slack space: Block bitmap	27648 (100%)	27648
Slack space: Superblock	24576 (100%)	24576
Slack space: Files	27648 (100%)	27648
OSD2: Inode	27648 (100%)	27648
Reserved space in inodes	27648 (100%)	27648
Extended attributes in inodes	27648 (100%)	27648
Reserved inodes	27648 (100%)	27648
Partition boot sector	27648 (100%)	27648
Superblock: Backup copies	24576 (100%)	24576
Reserved space: Group descriptor	27648 (100%)	27648
Growth blocks	27648 (100%)	27648

Table 7: Full benchmark of proposed tool.

As can be seen in Table 7, the two superblock options are performed less than the other options. There is one small mistake here: the 'Superblock: Backup Copy' is also not run if the block size is smaller or equal than 1,024 bytes. However, this method is possible with every block size, as long as there are at least 3 block groups. After this bug is fixed, the benchmark is ran again, which produced the results as presented in Table 8:

Technique	Times found	Times applied
Slack space: Inode bitmap	30720 (100%)	30720
Slack space: Block bitmap	30720 (100%)	30720
Slack space: Superblock	24576 (100%)	24576
Slack space: Files	30720 (100%)	30720
OSD2: Inode	30720 (100%)	30720
Reserved space in inodes	30720 (100%)	30720
Extended attributes in inodes	30720 (100%)	30720
Reserved inodes	30720 (100%)	30720
Partition boot sector	30720 (100%)	30720
Superblock: Backup copies	30720 (100%)	30720
Reserved space: Group descriptor	30720 (100%)	30720
Growth blocks	30720 (100%)	30720

Table 8: Full benchmark of proposed tool, where the bug is fixed.

As can be seen in Table 8, the bug regarding to the 'Superblock: Backup Copies' is now fixed. The results are the same with the 'Search for a string' and 'search for the unknown'.

5.3.4 Cross-check with Fishy

Fishy is capable of hiding data using 'Superblock Slack', 'growth blocks', 'File Slack', 'OSD2' and 'obso_faddr', as stated in Chapter 2. Of all these methods, only 'obso_faddr' is not implemented in the detection tool.
For the other four hiding techniques, an experiment is ran, checking if the detection tool can detect the hiding techniques. All the possible combinations are tried. The results are shown in Table 9.

	Detected	Applied
Superblock slack	96 (100%)	96
Growth blocks *	120 (100%)	120
File slack	120 (100%)	120
OSD12	120 (100%)	18

Table 9: Full benchmark of proposed tool.

A note should be placed by growth blocks. As explained in Chapter 4, the Hide and Detection tool skips the first couple of bytes to avoid E2FSCK errors. However, Fishy does not do this. If one tries to hide data which is smaller than the first skipped bytes, and thus stored in the part which could produce errors, the detection tool does not detect this. However, when the string is large enough to skip the first couple of bytes, the detection tool does. In this experiment, a string is chosen which is long enough to skip the first couple of bytes, and thus be detected by the detection tool.
Furthermore, superblock slack is run less than the other three methods. This is due the same fact as explained in Chapter 4; the superblock slack needs a blocksize of minimum 1024 bytes.

5.3.5 Execution time

The last experiment ran for the detection tool is based on run time, where the detection tool proposed in this thesis is compared with the file system check tool E2FSCK. On every possible image in the Image Catalog, each tool is ran 1.000 times on each image. The results between the two tools are shown in Table 10:

	Average time	Total execution time
E2FSCK	0.0018 seconds per image	33 seconds
Proposed tool	0.165 seconds per image	49 minutes, 28 seconds

Table 10: Execution time of the two tools.

As can be seen in Table 10, the detection tool proposed in this tool is a lot slower than E2FSCK. This can be explained by the programming language. E2FSCK is written in C, while the detection tool proposed in this thesis is written in Python. The execution time could probably be sped up if the detection tool is rewritten in C.
Despite the fact that the detection tool is slower than E2FSCK, we think that using this tool is still a lot faster than manually examining a file system.

6 Conclusions and Further Research

In this thesis, a new Hide and Detection tool for data hiding techniques in the EXT4 file system is presented. These two tools can be used to, respectively, hide and automatically detect hidden data in the EXT4 file system. By automatically detecting hidden data, it is easier for computer forensics researcher to find hidden data and thus catch any cyber criminals. Furthermore, the Hide tool can be used to apply any data hiding techniques on a file system. There was currently one tool, Fishy, publicly available. However, this tool only implemented five hiding techniques, which made it very impractical to hide data using any other technique. The Hiding tool addresses this problem

Furthermore, a benchmark is proposed to check the effectiveness of such detection tools. There were currently no test images available online. With the benchmark proposed in this thesis, it is very simple to create a large set of test images and run the detection tools on them. This benchmark can also, with minimal changes, be used to test any new detection tools.

In conclusion, the detection tool from this thesis successfully detects all the hidden data in the file systems, with a given and a unknown string. Autopsy can only detect 4 different hiding techniques with the string provided, but 3 of them not every time. E2FSCK successfully detects three hiding techniques without a string provided, but these messages can easily be bypassed. The experiment with Fishy shows that the detection methods are rightfully implemented. At last, the runtime experiment show that our detection tool is slower than E2FSCK. This may be fixed by implementing the detection tool in C, which could be interesting for further research.

These results mean that it is possible to create a detection tool for data hiding techniques in the EXT4 file system. The benchmark proposed and the experiments run in this thesis show that the detection tool is reliable and successfully detects the hiding techniques, even if they are combined.

For further research, it could be interesting to check the tool on even larger images. Since the intention of this tool is to speed up forensic research, it is important that the tool works relative fast on larger images. Furthermore, the benchmark of this tool is only tested on freshly created images, with only one file stored. It would be interesting to see if the tool performs the same at more filled images. At last, not every possible data hiding technique is yet implemented, since there was no reliable way found to successfully, automatically detect these methods. It would be interesting to see if there can be a way found to successfully, automatically detect these methods.

Furthermore, it should be possible to use the benchmark proposed in this thesis for new detection tools. One could make a detection tool for other file systems, such as NTFS and APFS, and use the benchmark proposed in this thesis to extensively check their detection tool. It would be interesting to see if it is possible to create a detection tool for other file systems and to check the tools effectiveness with the benchmark.

References

[1] Thomas Göbel und Lorenz Liebler Adrian V. Kailus, Christian Hecht. fishy – ein framework zur umsetzung von verstecktechniken in dateisystemen. 2018.

[2] XML Catalogs. Xml catalogs. (Accessed: 27-06-2023). URL: https://xmlcatalogs.org/.

[3] Open Text Corporation. Opentext encase forensic. (Accessed: 28-06-2023). URL: https://www.opentext.com/products/encase-forensic.

[4] Cubinator. Ext4 python library. (Accessed: 22-06-2023). URL: https://github.com/cubinator/ext4.

[5] dasec research group and University of Applied Sciences several bachelor students of the Hochschule Darmstadt (h_da). fishy. (Accessed: 20-07-2023). URL: https://github.com/dasec/fishy.

[6] Cheong Kai Wee Ewa Huebner, Derek Bem. Data hiding in the ntfs file system. 2006.

[7] Exterro. Ftk forensic toolkit. (Accessed: 20-07-2023). URL: https://www.exterro.com/forensic-toolkit.

[8] Exterro. Ftk imager - exterro. (Accessed: 27-06-2023). URL: https://www.exterro.com/ftk-imager.

[9] Python Software Foundation. Pyhthon elementtree library. (Accessed: 22-06-2023). URL: https://docs.python.org/3/library/xml.etree.elementtree.html.

[10] Python Software Foundation. Python - programming language. (Accessed: 22-06-2023). URL: https://www.python.org/.

[11] Python Software Foundation. Python itertools library. (Accessed: 22-06-2023). URL: https://docs.python.org/3/library/itertools.html.

[12] Python Software Foundation. Python os library. (Accessed: 22-06-2023). URL: https://docs.python.org/3/library/os.html.

[13] Google. Android kernel file system support. (Accessed: 13-06-2023). URL: https://source.android.com/docs/core/architecture/android-kernel-file-system-support.

[14] Thomas Göbel and Harald Baier. Anti-forensic capacity and detection rating of hidden data in the ext4 filesystem. *Advances in Digital Forensics XIV*, pages 87–110, 2018.

[15] Thomas Göbel and Harald Baier. fishy - a framework for implementing filesystem-based data hiding techniques. *Proceedings of the 10th EAI International Conference on Digital Forensics & Cyber Crime (ICDF2C)*, 2018.

[16] David Dampier Jeremy Davis, Joe MacLean. Methods of information hiding and detection in file systems. *IEEE*, 2010.

[17] BASISTECH LLC. Autopsy - digital forensics. (Accessed: 27-06-2023). URL: https://www.autopsy.com/.

[18] BASISTECH LLC. Autopsy user documentation: Ad hoc keyword search. (Accessed: 27-06-2023). URL: http://sleuthkit.org/autopsy/docs/user-docs/4.19.3/ad_hoc_keyword_search_page.html.

[19] BASISTECH LLC. The sleuth kit (tsk) & autopsy: Open source digital forensic tools. (Accessed: 27-06-2023). URL: https://sleuthkit.org/.

[20] Paul Watters Mamoun Alazab, Sitalakshimi Venkatraman. Digital forensic techniques for static analysis of ntfs images. 2015.

[21] Rajeck Massa. Dhext4. (Accessed: 19-07-2023). URL: https://github.com/RajeckMassa/DHEXT4.

[22] Reuters. Encrypted phone service 'encrochat' shutdown leads to 6,500 arrests, europol says. (Accessed: 28-06-2023). URL: https://www.reuters.com/world/europe/encrypted-phone-service-encrochat-shutdown-leads-6500-arrests-europol-2023-06-27/.

[23] Reuters Staff. 'gold mine': European police target secret phone network used by crime rings. (Accessed: 28-06-2023). URL: https://www.reuters.com/article/us-netherlands-france-encryption-idINKBN2431YV.

[24] EXT4 team. Ext4 disk layout. (Accessed: 13-06-2023). URL: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[25] TechTarget. What is steganography? (Accessed: 13-06-2023). URL: https://www.techtarget.com/searchsecurity/definition/steganography.

[26] Harald Baier Thomas Göbel. Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. 2018.

[27] Jan Türr Thomas Göbel, Harald Baier. Revisiting data hiding techniques for apple file system. *ACM*, 2019.

[28] Theodore Ts'o. E2fsprogs. (Accessed: 27-06-2023). URL: https://e2fsprogs.sourceforge.net/.