IT Security Infrastructures Lab
Department of Computer Science
Friedrich-Alexander University Erlangen-Nürnberg

Master's Thesis

Timestomping on Disk Images

Niclas Pohl

Erlangen, May 23, 2024

Examiner: Prof. Dr.-Ing. Felix Freiling
Advisor: Lena Voigt, M.Sc.

Eidesstattliche Versicherung / Affidavit

Ich, Niclas Pohl, versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

I, Niclas Pohl, hereby declare formally that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from the sources are marked as such. This thesis was not submitted in the same or a substantially similar version to any other authority to achieve an academic grading.

Nutzungsrecht / Right of Use

Der Friedrich-Alexander-Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für IT-Infrastrukturen, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Ergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

The Friedrich-Alexander University Erlangen-Nürnberg, represented by the IT Security Infrastructures Lab of the Department of Computer Science, is entitled to use the results of this thesis non-exclusively, indefinitely and locally unrestricted for the purpose of research and teaching including any intellectual property right or copyright.

Erlangen, 23. Mai 2024 Niclas Pohl

Zusammenfassung

Das Vorbereiten und Erzeugen von synthetischen Festplattenabbildern für die Lehre in der digitalen Forensik ist eine anstrengende und zeitaufwändige Aufgabe. Oftmals ist es nicht vermeidbar, dass sich beim Erzeugungsprozess Artefakte oder falsche Spuren einschleichen. In diesen Fällen wäre es vorteilhaft diese zu entfernen, bevor die Festplattenabbilder an Studenten ausgehändigt werden. Etwaige Daten, welche dafür modifiziert werden müssen, sind unter anderem Zeitstempel in Metadaten, Logfiles und Datenbanken.

Dieser Prozess der Zeitstempelmanipulation wird geläufig als 'Timestomping' bezeichnet. Timestomping ist eine Technik, welche der Anti-Forensik zugeordnet wird, und umfasst in den meisten Fällen die Manipulation von Metadaten-Zeitstempeln auf gemounteten Festplatten. Das Tool TimeStomp von Metasploit, welches auch der Namensvater dieser Technik ist, ist wohl das bekannteste Timestomping Programm. Tools wie TimeStomp sind durch das Betriebs- und Filesystem sehr restriktiert in den möglichen Änderungen, da bestimmte Zeitstempel einer gemounteten Festplatte nicht veränderbar sind. Desweiteren werden durch Ansätze auf gemounteten Festplatten auch unerwünschte Änderungen an den Daten durchgeführt, zum Beispiel durch das Betriebssystem.

Eine Möglichkeit die Restriktionen von Betriebs- und Filesystemen zu umgehen und gleichzeitig die Menge an unerwünschten Spuren zu minimieren, wäre die Verwendung eines Tools, welches auf ein nicht gemountetes Dateisystem zugreifen und dieses verändern kann.

Das Ziel dieser Masterarbeit ist es, ein solches Tool für das ext4 Dateisystem zu entwickeln. Das Tool DiskForge soll desweiteren Teil eines erweiterbaren Frameworks sein, welches nach Beendigung dieser Arbeit als Ausgangspunkt für weitere Disk Image Manipulation Tools dienen soll. Während dieser Arbeit wurde nicht nur eine Timestomping Funktionalität für Metadaten Zeitstempel geschaffen, sondern wir haben auch diese Funktionalität für bestimmte Datenbanken und Logfiles erweitert. Im Anschluss an unsere Implementierung haben wir unser Tool auch mit mehreren schon bestehenden Ansätzen verglichen.

Abstract

Generating and preparing synthetic disk images for digital forensics education can be tedious. Often, unwanted traces and artifacts are created during the generation process. Modifying the disk images before they are distributed to students is desirable. Possible targets for these modifications include file timestamps, log events, and database entries.

The process of manipulating file timestamps is commonly referred to as timestomping. Timestomping is an anti-forensics technique that manipulates file metadata, usually on mounted file systems. One of the most prominent timestomping tools is TimeStomp by Metasploit, but its functionality is limited to mounted and exploited file systems. A tool that works directly on unmounted disk images would have fewer limitations for our use case, as the operating system would not restrict it, and it could surgically alter single bytes in the disk image. This approach also has the benefit that no bytes get modified by the mounting process. Using this approach, metadata that is not modifiable on mounted disk images, like the ext4 inode Creation timestamp, can be modified.

This thesis aimed to develop DiskForge, an extensible framework for anti-forensic manipulations on unmounted disk images. In this scope, we implemented timestomping functionalities for the ext4 file system. The timestomping functionality is not restricted to inode metadata but can also manipulate various Ubuntu logfiles and browser databases. Additionally, we compare our approach using DiskForge against other existing methods for manipulating inode metadata, log files, and browser databases. We analyzed the intended and unintended modifications to the disk image for each approach. Our evaluation showed that DiskForge can manipulate inode metadata, log files, and browser databases without leaving explicit traces, outperforming most evaluated approaches.

Contents

1

Introduction

1.1. Motivation

In today's world, cybersecurity is becoming increasingly important, and cybersecurity professionals are in high demand. According to the BSI [2], in 2023, there were 68 successful ransomware attacks on companies in germany alone. New jobs are being created in law enforcement and the industrial sector to keep up with the changes in an increasingly digital society.

Training new cybersecurity professionals and students is a time-consuming and resource-intensive process. One of the subfields of cybersecurity this thesis belongs to is digital forensics. Experienced digital forensics professionals must teach the next generation, and lecture materials, including forensic disk images, must be created for the students. In the past, students have examined second-hand hard drives for these lectures. This is no longer possible due to ethical and legal concerns, as these hard drives could contain private or potentially illegal materials. To circumvent this issue, synthetic disk images should be created and seeded with information for the students to find. However, creating these images by hand is a tedious task. Therefore, Scanlon et al. developed a tool for the automated planting of forensic evidence [20]. However, it is possible that during the automated image creation, traces are generated that do not fit or contradict the exercise's narrative. As a result, students are either told to ignore these traces or attempts are made to scrub them. However, very naive approaches like mounting the disk image produce unwanted traces. So, this second approach is usually not used.

It would be best if there were no need to modify disk images. However, creating them without unwanted traces is difficult, so disk image manipulation is necessary. For this procedure, it would be advantageous not to introduce unwanted traces. One approach is to work on an unmounted disk image and access and modify the raw data directly within the image file. This approach

enables complete control of the changes made to the disk image. Unfortunately, to our knowledge, no such tool currently exists for the ext4 file system.

1.2. Task

One alternative to modifying timestamps on mounted forensic disk images is manipulating them directly inside the raw disk image without first mounting it. This approach mitigates traces generated by the mounting process and the operating system, making the manipulation - in theory - more challenging to detect.

This thesis aims to create an extensible manipulation framework for unmounted disk images. As a first step, we aim to implement a new approach for timestomping on the ext4 file system. Our meticulous evaluation of this new tool's capabilities will involve a comprehensive comparison with multiple existing approaches, providing a robust assessment of its effectiveness. The traces generated by each approach will be meticulously analyzed and compared, offering valuable insights into their respective strengths and weaknesses.

Furthermore, this thesis will explore different methods of manipulation detection on ext4 and assess the usability of our tool.

1.3. Related Work

At the time of writing this thesis, no suitable tool is available to manipulate unmounted ext4 disk images for our use case. However, research has been conducted on similar topics.

Joakim Schicht [**21**] created SetMACE, a tool for modifying the timestamps of the *LastWriteTime*, *LastAccessTime*, *CreationTime*, and *ChangeTime* values on the NTFS file system with nanosecond precision. The tool uses a custom driver to bypass the Windows security mechanism that blocks direct write access within volume space. Furthermore, SetMACE can manipulate timestamps on unmounted volumes by internally resolving the file system.

SetMACE and DiskForge, the framework developed during this thesis, have similar approaches to bypassing the system's restrictions by altering file system data directly. The main difference between both approaches is the underlying file system for which the tools are developed. DiskForge was explicitly created to manipulate ext4 disk images, while SetMACE was designed for the NTFS file system. As both file systems work internally in entirely different ways, the implementation of both tools strongly differs.

Freiling et al. (2018) [**10**] experimented to evaluate how difficult it is to create a forged disk image and how well it can be detected. The test group for this experiment consisted of 14 students enrolled in a graduate-level digital forensics course. This experiment aimed to determine what tampering class makes a forgery look like an original and leaves no trace of the forgery. Contrary to common approaches, the evidence should not be overwritten or destroyed. The task for the students was divided into two parts:

1. Set the download time of certain files to an arbitrary point before the actual download occurs.
2. The students were given a forensic disk image and were asked to determine whether it was a fake or an original.

The results of this experiment can be summarized as follows:

- All forgeries analyzed during the experiment were successfully classified. This suggests that it is challenging to create convincing forgeries.
- The effort required to classify an original correctly was less than required to detect a fake. This was surprising since it was assumed that a fake would be detected at the first sign.
- The effort to create a fake was greater than the effort to detect it.
- The less control over the manipulation process, the more complex the manipulation was and the easier it was to detect the forgery.

This experiment shows how difficult and time-consuming it is to produce a good forgery that is difficult or impossible to detect. DiskForge's goal is also to forge disk images so that there are as few traces of manipulation as possible, and the manipulation is challenging to distinguish from an original. We also want to reduce the time it takes to forge an image using DiskForge. DiskForge cannot remove all traces, especially implicit ones, though we hope to optimize the forgery process with our tool.

Schneider et al. (2020) [23] conducted a similar experiment, focusing on main memory images. The memory of the RAM is stored in main memory images so that it can be examined forensically later. In contrast to the previous experiment, Schneider et al. involved graduate-level students and professionals. The scenario for the experiment is as follows: An observation of a website that sells drugs revealed an SSH connection that could be traced to a user. During the house search of the user's home, a main memory image was created, but after this procedure, the power went out, and the hard drives were encrypted. The investigators' objective is to identify traces of the SSH connection on the main memory image. The experiment was divided into three parts:

1. The manipulation task with the students: For this, the students received a main memory image that did not include an SSH connection. They were asked to change the image so that this connection could be found.
2. The individual analysis task with the students: Here, the students were given a randomly selected clean or manipulated image and were asked to distinguish whether it was an original or a fake.
3. A repetitive group analysis task with the professionals: In this phase, teams identified as many images as possible within a specified time frame. The objective was to distinguish between 20 original and 20 manipulated images.

The first experiment quantified the effort required to forge the image. In the second experiment, the quality of the forgeries was evaluated, focusing on their recognizability. The results indicated that 20 participants correctly classified their image, while two incorrectly classified counterfeits as originals. The results of the third experiment demonstrated a high correct detection rate, comparable to the second experiment. In this experiment, 159 out of 183 challenges were correctly classified. Moreover, the professionals' analysis effort was less than the students. This is consistent with the findings of Freiling et al. [10], which demonstrated that it is challenging to create convincing forgeries and that it is easier to recognize forgeries than to create them. Schneider et al.'s work presents an additional intriguing approach to manipulation. In the future, DiskForge could be expanded to encompass both disk image manipulation and main memory image manipulation, thereby encompassing a more comprehensive range of manipulation operations.

Schneider et al. (2022) [22] conducted another study to elucidate the reasons why in studies such as Freiling et al. [10], the creation of manipulations is considerably more challenging than the detection of them. The execution of the study by Schneider et al. encountered several challenges, necessitating the repetition of experiments. In this work, the work by Freiling et al.'s [10] and

Schneider et al.'s [23] previous experiment was subjected to a comprehensive examination, as were the three experiments conducted in this study. For their study, an experiment was conducted with graduate-level students. In this experiment, the students were required to erase the traces of a website access and file download from the disk image without it being detected. Following this, the students were presented with two images and asked to determine whether they were manipulated or unmanipulated disk images. This experimentation was replicated with slightly different parameters, with the test description being modified to increase comprehension. The experiment was repeated a third time, this time with professionals. In this iteration, the teams checked ten images and determined whether they were fakes or originals. Three key insights were derived from the experiments:

1. Accept small numbers: Given the necessity of participants with specialized knowledge to conduct the experiments, a smaller number of participants must be sufficient.
2. Consider the relativity of task definition: When creating the task of manipulation exercises, it is challenging to formulate the description unambiguously, as this may result in the experiment failing.
3. Reduce noise in data collection: Analyzing free text responses can present significant challenges in extracting the requisite data. A questionnaire may offer a more straightforward approach to addressing this issue.

Like the two preceding studies, this study demonstrated the challenges of creating authentic-looking forgeries and conducting forgery experiments. With our work, we hope to ease the creation of credible forgeries. It would be interesting to conduct a similar study in the future, this time using forgeries created with DiskForge.

In their research, **Göbel and Baier (2018) [12]** investigated the possibilities of steganographic data hiding in the ext4 file system. Two important points for the research approach were:

1. An investigator should be unable to tell whether the stored data is injected or real.
2. The injected data should be protected from being overwritten by the file system.

The ext4 inode timestamps, especially the extra fields of the timestamps, were chosen as the carrier for the steganographically stored data. The data was stored in the 30 most significant bits since changing the two least significant bits would change the timestamp date. During execution, the file to hide is split into chunks and written to the timestamps. In contrast to DiskForge, a mounted disk image is used for this, as evident by the source code [1].

Although data hiding is not currently DiskForge's primary purpose, the framework can be extended to serve as a mechanism for hiding data in the ext4 file system on unmounted disk images in the future. This extension could extend Göbel and Baier's research to find more places in the ext4 file system where data can be efficiently hidden.

While limited research exists on the detectability of ext4 timestamp manipulation, some studies have been conducted on the NTFS file system. In the following section, we will present an exemplary approach for detecting timestamp manipulation and discuss its applicability to ext4.

In their research, **Alji and Chougdali (2019) [1]** looked at whether and how machine learning could be used to detect timestamp manipulation on NTFS. They argue that manually searching for manipulations resembles "looking for a needle in a haystack" [1]. They used machine learning to automate this process. First, the data utilized to train the model was extracted from a synthetically generated disk image. The dataset was then split into training and test data. In this case, binary logistic regression was used as the training algorithm. This experiment showed

[1] `https://github.com/dasec/ext4-timestamp-magic/blob/master/timestamp_magic.py` (*Accessed On: 18.05.2024*)

that the trained algorithm could detect newly manipulated files (files it has not been trained on) in the learned context. As Alji and Chougdali note in their conclusion, their research has several limitations. One is that this method was only tested on a single, purely synthetic disk image.

By using DiskForge, one could quickly and efficiently create the larger number of manipulated disk images needed to test and refine Alji and Chougdali's approach on a larger dataset for the ext4 file system. In addition, DiskForge could be used to manipulate non-synthetic disk images to test the approach in a more realistic test environment.

1.4. Results

This thesis developed the *DiskForge* framework, which successfully implemented a timestomping algorithm for unmounted disk images. The framework extended the classical act of timestomping from manipulating timestamps in the inode metadata to also manipulating timestamps in databases and logfiles. During our testing, we were able to prove that this approach manipulates a much smaller number of bytes than other approaches: For metadata manipulations, by a factor of 365 times fewer bytes than an approach using Touch; For logfile manipulations, by a factor of 2 times fewer bytes than an approach using Gedit; Moreover, by a factor of 4153 times, there are fewer bytes for database manipulations than the SQLiteBrowser approach. In this context, manipulating fewer bytes is better as it leaves fewer traces for detection.

However, our framework has certain limitations. The timestomping functionality is only implemented for a few specific log files. Also, block allocation functionality was not implemented due to time constraints. These functionalities may be added in the future. Furthermore, since the framework is extensible, future modules can be added using the implemented utility modules.

Finally, this thesis has shown how difficult it is to create a convincing forgery. This is because most of the framework is designed to manipulate explicit time information and only tries not to violate implicit time information. While analyzing the results, we noticed several implicit traces, which could be used to find discrepancies within the disk image that could indicate the presence of timestomping. One example is the correlation between the creation timestamp of a folder and the modified timestamp of a file contained in it. The file's modified timestamp cannot typically (without manipulation) be earlier than the file's creation timestamp.

In addition to this thesis, we provide the framework's source code[2], a documentation[3] of this framework, and a collection of manipulated disk images. The dataset may be utilized to replicate Schneider et al.'s [23] experiments or train machine-learning approaches similar to those developed by Alji and Chougdali [1]. Additionally, it offers a comprehensive overview of the framework's capabilities.

1.5. Outline

The remainder of this thesis is structured as follows: Chapter 2 provides necessary background information about the ext4 file system, focusing on this thesis's areas of particular interest. Chapter 3 shows the tool's implementation and explains selected mechanisms. In Chapter 4, we

[2] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/tree/main/DiskForge?ref_type=heads` (*Accessed On: 18.05.2024*)

[3] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/tree/main/DiskForge_Documentation?ref_type=heads` (*Accessed On: 18.05.2024*)

meticulously evaluate our approach against many different approaches, ensuring a comprehensive understanding of its effectiveness. Chapter 5 focuses on possible traces left by disk image manipulation approaches, and Chapter 6 focuses on the limitations of our framework's current implementation and highlights areas for future work. Finally, Chapter 7 discusses our tool's viability and summarizes this thesis's results.

1.6. Acknowledgments

The writing of this thesis was guided by Lena Voigt. Many thanks for the helpful feedback and advice as well as the collegial atmosphere! Also, special thanks to Professor Freiling for overseeing this work.

Special thanks go to Hal Pomeranz and Srivathsa Dara for their respective blog series about the ext4 file system.

Special thanks also go to Denis Yablochkin. Without him, I would still curse ext4 inode checksums to this day.

I would also like to thank my fellow students and everyone who helped me with the correction of this thesis.

My deepest gratitude goes to my father and mother. I would never have made it this far without you.

2

Background

This chapter provides some background information before discussing the implementation and evaluation of the DiskForge tool. This section first covers anti-forensics basics and the *Timestomping* technique. Next, an overview of implicit and explicit time information is provided. Finally, a rough overview of the ext4 file system is given.

2.1. Anti-Forensics

To differentiate digital forensics from regular forensics, we would like to examine the definition of digital forensics first. According to the National Institute of Standards and Technology (NIST), digital forensics is "the application of computer science and investigative procedures involving the examination of digital evidence - following proper search authority, chain of custody, validation with mathematics, use of validated tools, repeatability, reporting, and possibly expert testimony" [7]. Thus, digital forensics is the application of forensic investigations in the digital world.

Conversely, techniques employed to impede forensic investigations are designated as 'anti-forensics'. An early definition of digital anti-forensics is given in Phrack magazine[1]. It defines anti-forensics as "the removal, or hiding, of evidence in an attempt to mitigate the effectiveness of a forensics investigation" [17]. Thus, anti-forensics is the art of hindering forensic investigations.

There are several anti-forensic techniques, some of which will be discussed in more detail below:

- **Generic data hiding:** Garfinkel [11] describes the so-called anti-forensic technique 'generic data hiding'. Data is stored in unallocated or otherwise inaccessible locations so that forensic

[1] `http://phrack.org/` (*Accessed On: 18.05.2024*)

tools or investigators overlook it. An example is Metasploit's 'Slacker', which can hide data in FAT and NTFS's Slack space.

- **Artifact Wiping:** is the process of deleting artifacts created by an attacker. In contrast to the conventional approach of merely de-allocating a file upon deletion, various tools[2] are available to remove the file from the hard disk altogether. These tools employ a method of overwriting a file's unallocated storage location several times, rendering the file irrecoverable in the best-case scenario [15].
- **Trail obfuscation:** is a procedure designed to confuse forensic investigators. One method of achieving this is defragmentation, which may result in overwriting existing traces in unallocated memory [15].
- **Attack against computer forensics tools and processes:** Another method of impeding forensic analysis is to target computer forensics tools (CFTs). One such approach is the utilization of so-called zip bombs[3], which are zipped folders containing numerous zipped folders, each of which contains zipped folders, and so on. A particularly well-known example of a zip bomb is '42.zip'[4], a 42 kilobyte directory that expands to a size of 4.5 petabytes when unzipped. Such a deluge of data can overwhelm the forensic tool, leading to its stalling and potential failure. Tools like EnCase, which are designed to unpack zip archives automatically, are particularly susceptible to this attack.

Another well-known anti-forensic technique is described in greater detail below.

Timestomping:
When analyzing a disk image, a forensic investigator might aim to create a timeline to reconstruct the events within a given timeframe. However, the process can be hindered or manipulated through timestomping, resulting in false results that match the adversary's narrative. It is also important to note that timestomping is not solely employed for harmful activities but is also utilized in other contexts. For instance, it can be utilized in research or, as illustrated in our use case, in forensic education to alter traces on a disk image. Timestomping is named after the well-known tool TimeStomp[5] from Metasploit. This technique aims to alter the timestamps of a file or directory. It is identified by MITRE ATT&CK T1070.006 as "a technique that modifies the timestamps of a file (the modification, access, creation, and modification times), often to mimic files that are in the same folder" [14]. By doing so, malicious files can be hidden among harmless files, or forensic investigations can be hindered. The timestamps utilized by the ext4 file system are described in Table 2.1. In this thesis, we will utilize the terms Accessed, Modified,

Timestamp Name	Description
a_time	The last time the file was accessed.
m_time	The last time the file was modified.
c_time	The last time the status of the file changed.
cr_time	The time the file was created.

Table 2.1.: Timestamps used by ext4 and their usage

Changed, and Created when referencing these timestamps. Tools like Touch[6] can easily modify some of the timestamps on Unix-based systems. However, this approach can only modify the

[2] For example, shred `https://linux.die.net/man/1/shred` (*Accessed On: 18.05.2024*)

[3] `https://www.microsoft.com/en-us/windows/learning-center/what-is-a-zip-bomb` (*Accessed On: 18.05.2024*)

[4] `https://github.com/iamtraction/ZOD` (*Accessed On: 18.05.2024*)

[5] `https://www.offsec.com/metasploit-unleashed/timestomp/` (*Accessed On: 18.05.2024*)

[6] `https://man7.org/linux/man-pages/man1/touch.1.html` (*Accessed On: 18.05.2024*)

Accessed and Modified timestamps and set the Changed timestamp to the current system time. The Creation timestamp cannot be modified using this approach. To our knowledge, there is no trivial way to freely modify the Changed and Created timestamps on a mounted ext4 disk image. The image must be modified in an unmounted state to manipulate the Changed and Creation timestamps without restrictions.

2.2. Temporal Data & Time Information

In the context of timestomping, the term 'timestamps' usually refers to the file metadata timestamps. With its innovative approach, DiskForge pushes the boundaries of timestomping beyond file metadata timestamps, encompassing a wide range of log files and browser databases. The term 'temporal data' is crucial in our discussion. It refers to the timestamps in file metadata, logfiles, and databases.

Furthermore, explicit and implicit time information plays a significant role in this work, and the following definition proposed by Dreier et al. [8] is the foundation for our evaluation.

Explicit Time Information: The information that is explicitly stated. In the case of timestamps, the timestamp itself is the explicit information.

Implicit Time Information: The information that can be gathered from the context. For example, the chronological order of events in logfiles or the IDs of the entries in Firefox's places.sqlite database.

The primary objective of DiskForge is to manipulate explicit time information. Additionally, we endeavor to minimize the violations of implicit time information, although this is not the focus of this work.

2.3. Ext4

This section provides a rough overview of selected ext4 functionalities and structures essential for creating the DiskForge framework. Most of the information presented is aggregated from the ext4 wiki page [9], the Oracle blog posts of Srivathsa Dara [4] [5], the SANS blog posts of Hal Pomeranz [19] and the book 'File System Forensic Analysis' by Brian Carrier [3].

The fourth extended file system, short ext4, is a journaling file system widely used among Linux distributions as the default file system. It is the successor to the ext3 file system, and many of its internal structures are extensions of the corresponding ext3 structures.

2.3.1. Layout

Ext4 divides the storage device into logical blocks grouped into block groups. The default block size is typically 4096 bytes. The layout of block group 0 is illustrated in Figure 2.1.

The following structures serve as the foundation for the ext4 file system:

- **ext4 Super Block:** The ext4 file system employs the superblock to store fundamental file system data, including the number of inodes and unallocated blocks. The length of the superblock is 1024 bytes. One distinctive characteristic of the superblock in group 0 is

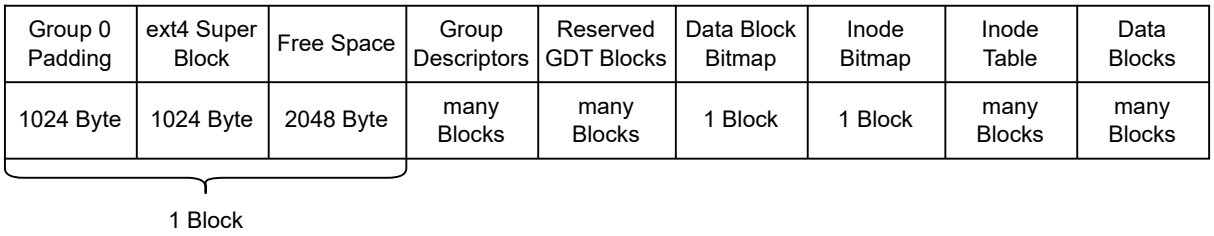

Figure 2.1.: Layout of the block group 0 with a block size of 4096 Bytes. Based on [4] and [9]

its offset of 1024 bytes, in contrast to the superblocks in the other groups. For instance, the 1024 bytes preceding the superblock of group 0 can be utilized to install an x86 boot sector [9] [3].

- **Group Descriptors:** The Group Descriptor Tables comprise a list of Group Descriptor data structures. Each entry in the list corresponds to a block group, containing information such as the start address of the block and inode bitmap [3].
- **Reserved GDT Blocks:** These blocks can be utilized for file system expansion [9].
- **Data Block Bitmap:** The block bitmap records which blocks are utilized and which are not. Each bit in the block bitmap is associated with a specific block within the block group [3].
- **Inode Bitmap:** The inode bitmap is analogous to the block bitmap in that it tracks which inodes are in use and which are not. Analogous to the block bitmap, a single bit is associated with a single inode. [9].
- **Inode Table:** In the ext4 file system, file metadata is stored in a so-called inode. The inodes of a group are then stored in an inode table, which is a list of inode structures [9].
- **Data Blocks:** The data blocks store the file contents saved in the ext4 file system [9].

2.3.2. Flexible Block Groups

In contrast to the structure shown in Figure 2.1, so-called 'Flexible Block Groups' can be utilized in ext4. Flexible Block Groups enable the placement of Data Bitmaps, Inode Bitmaps, and Inode Tables of multiple block groups in contiguous memory locations [9].

Table 2.2 provides an example of using Flexible Block Groups. In this illustration, the blocks within Block Group 0's range are marked in blue, while those within Block Group 1's range are marked in red. Due to the flexible block groups' arrangement, the Data Bitmap, Inode Bitmap, and Inode Table of Block Group 1 are marked in blue. This is because their blocks are within the range of Block Group 0. As a result, the Data Bitmap of group 0 (block 1029) is directly in front of the Data Bitmap of group 1 (block 1030), and the same applies to the Inode Bitmaps and the Inode Tables.

2.3.3. Inodes

An Inode is an ext4 file data structure that stores information about a file or directory. A new feature of ext4 is the ability for inodes to have a length of up to 256 bytes instead of only 126 in ext3. However, it is essential to note that even if the length is 256 bytes, the inode structure utilizes only 156. The remaining 100 bytes serve as a buffer to the next inode. As shown in Figure 2.2, the Accessed, Changed, and Modified attributes are stored at the same offset as in ext2/3. In the extended space of the inode, ext4 introduces a fourth timestamp called Created,

```
Group: 0:                                    Group: 1:
Inode Range: 1 - 8192                        Inode Range: 8193 - 16384
Block Range: 0 - 32767                       Block Range: 32768 - 65535
Layout:                                      Layout:
  Super Block: 0 - 0                           Super Block: 32768 - 32768
  Group Descriptor Table: 1 - 4                Group Descriptor Table: 32769 - 32772
  Group Descriptor Growth Blocks: 5 - 1028     Group Descriptor Growth Blocks: 32773 - 33796
  Data bitmap: 1029 - 1029                     Data bitmap: 1030 - 1030
  Inode bitmap: 1045 - 1045                    Inode bitmap: 1046 - 1046
  Inode Table: 1061 - 1572                     Inode Table: 1573 - 2084
  Data Blocks: 9253 - 32767                    Data Blocks: 33797 - 65535
```

Table 2.2.: Layout of Group 0 (left) and Group 1 (right) as generated with fsstat
Values highlighted in **blue** belong to the block range of group 0.
Values highlighted in **red** belong to the block range of group 1.
The use of flexible block groups allows the blocks of the Data Bitmap (block 1030), Inode Bitmap (block 1046), and the Inode Table (blocks 1573-2084) of Group 1 (right) to be situated within the range of the blocks of Block Group 0 (block range: 0-32767).

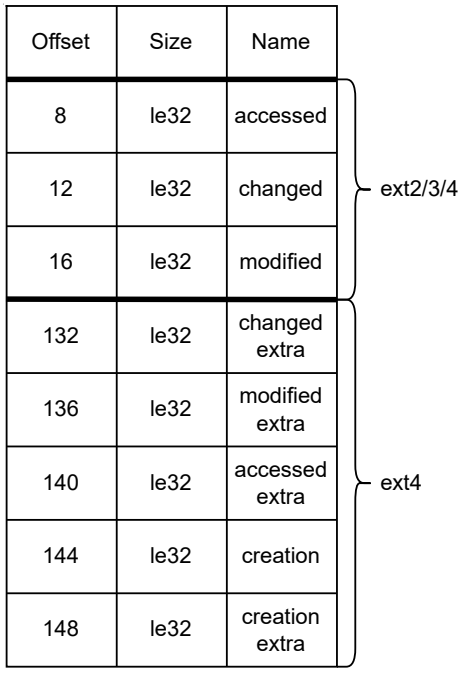

Figure 2.2.: Important values stored in an inode

which indicates when an inode was created. The timestamps store a so-called epoch timestamp, defined as the number of seconds that have elapsed since the start of the epoch, January 1. 1970, at midnight. The epoch timestamp is stored in these values as a 32-bit signed integer.

In addition, the ext4 file system includes an *extra* field for each timestamp. This field stores a 32-bit little-endian value, structured as described in Figure 2.3. The latter shows that the

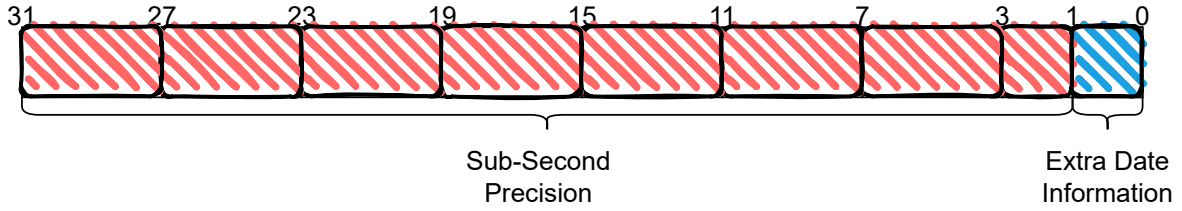

Figure 2.3.: Structure of the extra timestamp fields of the ext4 inode structure
The 30 most significant bits (marked in **red**) are employed to store the microseconds of a timestamp. The two least significant bits (marked in **blue**) are utilized to extend the date stored in the corresponding timestamp.

extra field stores two distinct values. One value adds sub-second precision to the corresponding timestamp, while the other extends the saved date as outlined in Table 2.3. It is important to note that the extra date information occupies the two least significant bits.

Extra Epoch Bits	Most Significant Bit of 32-bit time	Valide Time Range
0 0	1	1901-12-13 to 1969-12-31
0 0	0	1970-01-01 to 2038-01-19
0 1	1	2038-01-19 to 2106-02-07
0 1	0	2106-02-07 to 2174-02-25
1 0	1	2174-02-25 to 2242-03-16
1 0	0	2242-03-16 to 2310-04-04
1 1	1	2310-04-04 to 2378-04-22
1 1	0	2378-04-22 to 2446-05-10

Table 2.3.: Extra Epoch Bits and their corresponding time ranges as outlined in [9]

2.3.4. Extent Tree

Finally, this chapter concludes with an explanation of how ext4 manages data storage locations. Ext3 employs a method of block mapping referred to as *direct* and *indirect*. A file's initial 12 block addresses are stored directly within the inode. If additional blocks are required, an indirect block pointer is used, pointing to a block containing the block addresses. If this is insufficient, double indirect block pointers are employed. Finally, the highest level of indirect block mapping is represented by the triple indirect block pointer [3]. Figure 2.4 illustrates this concept's visual representation at varying block mapping levels. Unlike ext3, which directly maps blocks, ext4 uses a so-called *extent tree*. The tree begins within the inode with a header and a reference to the next lower-level extent block. This block can either be an *inner node*, which refers to further extent blocks, or a *leaf node*, which refers to direct memory blocks. Figure 2.5 provides an example of a sample extent tree. The inode, located on the far left, contains the first extent node,

which comprises a header and extent index within its *i_block*[7] attribute. The extent header is a data structure in all extent blocks and contains parameters such as the depth of the current block. The extent index refers to the next lower-lying extent block, in this case, an inner node. The inner node, in turn, comprises an extent header and several extent indexes, which point to the subsequent lower-lying extent blocks. In this instance, the extent indexes point to leaf nodes. The latter, in contrast, comprise an extent header but do not point to further extent blocks; instead, they point to the data blocks in which the file contents are stored.

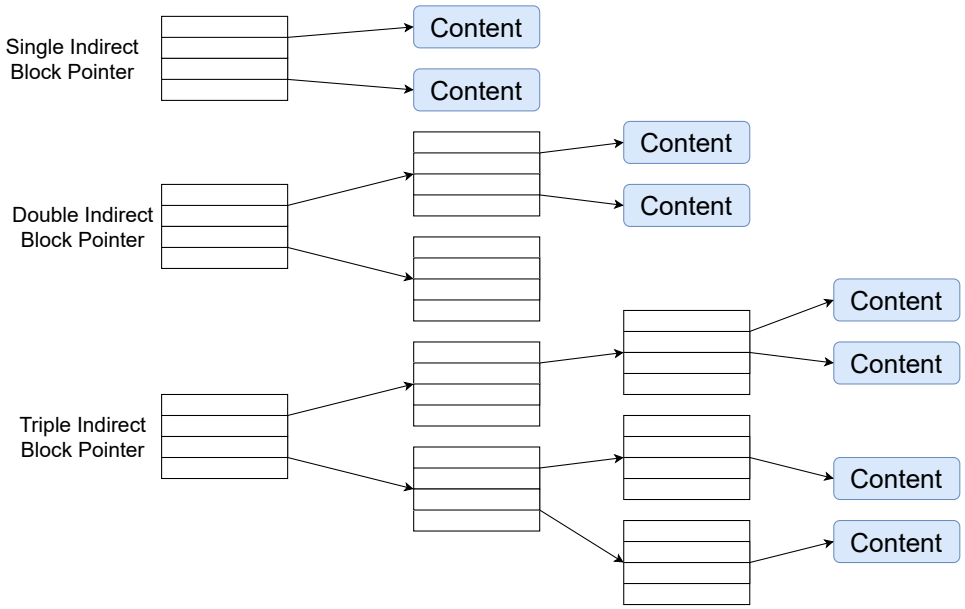

Figure 2.4.: Exemplaric graphical representation of indirect block pointer. Based on [3]

[7]The i_block is part of the inode structure and can be found in the inode struct table at offset 0x28 `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Inode_Table` (*Accessed On: 18.05.2024*)

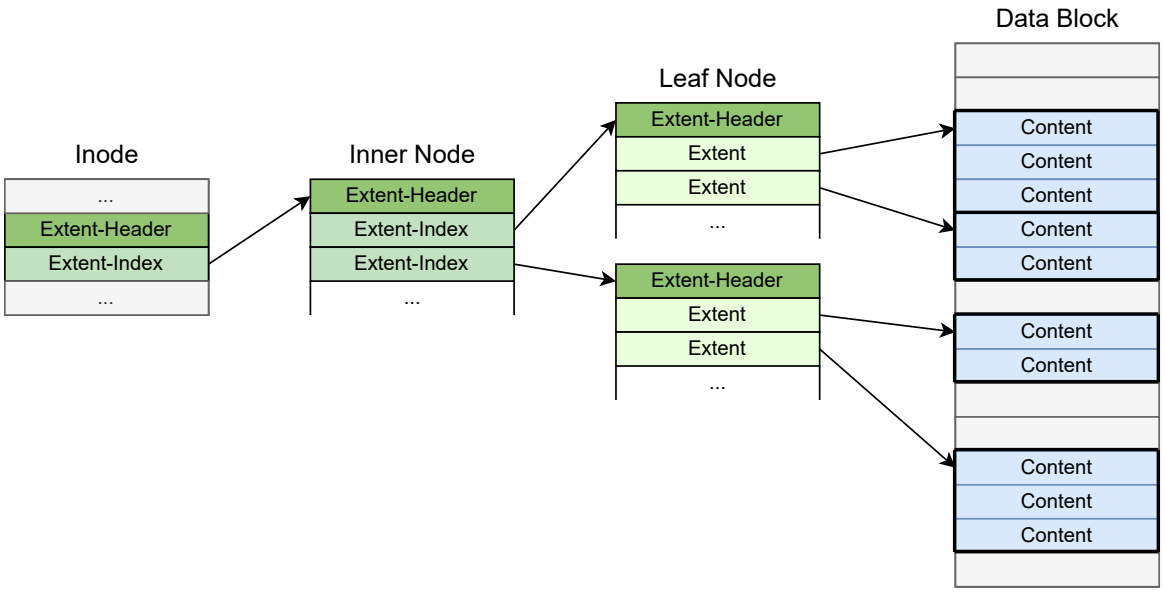

Figure 2.5.: Structure of an extent tree. Based on [13]

3 Implementation

The Implementation chapter provides an overview of the DiskForge framework. Furthermore, selected implementations are discussed in more detail. This includes extracting important data, calculating offsets, writing the data back, and parsing and manipulating logfiles.

3.1. Overview

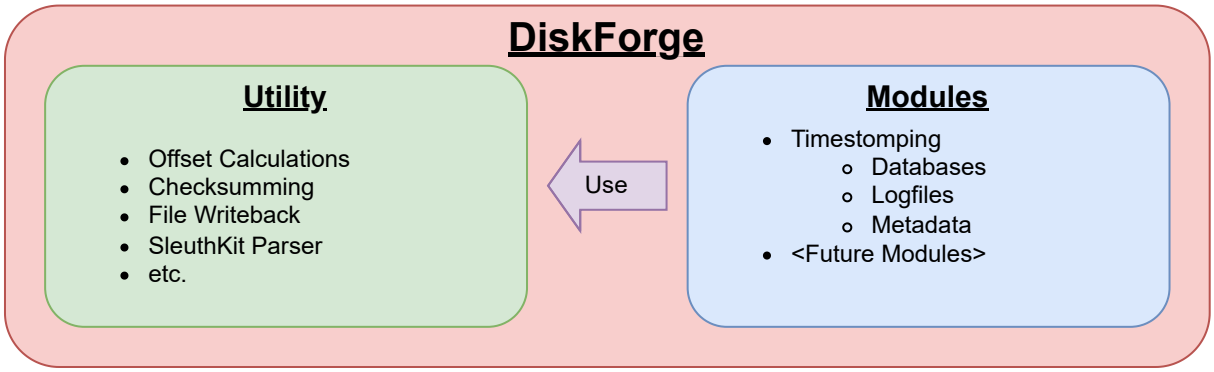

Figure 3.1.: Structure of the DiskForge framework with the Utility functions on the left and the Modules that utilize them on the right

As illustrated in Figure 3.1, the DiskForge framework can be divided into two parts. On the one hand, we have utilities that contain general-purpose functions and algorithms for various disk image manipulation tasks, such as writing data back to a disk image or calculating offsets of specific structures within the disk image. On the other side are the modules, which include

concrete implementations that use the utilities to perform specific manipulation tasks. Currently, a timestomping module is implemented, which provides submodules for manipulating timestamps in databases, log files, and inode metadata. From the beginning of the project, it was determined that the framework should be extensible. To this end, a division into *Utility* and *Modules* was established, which enables the creation of future modules based on the utilities.

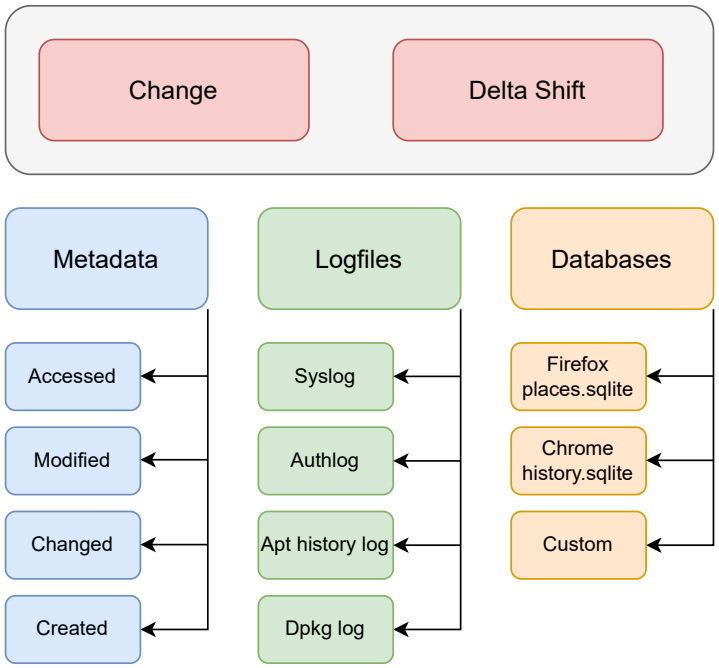

Figure 3.2.: Overview of the implemented Timestomping functionality

As illustrated in Figure 3.2, the implementation of the timestomping module consists of two functionalities for each of the three submodules. The first functionality is *Change*. This function enables the user to set an arbitrary timestamp, which may be displayed and stored. The second functionality, designated as *Delta Shift*, is presented on the opposite side. This functionality enables the user to shift a timestamp by a factor defined by the user, either into the future or the past.

The submodules of *Timestomping* can be divided into two groups. The first group comprises the *Metadata* manipulations, which include the *Metadata* submodule. The four ext4 inode timestamps, namely, Accessed, Modified, Changed, and Created, were selected as targets for the metadata submodule. For each of these timestamps, the main timestamp, which stores the date and time to the second precision, and the extra timestamp, containing the sub-second precision, can be modified. The implementation was tested under the Ubuntu 22.04.4 LTS operating system.

The second group encompasses file content manipulations. The submodules belonging to this group are *Logfiles* and *Databases*. The following logfiles were selected as targets for the log file submodule: *Syslog*, *Auth.log*, the *Apt history.log*, and *Dpkg log*.
The Logfile module can modify the time stamps of log events and restore the log entries to their correct chronological order. Moreover, additional time stamps embedded within the event text can be identified and modified for the Syslog file. This implementation has been tested for the log file formats used by default under Ubuntu 22.04.4 LTS. The database submodule comprises three distinct targets. The first is a functionality that enables modification of all databases that support SQL queries. Secondly, functions have been implemented for Firefox's

places.sqlite and Google Chrome's history.sqlite, which assist users in manipulating them and provide pre-written SQL queries for this purpose. These queries are populated by the user with their parameters. The implementation was tested on Firefox's places.sqlite file (Version 124.0.2) and Google Chrome's history.sqlite file (Version 123.0.6312.86). The remainder of this section is structured as follows. First, selected points from the *Utility* side of the tool are discussed in more detail. Then, the focus shifts to concrete implementations of the *Modules* side, where the *Logfiles* submodule is discussed.

3.2. Filesystem Metadata Extraction

All implemented modules require the metadata of the ext4 file system for a variety of tasks. Among other things, they are used to calculate the different offsets (as seen in Section 3.3) and match file names and inode numbers. In our opinion, one of the easiest ways to extract this metadata is to use The SleuthKit[1] (TSK). TSK is a collection of command line tools forensic investigators use to examine extX[2] file systems. There were two potential approaches to utilize TSK in our framework. Firstly, we could have employed the TSK's Python bindings[3]. However, we ultimately rejected this option due to concerns regarding the quality of the documentation and the possibility of unexpected behavior. Consequently, we opted for the second option, which involves executing and parsing the command-line tools via a Python script. To execute the command line tools, we employed the Python3 *subprocess*[4] library, which enables the execution of command line tools. The generated output is then parsed using regular expressions and stored in specially created objects for subsequent use.

In the implementation, the following commands are utilized:

- **mmls[5,6]:**
  The *mmls* command is used to obtain information about the partition layout of the disk image. Two essential values are the *sector size* and the partition *start* value. These two values are highlighted in color in Listing 3.1, which shows a sample output from *mmls*. The values mentioned above can be used to calculate the total offset of a partition in bytes. Further information is presented in Section 3.3.
- **fsstat[7,8]:**
  The *fsstat* command returns information about the file system of a disk image partition. This information includes the times when the partition was last mounted or modified, the size of its blocks and inodes, and the structure of the individual block groups. A color-coded section of an *fsstat* output can be found in Listing 3.2.

[1] `https://www.sleuthkit.org/sleuthkit/` (*Accessed On: 18.05.2024*)
[2] ext2, ext3, ext4
[3] `https://github.com/py4n6/pytsk` (*Accessed On: 18.05.2024*)
[4] `https://docs.python.org/3/library/subprocess.html` (*Accessed On: 18.05.2024*)
[5] `https://wiki.sleuthkit.org/index.php?title=Mmls` (*Accessed On: 18.05.2024*)
[6] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/SleuthKit/ExtractValuesFromMMLS.py` (*Accessed On: 18.05.2024*)
[7] `http://www.sleuthkit.org/sleuthkit/man/fsstat.html` (*Accessed On: 18.05.2024*)
[8] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/SleuthKit/ExtractValuesFromFSSTAT.py` (*Accessed On: 18.05.2024*)

- **istat**[9,10]**:**
  The *istat* command obtains metadata information about an inode. Listing 3.4 shows the shortened version of an *istat* output. It is possible to extract the four ext4 timestamps, the *size* of the file to which the inode refers, and the *memory blocks*.
- **fls**[11,12]**:**
  The *fls* command maps file names to inode numbers. For this purpose, the output of the *fls* command is split into tuples of *filename* and *inode number* for each line, as shown in Listing 3.3. The resulting inode number is used later in the execution of the framework for various commands or, for example, for offset calculation (as seen in Section 3.3).

The information gathered via TSK is used in the following section to calculate the offsets of blocks and inodes.

Listing 3.1: Excerpt of the *mmls* output of a sample disk image. We extract the **Sector Size** and the **Partition Start**

```
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Safety Table
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  Meta      0000000001   0000000001   0000000001   GPT Header
003:  Meta      0000000002   0000000033   0000000032   Partition Table
004:  000       0000002048   0000004095   0000002048
005:  001       0000004096   0001054719   0001050624   EFI System Partition
006:  002       0001054720   0062912511   0061857792
007:  -------   0062912512   0062914559   0000002048   Unallocated
```

3.3. Offset Calculations

This section describes the offset calculation employed by DiskForge. Initially, the computation of the offsets of individual blocks is explained. This is utilized, for instance, to determine the offsets of the direct blocks of the inodes and the offset of the inode table start block. The information is necessary for the calculation of the inode offset. The computation of the inode offset is presented subsequently.

[9] http://www.sleuthkit.org/sleuthkit/man/istat.html (*Accessed On: 18.05.2024*)
[10] https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/SleuthKit/ExtractValuesFromIstat.py (*Accessed On: 18.05.2024*)
[11] http://www.sleuthkit.org/sleuthkit/man/fls.html (*Accessed On: 18.05.2024*)
[12] https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/SleuthKit/ExtractValuesFromFLS.py (*Accessed On: 18.05.2024*)

Listing 3.2: Excerpt of the *fsstat* output of a sample disk image
We extract: **Inode Size**, **Block Size**, **Block Group Layout**

```
METADATA INFORMATION
------------------------------------------
Inode Size: 256

CONTENT INFORMATION
------------------------------------------
Block Size: 4096

Group: 0:
 Layout:
 Super Block: 0 - 0
 Group Descriptor Table: 1 - 4
 Group Descriptor Growth Blocks: 5 - 1028
 Data bitmap: 1029 - 1029
 Inode bitmap: 1045 - 1045
 Inode Table: 1061 - 1572
 Data Blocks: 9253 - 32767
```

Listing 3.3: Excerpt of the *fls* output of a sample disk image
We extract the **Inode Number** and the **File Name**

```
r/r 1180164: var/log/gpu-manager.log
r/r 1184404: var/log/syslog
r/r 1184407: var/log/kern.log
r/r 1184410: var/log/auth.log
r/r 1180732: var/log/dmesg
```

Listing 3.4: Excerpt of the *istat* output of a sample disk image
We extract: The **Inode Size**, the **Inode Times** and the **Direct Blocks**

```
inode: 1311789
Allocated
Group: 160
Generation Id: 2706489037
uid / gid: 1000 / 1000
mode: rrw-r--r--
Flags: Extents,
size: 5242880
num of links: 1

Inode Times:
Accessed: 2024-04-07 15:56:35.813059702 (CEST)
File Modified: 2024-04-07 16:00:18.435231298 (CEST)
Inode Modified: 2024-04-07 16:00:18.435231298 (CEST)
File Created: 2023-06-10 01:32:18.332726057 (CEST)

Direct Blocks:
2740941 2740942 2740943 2740944 2740945 2740946 2740947 2740948
5162222 5162223 5162224 5162225 5162226 5162227 5162228 5162229
```

3.3.1. Calculating Block Offset[13]

To write files back[14] or manipulate the inode metadata[15], we need to be able to calculate the offset of a single block. The formulas presented below should not be understood as formal concepts but merely reflect the practical approaches we use to implement DiskForge.

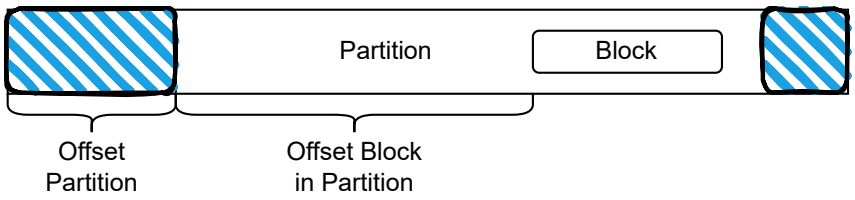

Figure 3.3.: Illustration of a disk image where we want to calculate the block offset

Figure 3.3 shows the different parts we need to calculate to get the full offset of a block inside the disk image in a simplified way. First, we have the *Offset Partition*. We likely do not modify files in partition 000 as seen in Listing ef lst:mmlssample, so we first need to calculate where the partition starts inside our disk image. Next, denoted by *Offset Block in Partition*, we also need to calculate the offset between the start of the partition and the block in question.

1. **Calculate the offset of the partition:**
   The first step is to calculate the offset of the partition where the block is located. In order to achieve this, we utilize the output of mmls, as illustrated in Listing 3.1.
   The provided data can be utilized to compute the following formula:

$$\text{Offset Partition} = \textbf{Begin Partition} \cdot \textbf{Sector Size} \tag{3.1}$$

2. **Calculate Block Offset in Partition:**
   The next step is to calculate the block offset within the partition. To calculate the block offset within the partition, we rely on the fsstat command as shown in Listing 3.2. With this data, we can calculate the *Calculate Block Offset in Partition*:

$$\text{Block Offset in Partition} = \text{Block Number} \cdot \textbf{Block Size} \tag{3.2}$$

3. **Calculate Full Offset:**
   In order to obtain the full offset of the desired block on the disk image, it is necessary to add both values:

$$\text{Full Block Offset} = \text{Offset Partition} + \text{Block Offset in Partition} \tag{3.3}$$

These computations are used in the following section to calculate the offset of an inode table inside the disk image.

[13] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/Calculations/OffSetCalculator.py` (*Accessed On: 18.05.2024*)
[14] as seen in Section 3.4.2: File Writeback
[15] as seen in Section 3.4.1: Metadata Writeback

3.3.2. Calculating Inode Offset[16]

This section outlines the methodology for calculating the offset of an inode and its values within a disk image. As previously stated, the presented formulas are not intended to be regarded as formalized concepts. Instead, they represent the practical approaches employed within the DiskForge framework. The ext4 wiki [9] serves as the foundation for these formulas.

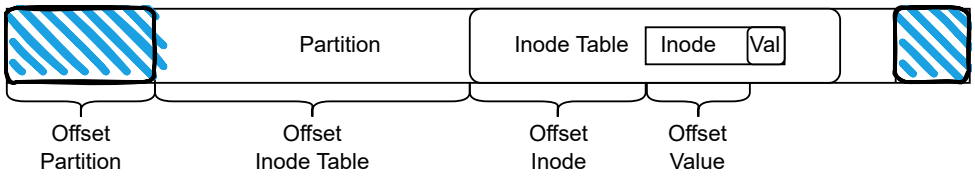

Figure 3.4.: Illustration of a sample disk image where we want to calculate the inode offset

Figure 3.4 illustrates the necessary calculations visualized in a simplified disk image. We first have analogous to Figure 3.3, the *Offset Partition* and, in this case, the *Offset Inode Table*. The *Offset Inode Table* is analogous to the *Offset Block in Partition*, as the inode table starts at the *Inode Table Start* block. Next comes the offsets inside the inode table structure. First, the *Offset Inode* needs to be calculated, which is the offset of the start of the inode structure inside the inode table. Last comes the *Offset Value*, which is the offset of a specific value inside the inode to the beginning of the inode.

This section aims to determine the offset of the inode table start block and that of an inode structure within this table.

1. **Calculate Block Group of Inode:**
   To identify the inode table in which an inode resides, it is first necessary to calculate the block group to which the inode belongs. This is achieved by first extracting the number of inodes per group from fsstat and then applying the following formula:

$$\text{Inode Group} = \left\lfloor \frac{\overbrace{\text{Inode Number}}^{\text{fls}} - 1}{\underbrace{\text{Inodes per Group}}_{\text{fsstat}}} \right\rfloor \tag{3.4}$$

2. **Extract Start of Inode Table:**
   The next step is determining the block at which the block group's inode table begins. As illustrated in Table 2.2, fsstat displays the beginning and end of the inode table for each block group. Identifying the block at which the inode table begins is sufficient for this analysis. The total offset of the *inode table* begin block can be calculated using the formula presented in Section 3.3.1.
3. **Calculate Index of Inode:**
   Calculating the inode's index inside the inode table is necessary to calculate the inode's offset within the inode table. This is achieved by using the following formula:

$$\text{Index Inode} = (\underbrace{\text{Inode Number}}_{\text{fls}} - 1) \mod \underbrace{\text{Inodes per Group}}_{\text{fsstat}} \tag{3.5}$$

[16] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/Calculations/OffSetCalculator.py` (*Accessed On: 18.05.2024*)

4. **Calculate Offset Inode to Inode Table:**
   Once the index of the inode has been obtained, it is possible to calculate the offset of the inode to the beginning of the inode table. This can be achieved by applying the following formula:

$$\text{Offset Inode to Inode Table} = \underbrace{\text{Index Inode}}_{\text{Step 3}} \cdot \underbrace{\text{Inode Size}}_{\text{fsstat}} \tag{3.6}$$

5. **Calculate full Offset of Inode:**
   The values acquired allow the calculation of the full offset of an inode inside a disk image using the following equation:

$$\text{Full Offset} = \underbrace{\text{Full Offset Inode Table}}_{\text{Step 2}} + \underbrace{\text{Offset Inode to Inode Table}}_{\text{Step 4}} \tag{3.7}$$

To access specific values within the inode, it is sufficient to add the offset of the desired value to the total offset of the inode. For this thesis, the most relevant values within the inode are shown in Table 3.1. An example using this formula can be found in the Appendix A.1.

Value	Offset
Accessed	8
Accessed Extra	140
Modified	16
Modified Extra	136
Changed	12
Changed Extra	132
Created	144
Created Extra	148

Table 3.1.: Overview of the ext4 inode values relevant to the thesis. Values were taken from the ext4 wiki [9].

3.4. Write Data to Disk Image

In order to avoid the approach of simply mounting the disk image and copying the files and data, a method was required to write the changed data back into the disk image. Consequently, two distinct workflows were devised, one for metadata and one for whole files.

3.4.1. Metadata Writeback[17]

The first mechanism for data writeback is initiated following the manipulation of metadata timestamps. Figure 3.5 illustrates the general workflow for modifying timestamps.

The metadata manipulation workflow illustrated in Figure 3.5 accepts the image file, the filename, and the new timestamp as input from the user. Initially, the filename is resolved to the corresponding inode number. The inode number can then be used to calculate the inode offset

[17] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/FileModification/timestamp_update.py` (*Accessed On: 18.05.2024*)

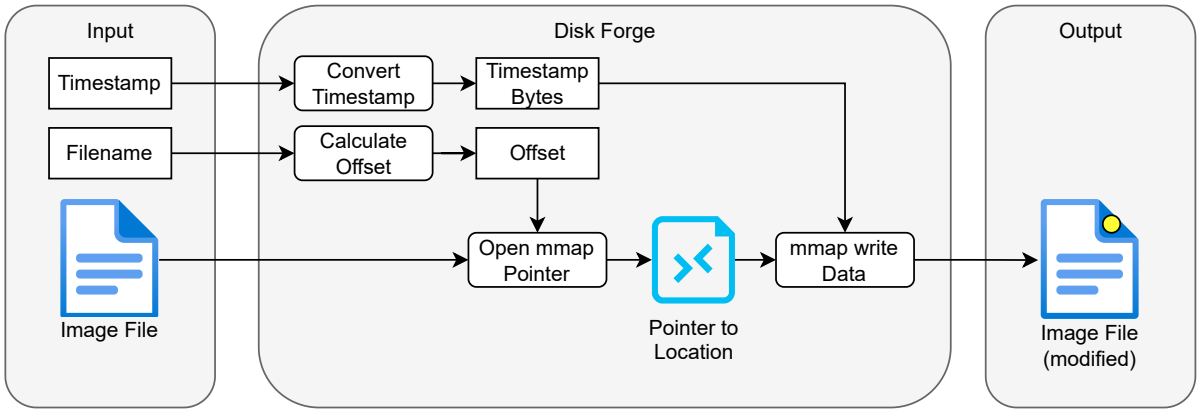

Figure 3.5.: Metadata Manipulation Workflow

using the procedure described in Section 3.3.2. This offset can then be used to create a pointer in the disk image at the location of the inode using *mmap*[18]. mmap is a Python3 library that can load a small, one-system-page-large area from a file into memory. This memory can then be modified, and the changes can be written back into the file. It has the advantage that only a tiny fraction of the entire disk image file must be loaded. Next, the timestamp is converted from a string to bytes in little-endian alignment and written within the area referenced by *mmap*. The output of this workflow is the modified disk image. Listing 3.5 employs pseudo-code to illustrate the methodology for writing back a modified ext4 inode timestamp to the disk image.

Listing 3.5: Pseudocode for the write-back of modified inode metadata timestamps. See Appendix B.1 for a more detailed code representation

```
Function Write Timestamp to Disk Image:
    Pagesize = getconf PAGE_SIZE
    Open DiskImage:
        Offset = Inode Offset aligned to Pagesize
        Index = Index of Inode in Block opened by mmap with Offset
        Pointer = mmap Pointer in the DiskImage at Offset
        Converted Timestamp = Timestamp Converted to Little Endian bytes
        for i < 4 do
            Write Converted Timestamp to Pointer at Index
        end
        Close Pointer
    Close DiskImage
```

Prior to the initialization of the *mmap* pointer, it is necessary to determine the *pagesize* of the system on which the framework is executed. This value is needed, as the offset of *mmap* needs to be aligned to the systems *pagesize*. This is achieved through the use of the getconf[19] command line utility, which returns the page size of the system when executed with the parameter *PAGE_SIZE*. Once the *pagesize* has been determined, the DiskImage is opened by the Python3 open()[20] function, and the offset of the inode and its index within its inode table block are calculated. The *mmap* pointer is then generated with the inode offset. Subsequently, the timestamp is

[18] https://docs.python.org/3/library/mmap.html
[19] `https://linux.die.net/man/1/getconf` (*Accessed On: 18.05.2024*)
[20] `https://docs.python.org/3/library/functions.html#open` (*Accessed On: 18.05.2024*)

converted from a string to little endian bytes. This conversion ensures that the individual bytes are available in the correct order for storage in memory. While *mmap* expects integers as input, in this case, the byte objects can be passed one after the other, as Python3 internally converts them to integers.

3.4.2. File Writeback[21]

This section describes how the modified files are written back to the disk image.

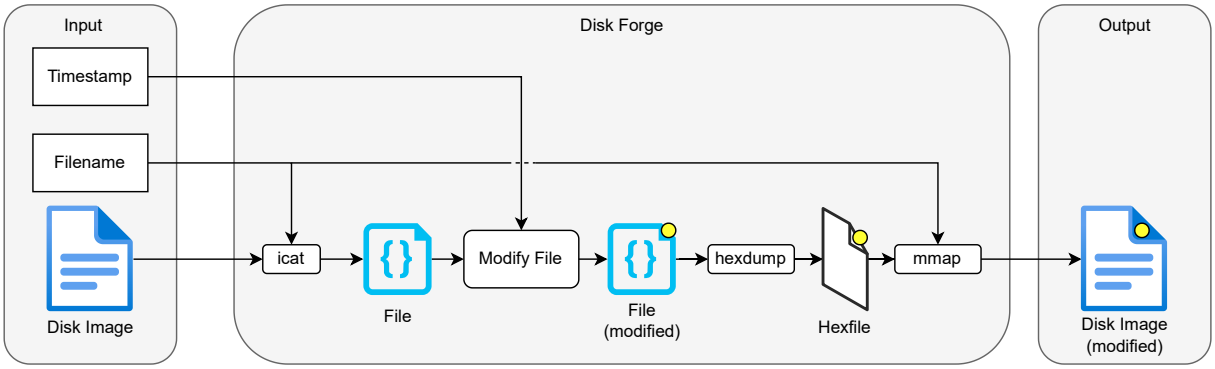

Figure 3.6.: File Manipulation Workflow

Figure 3.6 shows the workflow for modifying and writing back files. First, the file gets extracted using icat, and the extracted file gets modified. To write the file back, the bytes are extracted using hexdump, and these bytes are written back in place of the original file contents on the disk image.

For the sake of clarity, the write-back process is divided into three distinct phases, as outlined below:

1. Prewrite-back check (Listing 3.6)
2. Datawrite-back (Listing 3.7)
3. Postwrite-back check (Listing 3.8)

Listing 3.6 describes the prewrite-back check, which identifies potential sources of problems before the write-back process begins. The file system metadata is extracted with fsstat, which is required later in the write-back process. The inode metadata is extracted using istat, which is required to obtain the file size to which the inode refers. Subsequently, a hexdump of the modified file is generated, from which the hexadecimal bytes are extracted and incorporated into the bytes list. A distinction is made between three possibilities:

1. $Inode\_Size == Byte\_List\_Size$:
   The size stored in the inode matches the length of the bytes list. In this case, there should be no problem.
2. $Inode\_Size > Byte\_List\_Size$:
   The size stored in the inode is greater than the length of the bytes list. In this case, there should be no problem either. It is also possible to update the size value stored in the inode to reflect the length of the bytes list.
3. $Inode\_Size < Byte\_List\_Size$:
   The size stored in the inode is smaller than the length of the bytes list. In this case,

[21] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/FileModification/WriteByteToImageFileData.py` (*Accessed On: 18.05.2024*)

potential problems may arise if not all the data in the bytes list can be stored in the available blocks. At this point, functionality can be added to increase the memory allocated to an inode in the future.

The subsequent step is to initiate the data write-back process.

Listing 3.7 describes the data write-back process. First, the *pagesize* of the system is extracted via *getconf*, as described in Section 3.4.1. Next, the disk image is opened via Python3 open(), and the list with the inode's direct blocks, obtained via istat, is run block by block. The following steps are then carried out for each block separately:

1. Check that the block number is not 0.[22]
2. Calculate the offset of the current block using the formulas from Section 3.3.1.
3. The offset resulting from the second step can then be employed to calculate the page-aligned offset of the mmap pointer. This offset can then be utilized to create the pointer at the specified position.
4. Several bytes corresponding to the block size are written to the block.[23]
5. The mmap pointer is then closed, and the modified data is written back to the disk image.

Upon completion of this process, the number of bytes written back into the blocks is equal to the number of bytes available in the blocks. However, it is possible that not all bytes could be written back[24]. Therefore, a check is conducted to identify the remaining bytes.

Listing 3.8 outlines the performed postwrite-back check. For this purpose, the bytes that were not written back are transferred to a separate list. Subsequently, for each byte in the list, a determination is made as to whether it is 0x00, i.e., a zero byte:

- If the byte is *0x00*, no relevant data is assumed to have been lost.
- Otherwise, it is assumed that data loss has occurred.

A message with the result is returned to the user in both instances. However, it should be noted that data corruption could also occur with *0x00* bytes, but we chose to handle the behavior this way for now. The check is critical if the written-back file is larger than the original, as it determines whether it can be completely written back.

3.5. Inode Checksums[25]

It is possible to leave traces when modifying an inode's timestamps, as the *inode checksum* at offsets *0x74* and *0x82* would not match the stored metadata. According to the ext4 kernel wiki [9], the *inode checksum* is comprised of the UUID, the inode number, the inode generation, and the entirety of the inode, with the checksum fields set to 0. The *UUID* can be found in the *superblock* of block group 0. The bytes of these values are then appended and put through the *crc32c*[26] algorithm.

[22] This is done because we observed that the Firefox places.sqlite file refers to Block 0 as one of its allocated blocks. However, this can not be as Block 0 stores the ext4 superblock.

[23] The size of a block was previously extracted via fsstat in Listing 3.6. The data extracted by fsstat can be found in Section 3.2: Filesystem Metadata Extraction.

[24] As previously stated, our analyses revealed that the places.sqlite file specifies a larger inode size than the allocated blocks can accommodate

[25] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/Checksumming/generate_crc32c.py` (*Accessed On: 18.05.2024*)

[26] `https://en.wikipedia.org/wiki/Cyclic_redundancy_check` (*Accessed On: 18.05.2024*)

Listing 3.6: Pseudocode describing the prewrite-back check for the filewrite-back.
See Appendix B.2 for a more detailed code representation.

```
Function Write Modified File to Disk Image (Pre-Writeback Check):
    FS_Meta = Extract File System Metadata using fsstat
    Inode_Data = Extract Inode Data with istat
    Inode_Size = Size stored in Inode_Data
    Bytes = Extract Hexadecimal Bytes from Hexdump of Modified File
    if Inode_Size == length(Bytes) then
        No Errors
    else if Inode_Size > length(Bytes) then
        No Errors
        Potentially update Inode_Size
    else
        Potential Problem if we can not save all the Data
    end
    ModifyBinary()
```

Listing 3.7: Pseudocode describing thewrite-back procedure for a file.
See Appendix B.3 for a more detailed code representation.

```
Function Modify Binary (Write-back):
    Pagesize = getconf PAGE_SIZE
    Open DiskImage:
        foreach Block in Blocks do
            1. Check that Block does not refer to Number 0
            2. Calculate Offset of Block
            3. Open mmap Pointer at Offset
            4. Write Bytes to Pointer
            5. Close mmap Pointer
        end
    Close DiskImage
```

Listing 3.8: Pseudocode desribing the postwrite-back check for the filewrite-back.
See Appendix B.4 for a more detailed code representation.

```
Function Modify Binary (Data Loss Check):
    LeftOverBytes = Bytes not written Back
    foreach Byte in LeftOverBytes do
        if Byte == 0 then
            Everything Okay
        end
        else
            Data Loss
        end
    end
```

Our observations and those of Denis Yablochkin [25] indicate that this is only partially true. Ext4 employs the *crc32c* algorithm with the values above, but it does not directly store the output. Instead, it subtracts the output of the *crc32c* algorithm from the value *0xFFFFFFFF*.

In order to generate the checksum for our tool, we implemented a checksum generator. Listing B.9 illustrates the pseudocode for this procedure.

Listing 3.9: Pseudocode describing the checksumming process. See Appendix B, Listing B.9 for a more detailed code representation

Function *Inode Checksumming*:
    Inode_Data = Read Inode and Set Checksum fields to 0
    Generation_Data = Extract Generation Number of Inode
    UUID = Extract UUID from Superblock
    Inode_Number = Convert Inode Number to Little Endian Bytes
    Serialized = Append(**UUID** + **Inode_Number** + **Generation_Data** + **Inode_Data**)
    CRC32C_Checksum = Execute crc32c(**Serialized**)
    Modded_CRC32C = 0xFFFFFFFF - **CRC32C_Checksum**

First, the complete inode gets extracted, and the checksum fields are set to *0x00*. Next, the *generation number* gets extracted from within the inode. After this step, the *UUID* gets extracted from the ext4 *superblock*. All these values are appended with the inode number and put through the *crc32c* algorithm. At last, differing from the procedure outlined in the ext4 documentation, the value generated by the *crc32c* algorithm gets subtracted from *0xFFFFFFFF*. The result of the subtraction is then written back into the inode.

3.6. Logfiles

This section deals with manipulating log files as an example for the timestomping modules. First, we want to explain why we needed to differentiate the log file types used in the execution of the log file module and why we have created individual classes for the individual log file types in the utilities. Moving on, we'll provide a detailed overview of the Syslog class's structure and methods. This will serve as a practical example and help to understand its functionality better. We'll also delve into a unique feature of Syslog event processing: the generation of years.

Listing 3.10: Collection of different log file entries

```
syslog: Mar 10 14:27:47 Niclas-Rechner systemd[1] ...

dpkg: 2024-03-01 12:31:51 startup archives unpack

apt-history:
Start-Date: 2024-03-01 12:31:51
Commandline: /usr/bin/unattended-upgrade
Upgrade: libtiff5:amd64 (4.3.0-6ubuntu0.7, 4.3.0-6ubuntu0.8)
End-Date: 2024-03-01 12:31:52
```

As shown in Listing 3.10, the structure of logfile events varies considerably. Our initial approach involved parsing the logfiles using regular expressions, which proved highly susceptible to error. Consequently, we opted for an alternative approach, which we will now describe in detail.

A class file was created for each log file type to handle the entries' parsing, manipulation, and sorting. The structure of the Syslog class will be exemplified here.
It has to be noted that this implementation is for the standard layout[27] of the logfiles. Should the logfile's structure diverge from the standard structure implemented here, the source code may need to be extended.

3.6.1. Syslog[28]

Each logfile class file contains a class that represents the structure and operations of an entry. Furthermore, functions for parsing the log, filtering the events, and writing back the file are included.

The class file structure is generally similar for all log types:

- Header: Values stored by the Event Object
- ___init___(): Initalizes the object
- ___lt___(): Brings less than functionality
- change(): Functions for changing the timestamp
- shift(): Function for shifting the timestamp
- writeback(): Function for converting the object to a string

The points above are elucidated in greater detail below.

Values stored by the Event Object:
Each of the class objects stores at least the *timestamp* and a so-called *message*, which reflects the remainder of the event without the *timestamp* and the timestamp converted to unixtime in a variable called *timestamp_unix*. Some event objects also store other values. In the case of Syslog, an extra variable stores the *year*, and a boolean variable stores whether the keyword *audit()* appears in the message.

___init___():
The init() function is employed to instantiate the event object. This process is illustrated in Listing 3.11. A line from the log file[29] is passed to the function, which then employs the Python3 split()[30] function to split the string based on specific delimiters, in the case of Syslog, spaces. The resulting list of strings is then utilized to populate the *timestamp* variable, which consists of the first three values in the list, as can be observed in Listing 3.10. The remaining data is stored in the variable *message*. Then, the *message* variable is searched for the string *audit()*; if this is found, the variable *audit* is set to *True*. It should be noted that the variables *year* and *timestamp_unix* are not set at this point. This is done at a later stage, which is shown in Section 3.6.2.

[27] The log file layouts, which are used by Ubuntu 22.04.4 LTS. It is possible to set custom log file layouts, but these will not currently work with DiskForge.

[28] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/LogEntryStructures/sys_log.py` (*Accessed On: 18.05.2024*)

[29] **Note:** Some other log files such as the *apt/history.log* can contain multi-line events. In such cases, more than one line is submitted to init()

[30] `https://docs.python.org/3/library/stdtypes.html#str.split` (*Accessed On: 18.05.2024*)

___lt___():
The lt() (less than) function is employed in order to facilitate the use of the < operator. The internal distinction between the larger and the smaller object is illustrated in Listing 3.12. This operator is of significant importance, as it enables the Python3 sort()[31] function to be utilized on a list of event objects. This allows for the chronological sorting of the events.

change() and shift():
The following paragraph will describe the change() procedure, followed by shift().

In the change() method, the new timestamp is initially divided into its date and 24-hour time components. Next, the date is divided into year, month, and day. Subsequently, the previously saved timestamp is replaced by the month, day, and time (without the year, as the Syslog file does not save the year). Next, the previous unix-timestamp is temporarily stored, and the unix-timestamp of the new timestamp is calculated and then stored in the unix-timestamp variable. Finally, the timestamp message is searched for the old Unix time; if it is found, it is replaced by the new one. An example of such an extra timestamp in the message can be found in Listing 3.13.
The shift() process is distinct from that of change(). Once more, the previous unix-timestamp is stored temporarily. Subsequently, the previous value of the unix-timestamp is added to the shift factor and stored in the unix-timestamp variable. A new timestamp is then generated from the new value in the unix-timestamp variable using the Python3 datetime[32] library and stored in the timestamp field. Finally, as with the change() function, the *Message* is searched for the previous value of the Unix-timestamp variable and replaced with the new value if it is found.

writeback():
The writeback() function accepts the parameters of the event object and reassembles them into a string. In the case of Syslog events, the timestamp and message are concatenated and separated by a space.

Listing 3.11: Pseudocode outlining the initialization of an event object.
The full code can be found in Appendix B, Listing B.5

Function *___init___*:
1. Split Line using at spaces
2. Timestamp = First 3 Splitted Values
3. Message = Rest

Listing 3.12: The less than operator of the class

```
def __lt__(self, other):
   return self.timestamp_unix < other.timestamp_unix
```

Finding Events:
In order to be able to manipulate individual or multiple events in a log file efficiently, a method had to be devised to filter them in a meaningful way. The *filter_events* function was created for this purpose, the pseudocode of which can be found in Listing 3.14. Our event objects consist of the *Timestamp* and the *Message*, which contains the rest of the event. To filter the events, a *Target Timestamp* and a *Target Message*, which must be included in the event message, can be specified. If neither is specified, all events are returned. A special aspect of Syslog is that

[31] `https://docs.python.org/3/library/stdtypes.html#list.sort` (*Accessed On: 18.05.2024*)
[32] `https://docs.python.org/3/library/datetime.html` (*Accessed On: 18.05.2024*)

Function *change*:
1. Split Timestamp in Date and Time
2. Save old Unixtime in temporary value and compute and save new Unixtime in the event object
3. Search Message part of the event for old Unixtime and replace it with new Unixtime

Function *shift*:
1. Save old Unixtime in a temporary value
2. Add Shift Factor to old Unixtime
3. Generate data and time from new Unixtime
4. Search Message for Old Unixtime and replace with new Unixtime

Listing 3.13: Excerpt of a Syslog file highlighting the occurrence of an **extra timestamp** in an event

```
Apr 7 08:45:25 Niclas-Rechner kernel: [ 21.618066] audit: type=1400 audit(1712472325.546:114):
    apparmor="DENIED" operation="capable" class="cap" profile="/usr/lib/snapd/snap-confine" pid
    =2028 comm="snap-confine" capability=12 capname="net_admin"
```

the *Timestamp* is compared directly. In contrast, *unix-timestamps* are compared for the other logfiles. This functionality can be extended in the future so that time ranges can be specified.

Listing 3.14: Pseudocode for finding the searched events in the logfile. The user can supply a timestamp and a message, and the events are filtered for these parameters. If no parameters are submitted, all events get returned. Full code can be found at Appendix B, Listing B.10

```
Function Filter Events::
    foreach Event in Events do
        if Event.Timestamp == SearchedTimestamp OR Event.Timestamp == None then
            if Event.Message contains(SearchedMessage) OR Event.Message == None then
                Event is part of the searched Events
            end
        end
    end
```

As previously stated, the Syslog events lack the *year* component of the timestamp. Fortunately, a solution to this issue has been identified.

3.6.2. Year Generation

Two functions have been implemented to reconstruct the year of a Syslog event. These functions are named *year_generator_simple()* and *year_generator_complex()*. DiskForge uses the simple year generator by default, which provides satisfactory and reliable results in most cases. However, since there are instances where the simple approach can be erroneous, a second, more complex approach has been implemented. It provides more accurate results but has a considerably longer runtime than the simple approach. The two approaches are presented in the following two sections.

Simple Year Generation

Listing 3.15: Pseudocode describing the simple year generation process. The full code can be found in Appendix B, Listing B.7

```
Function Simple Year Generator:
    last_month = last month of file metadata
    last_year  = last year of file metadata
    reversed_list = list of events reversed
    foreach Event in reversed_list do
        if Event.Month > last_month then
            last_year -= 1
        end
        last_month = Event.Month
        Event.Year = last_year
    end
    Events = Unreversed reversed_list
```

In the simple approach, the metadata, namely the modified timestamp of the log file, is utilized. Listing 3.15 illustrates the sequence of the function in pseudocode. Initially, the modified timestamp of the inode is extracted and partitioned into year and month. Subsequently, the order of the event list is reversed to position the most recent event at the beginning. Then, the list is processed event by event. For each event, the month is determined whether it is greater than the month of the previous event or, in the case of the first event, greater than the month of the modified timestamp. If this is the case, a year is assumed to have elapsed, as the events are in descending order in the list, and the year is decreased by one. Subsequently, the saved last month is set to the month of the event, and the event receives the saved last year. This process is repeated until all events have been assigned a year. Once this is done, the list is reversed to its original order. As previously mentioned, this approach works in the majority of cases. However, in instances where the modified timestamp is incorrect, or there are time jumps of more than one year in the log file, this procedure does not provide accurate results. For this reason, we have decided to implement a second complex approach that solves this problem more effectively.

Complex Year Generation

The complex algorithm employs the log2timeline[33,34] [6] tool to analyze the log file and generate the years. Log2timeline, a timelining tool, can analyze entire disk images and individual logfiles and generate a timeline with them. In analyzing logfiles, the algorithm employs the timestamps of the files themselves and additional timestamps of events, as illustrated in Listing 3.13, to facilitate more accurate calculations of the year numbers. Listing 3.16 illustrates the operation of our *year_generator_complex* function in pseudocode. Initially, the psteal[35] function of log2timeline is employed to parse the log file. The output of this function is a CSV file whose entries are no longer in the same order as our events. Consequently, the subsequent phase of the process entails a comprehensive examination of the list of events in the CSV file, event by event. For each event in the CSV file, the message and timestamp of the log file events are compared to the

[33] `https://github.com/log2timeline/plaso` (*Accessed On: 18.05.2024*)
[34] `https://plaso.readthedocs.io/en/latest/` (*Accessed On: 18.05.2024*)
[35] `https://plaso.readthedocs.io/en/latest/sources/user/Using-psteal.html` (*Accessed On: 18.05.2024*)

event in question. The log file events that match are then assigned the year of the CSV event. This process is repeated for each event in the CSV file until the complete list has been processed.

As previously stated, this methodology is associated with a relatively lengthy execution time. However, in some instances, it may yield more optimal results than the simple approach. If the results of the simple approach appear to be erroneous, it is recommended that the complex approach be employed once more to obtain more accurate outcomes.

Listing 3.16: Pseudocode describing the complex year generation algorithm. The full code can be found in AppendixB, Listing B.8

```
Function Complex Year Generator:
    CSV = Parsed Logfile using log2timeline
    Events = Events parsed from Logfile
    foreach Entry in CSV do
        Timestamp = Extracted from Entry
        Message = Extracted from Entry
        Matches = Filter Events for Event who matches Timestamp and Message
        foreach Match in Matches do
            Match.Year = Timestamp.Year
        end
    end
```

4

Evaluation

This chapter presents a comparative analysis of the DiskForge framework against other approaches, namely Touch and Debugfs for Metadata manipulation, Gedit and Sublime Text for Logfile manipulation, and SQLiteBrowser for Database manipulation. To cover a wide range of use cases, we have designed a test case for each of DiskForge's three timestomping modules, described in more detail below. This thesis includes a dataset of manipulated disk images showcasing the diverse functionalities of the DiskForge framework. The presented test cases represent a mere fraction of this extensive dataset, which underwent comprehensive analysis. Consequently, the findings of this evaluation can be found within the dataset.

Metadata Test Case:
The objective of the metadata test case is to set the ext4 inode timestamps, Accessed, Modified, Changed, and Created, to predefined values. If a tool cannot set one of the timestamps to the specified value, it is then checked whether another value could be set for the timestamp. If this is also not possible, the timestamp will not be changed. The tools Touch, Debugfs, and DiskForge were selected for the metadata test case. The tools Timestomp and setMACE are also discussed, but they cannot pass the test case due to their incompatibility with ext4. Nevertheless, they offer interesting approaches and use cases.

Logfile Test Case:
The Logfile manipulation test case aims to identify events within a log file that contain a specific keyword and subsequently shift the timestamps of these events into the future by a specified factor. The final step of this test case is to restore the chronological order of the log file. For the Logfile test case, we evaluated the Gedit, SublimeText, and DiskForge.

Database Test Case:
The Database manipulation test case aims to identify and filter URLs in a browser database that

contain a specific keyword and then shift the timestamp of these entries by a specified factor. For the Database test case, we chose to evaluate the SQLiteBrowser and DiskForge.

Table 4.1 provides a comprehensive overview of the tools utilized for each test case.

	Touch	**Debugfs**	**Gedit**	**Sublime**	**SQLiteBrowser**	**DiskForge**
Metadata	✓	✓	✗	✗	✗	✓[1]
Logfiles	✗	✗	✓	✓	✗	✓[2]
Databases	✗	✗	✗	✗	✓	✓[3]
Version	8.32	1.46.5	41.0	Build 4196	3.12.1	

Table 4.1.: Overview of test cases and used approaches

We are going to evaluate each tool by the following points:

- **Preparation:** What preparation is needed to use the tool on a disk image?
- **Functionality:** What is the functionality of each tool, and how much can be manipulated?
- **Traces:** What traces are generated by the tool, and are they detectable?

Given the nature of a master's thesis, it is not feasible to present every detail of the analysis of each test case. Instead, selected parts will be investigated in greater depth. A collection of the manipulated disk images and files will be available for the interested reader in our GitLab project[4].

It is crucial to note that the program under examination does not directly cause some of the traces we encounter. Some are generated by other processes or the operating system, highlighting the influence of these external factors in our work. These traces can occur as most tools work on a mounted disk image. These traces show that even if we try to mitigate the manipulation's traces, some parameters are out of our control and can leave traces we would typically not expect.

For the test case execution and evaluation, we used a Laptop. Table 4.2 summarizes the relevant system specifications.

Component	**Specification**
OS	Ubuntu 22.04.4 LTS
OS Type	64-bit
CPU	i5-7200U CPU
Kernel	6.5.0-27-generic

Table 4.2.: System specifications of the laptop used for this evaluation

The rest of the evaluation section is structured as follows:
First, the general workflow of the test cases is described. This is followed by an analysis of the base disk image and the traces created by the mounting process. Next, the metadata, log files, and database test cases are executed, and each tool is evaluated and compared.

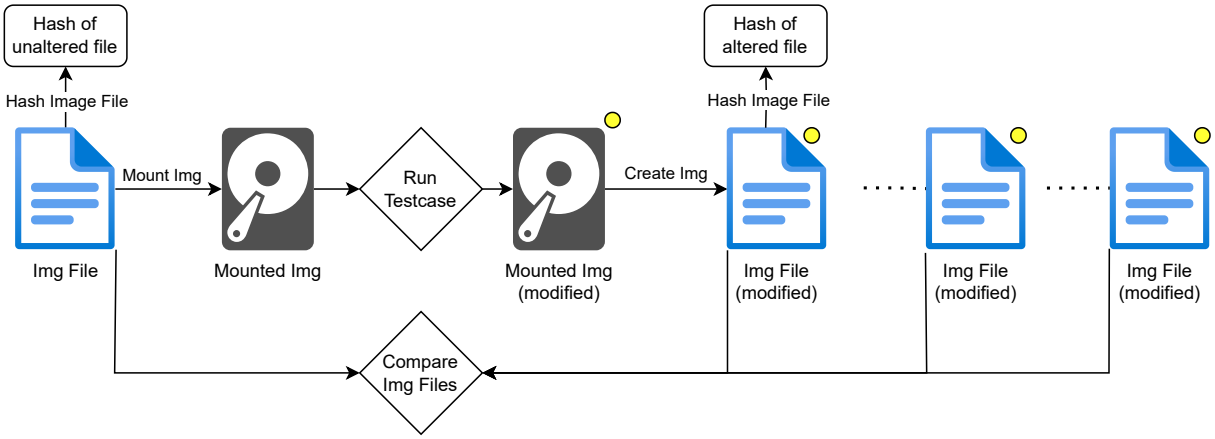

Figure 4.1.: Workflow of the test case execution process

4.1. Methodology

Figure 4.1 provides an overview of the workflow of a test case. The first step is to hash the copy of the unchanged disk image to ensure that each test case is executed on the same image. Next, the necessary preparations are made to execute the test case. The disk image must be mounted for all test cases except those of Debugfs and DiskForge. After all preparation work is completed, the test case is executed. If the disk image is mounted, it is unmounted in the subsequent step. For verification purposes, the disk images are hashed once more after completing the test case. In most cases, the hash will vary from one execution to another as the operating system leaves traces. For instance, the operating system modifies the *Last Written on* timestamp with the current date and time. It is expected that the repeated execution of Debugfs and DiskForge will generate identical changes and, consequently, identical hash values with each run. Nevertheless, the collected hash values are available in Appendix C and in our GitLab.

To identify the alterations resulting from the execution of the test case, we employ the *cmp*[5] command illustrated in Appendix D, Listing D.1. After that, we import the file generated by the *cmp* command into our CompareParser.py[6] script. This script provides us with data regarding the number of bytes altered, the blocks in which these changes occur, and the usage of these blocks in the unmodified and modified disk image and the modified inodes.

It is essential to repeat that not all changes originate directly from the tool under investigation. Other processes or the operating system may have contributed to these alterations. This challenge cannot be avoided when working with approaches that operate on a mounted disk image. Therefore, this factor is a significant advantage of DiskForge, as it does not rely on a mounted disk image.

Selective blocks were extracted and compared to the corresponding blocks from the unmodified disk image to understand the exact changes better. In the remainder of this evaluation, we

[1] Using the Metadata module
[2] Using the Logfile module
[3] Using the Database module
[4] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge` (*Accessed On: 18.05.2024*)
[5] `https://man7.org/linux/man-pages/man1/cmp.1.html` (*Accessed On: 21.05.2024*)
[6] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/Other/CompareParser.py` (*Accessed On: 18.05.2024*)

will focus more on the findings we could draw from this and less on how we obtained them. Nevertheless, we provide annotated hexdumps of these blocks in Appendix E and in our GitLab.

The steps mentioned above can be summarized as follows:

1. Create a copy of the base disk image.
2. Create *SHA-256* hash of the image to check that all tests run on the same image.
3. Apply the preparation needed for the tool.
4. Try to solve the test task.
5. If mounted, unmount the disk image.
6. Create *SHA-256* hash of the manipulated disk image.
7. Check the manipulated image against the base image.

In addition, the Mounting Artifacts Test case was conducted to elucidate the traces generated by mounting the disk image.

4.2. Baseline

The disk image selected for all test cases is the 'asservat_74382-23.img' provided by the FAU IT Security Infrastructures Lab. This disk image was chosen because the tool's intended use case is modifying disk images for digital forensics education purposes. This chosen file was utilized in digital forensics exercises, aligning with our intended use case. In this section, we aim to extract some fundamental data about this disk image, which will serve as reference points in the subsequent stages of the evaluation.

The disk image has a size of 32.2 GB and is divided into eight partitions, as can be seen in Listing 4.1. Of particular importance for this evaluation is the partition with the number *006*, which has an offset of *1,054,720* sectors to the beginning of the disk image file and contains an ext4 file system.

Listing 4.1: mmls output for the base disk image
Partition of Interest, **Start of the Partition**, **Sector Size**

```
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Safety Table
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  Meta      0000000001   0000000001   0000000001   GPT Header
003:  Meta      0000000002   0000000033   0000000032   Partition Table
004:  000       0000002048   0000004095   0000002048
005:  001       0000004096   0001054719   0001050624   EFI System Partition
006:  002       0001054720   0062912511   0061857792
007:  -------   0062912512   0062914559   0000002048   Unallocated
```

Subsequently, fsstat was employed to extract file system metadata, including the *Last Written*, *Last Checked*, and *Last Mounted* timestamps, as well as the *Last Mounted on* path. The shortened fsstat output is presented in Listing 4.2, showing that all timestamps are dated to 2023 and the *Last mounted on* path is set to '/'.

Furthermore, the following files were selected for the test cases:

Listing 4.2: fsstat output for the partition 006

```
Timestamps:
Last Written at: 2023-06-19 16:12:32 (CEST)
Last Checked at: 2023-06-10 01:05:57 (CEST)
Last Mounted at: 2023-06-19 16:12:34 (CEST)

Last mounted on: /
```

- *syslog*[7]: The Syslog file is employed in the log file test case.
- *places.sqlite*[8]: The places.sqlite file is employed in two distinct test cases: the metadata test case and the database test case.

For both files, the inode number was first extracted using fls, and then the inode metadata was extracted using istat. Of particular interest to us were the length of the file and the four ext4 inode timestamps. Listing 4.3 shows the shortened istat output for the Syslog file, whereas Listing 4.4 shows the output for the places.sqlite file. As seen in Listing 4.4, the direct blocks of the places.sqlite inode contains references to block 0, which is assigned to the ext4 superblock. It is important to emphasize that no file may store its data in this location, as doing so would corrupt the file system. We hypothesize that this discrepancy is due to the size of the places.sqlite file. According to the inode, the file's size is 5242880 bytes, which requires 1280 blocks for storage. However, places. sqlite allocated only 344 blocks, with the remaining blocks displayed by istat as 0. This indicates that the size of the places.sqlite file is faulty, the reason for this remains unknown.

Listing 4.3: Istat output for the Syslog file on the base image

```
Inode Times:
Accessed: 2023-06-18 18:35:11.788000060 (CEST)
File Modified: 2023-06-19 19:36:51.351086939 (CEST)
Inode Modified: 2023-06-19 19:36:51.351086939 (CEST)
File Created: 2023-06-18 18:35:11.788000060 (CEST)

Size: 771866
```

[7]Full path: /var/log/syslog
[8]Full path: /home/kassensystem/snap/firefox/common/.mozilla/firefox/ij9d8zia.default/places.sqlite

Listing 4.4: Istat output for the places.sqlite file on the base image

```
Inode Times:
Accessed: 2023-06-19 19:36:38.303086884 (CEST)
File Modified: 2023-06-19 19:36:43.735086907 (CEST)
Inode Modified: 2023-06-19 19:36:43.735086907 (CEST)
File Created: 2023-06-10 01:32:18.332726057 (CEST)

Size: 5242880

Direct Blocks:
2740941 2740942 2740943 2740944 2740945 2740946 2740947 2740948
5162222 5162223 5162224 5162225 5162226 5162227 5162228 5162229
...
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

4.3. Mounting Artifacts

We conducted the Mounting Artifacts test case to identify which traces are created when mounting a disk image. This allowed us to better distinguish between traces caused by mounting and those caused by the execution of the respective test case in subsequent test cases. The procedure for this test case is as follows: first, partition 006 of the disk image is mounted, and immediately after it is confirmed that the mounting was successful, it is unmounted again. Subsequently, the modified disk image is compared with the base disk image.

4.3.1. Execution

This section describes the procedure employed for executing the test case. The terminal commands utilized for these steps can be found in Appendix D Section D.2.

1. **Create Mountpoint:**
   The initial step in mounting the disk image is to create a new mount point, which all subsequent test cases can utilize.
2. **Identify Partition Offset:**
   The next step is to calculate the partition offset in bytes using the formula in Equation 4.1.

$$\begin{aligned} \text{Byte Offset} &= \text{mmls Offset} \cdot \text{Sector Size} \\ &= \mathbf{1054720} \cdot \mathbf{512} \\ &= 540016640 \end{aligned} \tag{4.1}$$

Note: The values for ***mmls Offset*** and ***Sector Size*** have been taken from the output of mmls, see Listing 4.1

3. **Mount Image File:**
   Next, partition *006* of the disk image partition can be mounted.
4. **Verify Mount:**
   It is now necessary to verify whether the mounting process was successful. If so, the next step may be initiated; otherwise, the previous step must be repeated.

5. **Unmount Partition:**
   The disk image partition can be unmounted at this point in the test case process.
6. **Verify Unmount:**
   Finally, it was rechecked that the partition had been successfully unmounted.

The modifications resulting from the mounting artifacts test case are presented and analyzed in the following section.

4.3.2. Analysis

In order to ascertain which alterations have been made to the disk image, the cmp command was first executed as shown in Appendix D Listing D.1. Our CompareParser was then run with the output of this command, which yielded the results depicted in Listing 4.5.

Listing 4.5: Output of the parsing tool for the mounting test

```
How many Bytes have been changed: 14
How many Blocks have been changed: 1
Block Numbers:
[0]
```

As illustrated by the output of the parsing tool in Listing 4.5, 14 bytes have been altered in block number 0. Block number 0 contains the superblock of the ext4 file system, which stores essential metadata. To ascertain the specific changes, we extracted block number 0 and compared it to block 0 of the base image.

The ext4 Superblock (Block 0):
In order to identify what changes have been made to block 0, we extracted the block from both the modified and unmodified disk images. Once it was established that all the changes were in the superblock, we removed the first 1024 and the last 2048 bytes of the block[9]. The remaining part of the two blocks was then subjected to a detailed examination in a hex editor. The resulting annotated hexdump can be found in Appendix E, Listing E.1. Next, the altered bytes were matched to the superblock structure described in the ext4 wiki [9]. This revealed that in addition to the *last mount time*, and *last write time*, the *mount count*, the *snapshot inode number*, and the *superblock checksum* were also changed. In Tables 4.3 and 4.4, the information contained in the *last mount time*, *last write time*, and the *mount count* was extracted and converted for the modified and unmodified disk image respectively. This revealed that the values in the superblock, see Table 4.4, now contain the timestamps of the mount attempt and that the *mount count* was increased by 1. Furthermore, a comparison was made between the results obtained and the output of fsstat in Listing 4.6, which matches the observed results. Notably, the *Last Mounted on* parameter, as illustrated in Listing 4.6, has not been modified.

[9]The structure of block 0 can be found in Section 2.3.1

Listing 4.6: fsstat comparison between base image and mount test image

```
Base Disk Image:
Last Written at: 2023-06-19 16:12:32 (CEST)
Last Checked at: 2023-06-10 01:05:57 (CEST)
Last Mounted at: 2023-06-19 16:12:34 (CEST)
Last mounted on: /

Mount Test Image:
Last Written at: 2024-03-22 12:45:43 (CET)
Last Checked at: 2023-06-10 01:05:57 (CEST)
Last Mounted at: 2024-03-22 12:45:20 (CET)
Last mounted on: /
```

	Hexadecimal	Decimal	Timestamp
s_mtime	52 62 90 64	1687183954	Monday, 19. June 2023 16:12:34 GMT+02:00 DST
s_wtime	50 62 90 64	1687183952	Monday, 19. June 2023 16:12:32 GMT+02:00 DST
s_mount_count	0F 00	15	

Table 4.3.: Value conversions for the base image.
Note: The Hexadecimal representation is in little-endian, as this is the endianness of the stored values

	Hexadecimal	Decimal	Timestamp
s_mtime	50 6F FD 65	1711107920	Friday, 22. March 2024 12:45:20 GMT+01:00
s_wtime	67 6F FD 65	1711107943	Friday, 22. March 2024 12:45:43 GMT+01:00
s_mount_count	10 00	16	

Table 4.4.: Value conversions for the mount test image.
Note: The Hexadecimal representation is in little-endian, as this is the endianness of the stored values

4.4. Metadata Manipulation

The objective of the Metadata Manipulation test case is to ascertain the efficacy of the tested tools in modifying the timestamps of ext4 inodes and the resulting traces. The test case is based on the *places.sqlite*[10] file, which can be found on the disk image with inode number 1311789. Table 4.5 shows the timestamps of this inode on the unmodified disk image. Table 4.6 shows the values to which each tested approach should set the corresponding timestamp. We evaluated whether the tools can complete all of these modifications and determine how the execution of each approach modifies other data. If an approach fails to set a timestamp to the specified value, namely Touch, we try to set an arbitrary value for that timestamp. If this also fails, we leave the timestamp unchanged. Besides Touch, Debugfs, and DiskForge, we briefly discuss Timestomp and SetMACE tools. These two tools can not complete this test case; they only work on the NTFS file system. Despite their inability to complete this test case, we include both tools in this section to highlight their unique use cases and techniques.

Timestamp Name	Timestamp Value
Accessed	2023-06-19 19:36:38.303086884
Modified	2023-06-19 19:36:43.735086907
Changed	2023-06-19 19:36:43.735086907
Created	2023-06-10 01:32:18.332726057

Table 4.5.: Original pairs of timestamp and corresponding value

Timestamp Name	Timestamp Value
Accessed	2024-01-02 01:02:03.123456789
Modified	2025-02-03 01:02:03.123456789
Changed	2026-03-04 01:02:03.123456789
Created	2027-04-05 01:02:03.123456789

Table 4.6.: Pairs of timestamp and corresponding value it should be set to by the test case execution

4.4.1. Touch

The first approach we evaluated is using the Touch[11] command to manipulate metadata timestamps. The touch command is part of the coreutils package, which comes preinstalled with the Ubuntu operating system. It is a terminal command that can be utilized to create files and modify the timestamps of files.

Test Case Satisfiability

Unfortunately, the Touch command cannot fully satisfy the specified test cases. The microseconds cannot be changed for the timestamps of Accessed and Modified. With Changed, the timestamp can only be set to the current date and time, and Created cannot be changed in any way. Listing 4.8 shows the modifications done to the test case to accommodate these restrictions.

[10] Full Path: *home/kassensystem/snap/firefox/common/.mozilla/firefox/ij9d8zia.default/places.sqlite*
[11] `https://man7.org/linux/man-pages/man1/touch.1.html` (*Accessed On: 18.05.2024*)

	Accessed	Modified	Changed	Created
Touch	✓/✗	✓/✗	✓/✗	✗

Table 4.7.: Summary: Test case satisfiability of the Touch approach

With this new set of parameters, Touch can complete the test case. It is also possible to change

Timestamp Name	Timestamp Value
Accessed	2024-01-02 01:02:03
Modified	2025-02-03 01:02:03
Changed	Current System Date and Time
Created	No Modification

Table 4.8.: Modified pairs of timestamp and corresponding value it should be set to by the Touch Metadata Test Case

the timestamp by changing the system time and using a script that changes the timestamp at the exact moment when the system time matches the specified time, but this is much more complicated. However, this approach would make it possible to 'set' the subseconds of the timestamp.

Test Case Execution

In order to run the test case, the disk image was first mounted following the instructions set out in Section 4.3.1. It was necessary to take into account the peculiarity of the Touch command. Although it is possible to change only the Accessed timestamp or the Modified timestamp, it is impossible to change the Changed timestamp alone. Therefore, the Changed timestamp must first be set with the Touch command, which changes also the Accessed and Modified timestamps. The Accessed and Modified timestamps can only be set then. If this sequence is not adhered to, modifying the Changed timestamp will overwrite the changes made to the Accessed and Modified timestamps. The commands used to execute the test case can be found in Appendix E, Listing E.1. At last, the disk image was unmounted again using the commands outlined in Section 4.3.
To determine whether the test case had been executed successfully, we employed the istat command to examine the newly generated timestamps of the places.sqlite file. The comparison between the expected timestamps in Table 4.8 and the modified timestamps in Listing 4.7 showed that the test case was successfully executed.

Listing 4.7: New Timestamps of the places.sqlite file

```
Inode Times:
Accessed: 2024-01-02 01:02:03.000000000 (CET)
File Modified: 2025-02-03 01:02:03.000000000 (CET)
Inode Modified: 2024-03-24 16:37:51.772748242 (CET)
File Created: 2023-06-10 01:32:18.332726057 (CEST)
```

Analysis

To identify the differences between the modified and the base disk images, the cmp command was first executed, followed by the CompareParser.py script. The results[12] of this analysis indicate that 13167 bytes in 13 blocks have been modified. Some selected changes are examined in greater detail below. As in the previous test case, block 0 has been modified. Also, blocks 3702784 to 3702794, which belong to the ext4 journal[13], have been altered. Finally, block 5242978, which belongs to block group 160, was also modified. A summary of the changed Blocks and their usage can be found in Appendix E, Table E.1.

The ext4 Superblock (Block 0):
First, we analyzed block 0. Based on the previous Mounting Artifacts test case results, we anticipated that this block would change. We extracted the block from this test case to ascertain whether the same bytes were altered in both test cases and compared it with block 0 of the Mounting Artifacts test case. Our findings revealed that more bytes were modified in the Metadata Touch test case. In contrast to the Mounting Artifacts test case, the *Last Mounted on* parameter was also updated this time, as seen in Listing 4.8. An annotated hexdump of this comparison can be found in Appendix E, Figure E.2.

Listing 4.8: fsstat comparison between base disk image and metadata toch test case disk image

```
Last Written at: 2024-03-24 16:37:57 (CET)
Last Checked at: 2023-06-10 01:05:57 (CEST)
Last Mounted at: 2024-03-24 16:37:13 (CET)

Last mounted on: /mnt/master
```

Inode of the places.sqlite file (Block 5242978):
This block is part of block group 160 and is used as part of the inode table. Through our ComparisonParser tool, we have determined that the changed bytes belong to inode 1311789, which is the inode of the places.sqlite file. To ascertain which data was changed, we extracted the block and compared it with the block of the base disk image. A hexdump of this comparison can be found in Appendix E, Figure E.3. This revealed that the Accessed, Modified, and Changed timestamps, the Changed extra timestamp, and the *inode checksum* had been altered. In Table 4.9, the extracted data was converted into timestamps to verify that the test case had been completed. The altered timestamps correspond with the values entered for the test and the istat output of the inode in Appendix E, Listing E.3.

Timestamp	Hexadecimal	Decimal	Timestamp
Accessed	0x7B529365	1704153723	Tuesday, 2. January 2024 01:02:03
Modified	0x7B07A067	1738540923	Monday, 3. February 2025 01:02:03
Changed	0xCF480066	1711294708	Sunday, 24. March 2024 16:38:28
Changed Extra	0x48CF3CB8	3090992968	0.772748242

Table 4.9.: Conversion of the bytes into timestamps.
Note: All timestamps are given in GMT+01:00 (CET)

[12] A more detailed output can be found in Appendix E Listing E.2
[13] https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Journal_.28jbd2.29 (*Accessed On: 18.05.2024*)

Verdict

The Touch command is a quick and easy way to manipulate some file metadata timestamps. However, its capabilities are severely limited for this application, as the timestamps of the Accessed and Modified fields show a nulled microsecond field after the manipulation. Also, modifying the Changed timestamp is limited to the current system time. The Created timestamp cannot be modified at all. As mentioned in Section 4.4.1, changing the system time to set the timestamps would be a solution to changing these timestamps. Also, a copy operation can change the created timestamp by copying the file, deleting the original file, and setting the copied file to the name of the original file. For our use case, there are better choices than the Touch approach because it modifies the disk image's superblock and cannot accurately modify the timestamps.

4.4.2. Debugfs

The Debugfs[14] tool is an interactive file system debugger that can be used to debug extX file systems. One of its features is the ability to modify file system Metadata values. We used this feature to manipulate the file metadata timestamps.

Test Case Satisfiability

	Accessed	Modified	Changed	Created
Debugfs	✓	✓	✓	✓

Table 4.10.: Test case satisfiability of the Metadata – Debugfs test case

Debugfs works on an unmounted disk image that only needs to be configured as a loop device. Internally, it uses the e2fsprogs library to access file system metadata. Because of these two components, it can access and set most of the file system metadata, including file metadata timestamps. This allows Debugfs to complete the metadata test case.

Test Case Execution

In contrast to the Metadata Touch Test case, the disk image does not require mounting for Debugfs. However, a loop device must be created. The procedure for this is shown in Appendix E, Listing E.4. Next, Debugfs was started, and we navigated to the folder of places.sqlite within Debugfs. The Accessed, Modified, Changed, and Created timestamps were then changed, followed by the respective Extra timestamps. With the Extra Timestamps, it must be noted that multiplication was necessary first, as the value needed to be shifted by 2 bits - Equation 4.2 shows the calculation for this purpose. Finally, Debugfs could be closed. The commands for this process can be found in Appendix E, Listing E.5, and E.6.

$$\begin{aligned} \text{Extra Value} &= \text{Desired Value} \cdot 2^{\#\text{Bit Shift}} \\ \text{Extra Value} &= 123456789 \cdot 2^2 \\ \text{Extra Value} &= 493827156 \end{aligned} \tag{4.2}$$

[14] `https://linux.die.net/man/8/debugfs` (*Accessed On: 18.05.2024*)

At last, the loop device was freed using the command outlined in Appendix E, Listing E.7. To verify that the test case passed, we utilized *istat* command, which resulted in the timestamps shown in Appendix E, Listing E.8. Unfortunately, the timestamps do not match the expected result of the test case. The Debugfs utility apparently interprets the input of the date fields as *Greenwich Mean Time (GMT)*[15] *+00:00* and disregards possible alternate time zones. To resolve this issue, it was necessary to identify the time zones used for the timestamps, as shown by the istat command. The Accessed, Modified, and Changed timestamps were observed to be in the *CET (Central European Time)*[16] time zone, corresponding to *GMT+01:00*. The Created timestamp was found to be in the *CEST (Central European Summer Time)* time zone, which corresponds to *GMT+02:00*. The *GMT* offset of the *CE(S)T* time zones was subtracted from the corresponding timestamps to consider this behavior. This resulted in the updated timestamps shown in Table 4.11.

Timestamp Name	Timestamp Value
Accessed	2024-01-02 00:02:03.123456789
Modified	2025-02-03 00:02:03.123456789
Changed	2026-03-04 00:02:03.123456789
Created	2027-04-04 23:02:03.123456789

Table 4.11.: Updated pairs of timestamp and corresponding value

To obtain correct results, the test case was re-executed according to the steps previously described in the initial test case execution. This time, the updated timestamp values were employed to accommodate the time zones. The commands using the updated timestamps can be found in Appendix E, Listing E.9.

Following our exit from Debugfs, we released the previously allocated loop device again. Listing 4.9 shows the output of the istat command for the updated Debugfs commands.

Listing 4.9: Istat output for the Metadata - Debugfs test case

```
Inode Times:
Accessed: 2024-01-02 01:02:03.123456789 (CET)
File Modified: 2025-02-03 01:02:03.123456789 (CET)
Inode Modified: 2026-03-04 01:02:03.123456789 (CET)
File Created: 2027-04-05 01:02:03.123456789 (CEST)
```

The timestamps displayed in Listing 4.9 now correspond to the timestamps indicated in the test case description, which marks a successful test case completion.

Analysis

In order to identify which data had been modified, the cmp command and the CompareParser were executed. This resulted in 36 Changed Bytes in 1 Block[17]. In this test case, only the block number 5242978 has been changed. The results of the CompareParser indicate that the changed inode in the table is the inode number 1311789, which belongs to the places.sqlite file. A summary of the changed blocks and their usage can be found in Appendix E, Table E.2.

[15] `https://en.wikipedia.org/wiki/Greenwich_Mean_Time` (*Accessed On: 18.05.2024*)
[16] `https://en.wikipedia.org/wiki/Central_European_Time` (*Accessed On: 18.05.2024*)
[17] More of the Compare Parser output can be found in Appendix E, Listing E.10

Inode of the places.sqlite file (Block 5242978):
This block belongs to group 160 and is used as an inode table block. Our parsing tool indicates that the metadata of inode 1311789 had been altered within the inode table block.

The block was extracted and compared to the corresponding block from the unaltered disk image to identify the specific alterations. It was determined that the Accessed, Modified, Changed, and Created timestamps, as well as their extra timestamps and the inode checksums, had been set to the values defined in the test case. The annotated comparison of both hex dumps can be found in Appendix E, Figure E.4.

Verdict

Debugfs is an effective tool for modifying inode timestamps under ext4. By utilizing the e2fsprogs[18] library, DiskForge can effectively replicate the behavior of ext4, modifying only the minimum number of bytes necessary to complete the test case. Beyond that, no other data was altered. The sole disadvantage of Debugfs for our use case is that the timestamps are interpreted as GMT+00:00, and the user has to manually convert the timestamps.

4.4.3. DiskForge

The DiskForge tool is the framework we developed during this master thesis. We utilized the *Metadata* module with the *changeTimestamps* functionality for the Metadata test case.

Test Case Satisfiability

	Accessed	Modified	Changed	Created
Temporal Forge	✓	✓	✓	✓

Table 4.12.: Test case satisfiability of the Metadata-DiskForge test case

The DiskForge tool can access and modify all of the necessary timestamps, enabling the completion of all test cases. Consequently, the test case was executed without needing any modifications to its parameters.

Test Case Execution

Since the disk image for the DiskForge test case did not require mounting, we could commence the test case immediately. We employed the command illustrated in Listing 4.10 to execute the test case. We then utilized istat to verify that the timestamps had been altered as intended. As illustrated in Listing 4.11, all timestamps were set to the values defined in the test case.

Analysis

Since the hashes of the Debugfs[19] and DiskForge[20] test cases match, we assumed the modifications were identical. Nevertheless, we analyzed the changes below. First, we analyzed the changes using

[18] `https://e2fsprogs.sourceforge.net/` (*Accessed On: 18.05.2024*)
[19] Debugfs' modified Disk Image hash is found in Appendix C
[20] Diskforges modified Disk Image hash is found in Appendix C, Listing C.8

Listing 4.10: DiskForge command utilized for the Metadata Manipulation Test Case and description of the parameters

```
python3 ./DiskForge.py --func timestomp --target metadata --mode change --img <Path to Image> --
    file <Path to places.sqlite> --partnum 6 --timestamp accessed file_modified inode_modified
    file_created --ts 2024-01-02_01:02:03.123456789 2025-02-03_01:02:03.123456789 2026-03-04_01
    :02:03.123456789 2027-04-05_01:02:03.123456789
```

The parameters are explained in the following:

- *func*: Functionality we want to use
- *target*: The type of data we want to manipulate
- *mode*: How we want to manipulate the data
- *img*: Path to the disk image file
- *file*: File we want to manipulate
- *partnum*: Partition on which the file resides
- *timestamp*: What timestamps should be modified
- *ts*: Values of the new timestamps

Listing 4.11: istat output for the modified places.sqlite file

```
Inode Times:
Accessed: 2024-01-02 01:02:03.123456789 (CET)
File Modified: 2025-02-03 01:02:03.123456789 (CET)
Inode Modified: 2026-03-04 01:02:03.123456789 (CET)
File Created: 2027-04-05 01:02:03.123456789 (CEST)
```

cmp and the parsing tool, which shows changes in 1 block and 36 bytes, just like in Section 4.4.2. An overview of the changed blocks and their usage can be found in Appendix E, Listing E.3. Subsequently, the block was extracted and compared to the base image block. The modifications observed in the DiskForge test case were identical to those created by the Debugfs test case.

Verdict

DiskForge completed the test case without modifying the parameters. Furthermore, as with Debugfs, only a few bytes were altered, indicating no external tracks were created. In execution, the sole distinction between DiskForge and Debugfs is that DiskForge considers the time zone. Otherwise, the modifications made by Debugfs and DiskForge are identical, which is a positive aspect as Debugfs employs the e2fsprogs library, the same library utilized by ext4.

4.4.4. Comparison

Table 4.13 illustrates that only Debugfs and DiskForge could fulfill all test cases. Furthermore, the modifications performed by Debugfs and DiskForge are identical. Table 4.14 depicts the number of bytes and blocks altered and the number of bytes altered as a percentage to facilitate a more straight forward comparison. The value of 100% corresponds to the highest number of altered bytes. As illustrated in the preceding table, Debugfs and DiskForge each modify a relatively small fraction of Touch's altered number of bytes. This discrepancy is primarily attributable to Touch modifying not only the inode of places.sqlite, but also the superblock and the journal.

	Accessed	Modified	Changed	Created
Touch	✓/✗	✓/✗	✓/✗	✗
Debugfs	✓	✓	✓	✓
DiskForge	✓	✓	✓	✓

Table 4.13.: Overview of the tested tools and which test cases they could fulfill

	Changed Bytes (absolut)	Changed Bytes (percent)	Changed Blocks
Touch	13167	100%	13
Debugfs	36	0.27%	1
DiskForge	36	0.27%	1

Table 4.14.: Comparison of the amount of modified data between the Touch, Debugfs, and DiskForge approach.
Note: 100% is equal to the number of the most changed bytes (absolut)
The Tool with the lowest number of changed bytes is marked in green.

4.4.5. Excursion

TimeStomp

The first tool we will discuss is MetaSploit's TimeStomp[21]. TimeStomp is a post-exploitation tool that enables the user to modify file timestamps on a compromised system. It is only compatible with NTFS systems, so we did not directly evaluate it with a test case.

After successfully exploiting a system utilizing the MetaSploit framework, a user may deploy TimeStomp. This tool enables the user to conceal their traces.

The tool's functionality is restricted to a system that has been compromised using the Metasploit Framework. Consequently, the tool is not particularly useful for our approach, which would necessitate the creation of a disk image on a virtual machine and subsequent exploitation of the machine in question. Furthermore, if the disk image is not bootable, this approach would require using an exploitable VirtualMachine that employs the disk image as a hard drive.

SetMACE

A tool quite similar to our approach is setMACE [21] by Joakim Schicht. Like DiskForge, this tool accesses the disk directly, thus circumventing all restrictions imposed by the file and operating system. It resolves the file system internally and writes changes directly to the disk.

As it has no restrictions imposed by the file and operating system, it can access and modify all four MACE[22] timestamps of a file. However, since NT6.x Windows blocks direct write access within the volume space. One option to circumvent this restriction is to work on unmounted disks, as with DiskForge. However, as the tool should also be able to modify the system drive or other non-unmountable volumes, a driver was implemented to circumvent the write restriction. In the event of a driver failure, setMACE attempts to unmount the disk to access the volume

[21] `https://www.offsec.com/metasploit-unleashed/timestomp/` (*Accessed On: 21.05.2024*)
[22] Modified, Accessed Changed, Entry Modified

directly, analogous to DiskForge. As with the DiskForge tool, the manipulations are difficult to detect, and the detection is likely based on the violation of implicit time information.

4.5. Logfile Manipulation

The objective of the Logfile Manipulation test case is to shift the timestamps of all entries containing the keyword '*SMBIOS*' from the disk image's Syslog file by '*+12 minutes*' and then restore the entries to their chronological order. Listing 4.12 displays the entries containing the keyword. To perform this test, we selected the Syslog file 'asservat_74382-23.img' in the directory 'var/log/.' Our methodology employs text editors to modify the log files. The two text editors selected for this purpose are Gedit and Sublime Text, as demonstrated by Niclas Pohl [18] to exhibit disparate behaviors when modifying files. Additionally, DiskForge with the Logfiles module and the LogDeltaShift functionality was utilized.

Listing 4.12: Relevant events of the Syslog file and their changed state after the test case completion with their respective line number

```
Unaltered:
1326 Jun 18 22:49:45 kassensystem-orlandoerstube kernel: [ 0.000000] SMBIOS 2.5 present.
3532 Jun 19 16:12:38 kassensystem-orlandoerstube kernel: [ 0.000000] SMBIOS 2.5 present.

Altered:
3506 Jun 18 23:01:45 kassensystem-orlandoerstube kernel: [ 0.000000] SMBIOS 2.5 present.
5527 Jun 19 16:24:38 kassensystem-orlandoerstube kernel: [ 0.000000] SMBIOS 2.5 present.
```

4.5.1. Gedit

Gedit[23] is a lightweight open-source text editor that is installed by default on the Ubuntu operating system.

Test Case Satisfiability

As the Syslog file utilized by the disk image is a text file, Gedit can execute the test case without further modifying the parameters.

Test Case Execution

Before initiating the test case, the disk image needed to be mounted so that the file could be accessed via Gedit. The procedure outlined in Section 4.3.1 was employed for this purpose. The subsequent steps required to execute the test case are presented below.

1. First, the File was searched for the events containing the keyword '*SMBIOS*', which resulted in the events shown in Listing 4.12.
2. Next, the timestamp of the events was shifted by *+12 minutes*, which resulted in the values shown in Listing 4.13.
3. Following the shift, the events in the Syslog file were no longer in the correct chronological sequence. The correct placement was manually identified to resolve this issue, and the events were relocated to their respective position. Listing 4.12 shows the changed events with their correct placement.

[23] `https://help.gnome.org/users/gedit/stable/` (*Accessed On: 18.05.2024*)

Once all modifications had been saved, Gedit was closed, resulting in the appearance of the warning in Listing 4.14. This warning should not affect the test case, but we noted it nevertheless.

To verify whether the test case was successful, we initially employed istat to ascertain whether any alterations had been made to the inode metadata. The istat output for inode number 1184404 is presented in Listing 4.15. Interestingly, this inode had been deleted, prompting us to utilize fls to identify the new inode number of the Syslog file, which yielded the number 1184217. The results of the istat output for this inode can be seen in Listing 4.16. It is apparent that all timestamps, including the File Created timestamp, have been altered, indicating that a new file has been created internally. Finally, the file was extracted from the disk image and verified to have undergone the desired test case changes.

Listing 4.13: The Events after the 12 minute shift

```
1326 Jun 18 23:01:45 kassensystem-orlandoerstube kernel: [ 0.000000] SMBIOS 2.5 present.
3532 Jun 19 16:24:38 kassensystem-orlandoerstube kernel: [ 0.000000] SMBIOS 2.5 present.
```

Listing 4.14: Warning Message produced by Gedit on closing

```
** (gedit:69571): WARNING **: 16:56:58.844: Set document metadata failed: Setzen des Attributs
    metadata::gedit-spell-language nicht unterstützt

** (gedit:69571): WARNING **: 16:56:58.847: Set document metadata failed: Setzen des Attributs
    metadata::gedit-encoding nicht unterstützt

** (gedit:69571): WARNING **: 16:57:00.015: Set document metadata failed: Setzen des Attributs
    metadata::gedit-position nicht unterstützt
```

Listing 4.15: istat output of the original Syslog file inode. The old inode was deleted as indicated by the size of 0 and the deleted timestamp.

```
size: 0
Inode Times:
Accessed: 2024-03-31 16:44:33.942875782 (CEST)
File Modified: 2024-03-31 16:56:58.834451629 (CEST)
Inode Modified: 2024-03-31 16:56:58.834451629 (CEST)
File Created: 2023-06-18 18:35:11.788000060 (CEST)
Deleted: 2024-03-31 16:56:58 (CEST)
```

Listing 4.16: istat output of the new Syslog file inode. As indicated by the updated File Created timestamp

```
size: 771866
Inode Times:
Accessed: 2024-03-31 16:56:58.378447177 (CEST)
File Modified: 2024-03-31 16:56:58.802451317 (CEST)
Inode Modified: 2024-03-31 16:56:58.834451629 (CEST)
File Created: 2024-03-31 16:56:58.378447177 (CEST)
```

Analysis

In the subsequent step, we compared the modified and base disk images and utilized the CompareParser tool to parse the output of the cmp command. This process revealed that 839757 bytes in 252 blocks had been altered. A comprehensive output of the parser is presented in Appendix E, Listing E.11, and an overview of the modified blocks and their usage is provided in Appendix E, Listing E.4. In the following, a few selected changes are examined in greater detail.

ext4 Superblock (Block 0):
Upon examination of block 0, identical alterations were observed as in the Metadata Touch test case. In addition, the *Free Blocks* and *KiB Written* values were also altered. Table 4.15 illustrates the changes to both values. The new Free Block results from the fact that the inode no longer utilizes an additional Extent Block, as evidenced by istat. An annotated hexdump of block 0 can be found in Appendix E, Figure E.5.

	Hexadecimal	**Decimal**
Free Blocks (old)	0xE53C3A00	3816677
Free Blocks (new)	0xE63C3A00	3816678
KiB Written (old)	0xB0506C0200000000	40652976
KiB Written (new)	0xC0546C0200000000	40654016

Table 4.15.: Conversion of the hexdump values of the changes observed in block 0 that were not present in block 0 of the Metadata Touch test case. The changed values are the number of free blocks in the partition and the number of KiB written in the lifetime of the partition.

Data Block of the /var/log/ Directory (Block 4733203):
This block represents the data block of the /var/log/ folder. In ext4, each folder is associated with a data block that stores information about the files within the folder. Upon extraction of the block, it was observed that the inode number field of the Syslog file had changed. Table 4.15 shows the old and new values and their conversions to decimal numbers. This indicates that the previous inode number, 1184404, of the Syslog file has been superseded by 1184217 because Gedit created a new Syslog file. An annotated version of the block hexdump can be found in Appendix E, Figure E.6.

File and Directory Metadata Changes:
During our analysis of the changes, we observed that the metadata of several folders had been altered. This alteration was likely not caused by Gedit but rather by the operating system. This hypothesis is supported by the fact that other traces than these were discovered when the test case was repeated. Nevertheless, these traces are part of the changes made during the execution of the test case. Table 4.17 provides a chronological sequence of the metadata changes. A more detailed version of this table can be found in Appendix E, Table E.7.

Table 4.16.: Timeline of the Metadata changes during the Logfiles Gedit test case

Time	**Modified Timestamp**	**File/Directory**
16:44	**Accessed**	root, bin (Symlink), lib (Symlink), lib32 (Symlink), lib64 (Symlink), sbin (Symlink), libx32 (Symlink), /var/, /var/log/, Syslog (old)
16:56	**Accessed**	var/log/syslog.1, var/log/kern.log.1, var/log/dmesg.0, var/log/dmesg, var/log/auth.log.1, var/log/vboxadd-setup.log.1, var/log/faillog, var/log/lastlog, var/log/wtmp, Syslog (New)
	Modifed, **Changed**	Syslog (Old), Syslog (New)
	Deleted	Syslog(Old)
	Created	Syslog (New)

New Inode:
As previously mentioned, the Syslog file utilizes a different inode in the modified disk image. The reason for this lies in the middleware library employed by the text editor. As shown by Thierry and Müller [24], Gedit utilizes the GTK[24] middleware library, which exhibits the observed behavior. This middleware library saves a modified file by creating a new file with the same name, writing the altered data there, and then deleting the original file.

Unchanged Data:
Not only can the altered data leave traces for a forensic investigator, but the unchanged data can also hold traces. During the evaluation of this test case, it was noted that the blocks storing the original file had not been touched, and thus, the original file still exists in the disk storage and can be reconstructed.

Verdict

The Gedit text editor can complete the test case and successfully modify the Syslog file. However, creating a new Syslog file generates numerous traces, particularly given that the previous file's contents remain in memory. Additionally, metadata from several other folders was altered during the test case's execution. These changes were most likely created by the operating system or other processes, which represents one of the inherent risks associated with working on mounted disk images.

4.5.2. Sublime Text

Sublime Text is another lightweight text editor that can be modified with plugins. We are using an unmodified version of Sublime Text for this test case. The decision to test another text editor was motivated by the observation that this text editor exhibits a different behavior than Gedit concerning its file-saving process, as previously described by Niclas Pohl [18]. In contrast to Gedit, which creates a new file when a file is saved, Sublime Text retains the file and only allocates a new memory area for the data.

Test Case Satisfiability

As the Syslog file utilized by the disk image is a text file, Sublime Text can execute the test case without modifying the parameters further.

Test Case Execution

Given the analogous execution process of the *Logfile-Sublime Text* test case to that of Gedit, we direct the reader to Section 4.5.1 at this point. Notably, the inode number has remained the same upon examination of the test case result. Furthermore, we have observed that the hash sums of the modified Syslogfiles of the Gedit and Sublime Text test cases match, indicating that the contents of the two files are identical.

[24] `https://www.gtk.org/` (*Accessed On: 21.05.2024*)

Analysis

A comparison of the base and modified disk image revealed that 1591386 bytes in 418 blocks had changed. An overview of the modified blocks and their usage can be found in Appendix E, Table E.6. The following passage will examine some of the more notable changes.

New Syslog File Contents (Blocks 34048-34236):
A comparison of the istat output of the base image Syslog file with that of the modified Syslog file revealed that the block ranges used to store data blocks have changed. In the base image, the file contents were stored in blocks 1095681 to 1095713 in block group 33 and 1059563 to 1059736 in block group 32. The modified disk image now contains the new file contents in blocks 34048 to 34236 of block group 1.

Sublime Text Buffer (Blocks 34304-34492):
The data blocks in question belong to block group 1 and have been marked as unused in both the base and modified images. Further investigation revealed that Sublime Text used these blocks as a buffer for the Syslog file. It was observed that Sublime Text did not erase these traces after saving the file.

File and Directory Metadata Changes:
As observed in the Gedit test case, the metadata of the Syslog file was not the sole element affected; other folders were also modified. Table 4.17 presents a simplified sequence of these changes. The full timeline can be found in Appendix E, Table E.7.

Table 4.17.: Timeline of the Metadata changes during the Logfiles Sublime testcase

Time	Action	File/Directory
17:42	Accessed	root, /var, /var/log, Syslog
17:44	Modifed, Changed	Syslog

Unchanged Data:
As in the Gedit test case, the original data of the Syslog file can be found in the memory in the original memory blocks. These blocks could be extracted as part of a forensic investigation and serve as the basis for proving the manipulation.

Verdict

The text editor Sublime Text is capable of modifying the Syslog file. However, this process leaves numerous traces, including the original file data and a write buffer with the contents of the file. Given these considerations, we advise against utilizing this approach in our context.

4.5.3. DiskForge

The DiskForge tool is the framework we developed during this thesis. We used the Logfiles module with the LogDeltaShift functionality for the Logfile test case.

Test Case Satisfiability

The DiskForge tool can access and modify the Syslog file to run the test case. This means we can run the test case without having to change its parameters.

Test Case Execution

As previously stated, the disk image does not require mounting for DiskForge. Consequently, the test case was initiated without any further preparation. Listing 4.17 shows the command employed to execute the test case. Notably, the hash value of the modified Syslog file does not correspond to the values of the other two test cases. The reason for this observation is further discussed in Section 4.5.4.

Listing 4.17: DiskForge command utilized for the Logfile Manipulation Test Case and description of the parameters.
Note: Line breaks inserted to enhance readability

```
python3 ./TimeModNew.py --func timestomp --target logfile --mode shift
 --img <Path to Image>  --file <Path to File> --partnum 6
 --mess SMBIOS --shift_type minute --shift_time 12 --log_type Syslog
```

The parameters are explained in the following:

- *func*: Functionality we want to use
- *target*: The type of data we want to manipulate
- *mode*: How we want to manipulate the data
- *img*: Path to the disk image file
- *file*: File we want to manipulate
- *partnum*: Partition on which the file resides
- *–mess*: For which message we want to filter
- *–shift_type*: What unit should be shifted (day, hour, minute, second)
- *–shift_time*: How many *shift_type* units we want to shift
- *–log_type*: What type of logfile we want to manipulate

Analysis

The comparison between the base image and the disk image manipulated by DiskForge revealed 532959 changed bytes in 138 blocks. An overview of the changed blocks and their usage can be found in Appendix E, Table E.8.

The modified blocks can be divided into two groups. The first comprises the blocks from 1059591 to 1059664, while the second encompasses the blocks from 1059665 to 1059728. Both groups will be discussed in the following section.

Syslog File Contents (Blocks 1059591 - 1059664):
The blocks in question have undergone a series of alterations. The first event containing the keyword '*SMBIOS*'[25], situated at line *1326*, has been relocated and reinserted at line *3506*. Consequently, all bytes between the original and new positions have been shifted forward. For further clarification, refer to Appendix E, Figure E.7 and E.8, which provides an annotated hexdump illustrating this shift in precise detail.

Syslog File Contents(Blocks 1059665-1059728):
Blocks 1059665 to 1059728 contain the same shift traces as blocks 1059591 to 1059664, except that the second event (See Listing 4.12) is the cause this time.

[25] See Listing 4.12

Verdict

The DiskForge approach modifies a minimal number of bytes, specifically those within the file contents of the Syslog file that have undergone change or relocation in order to fulfill the test case. As only the bytes of the Syslog file's contents were altered, no modifications were made to other data structures of the disk image, thus preventing the generation of external traces.

4.5.4. Comparison

All three approaches were able to complete the test case. However, we noticed some significant differences when we analyzed the range of changes made to the disk image.

	Changed Bytes (absolut)	Changed Bytes (percent)	Changed Blocks
Gedit	839757	52.77%	254
Sublime Text	1591386	100%	418
DiskForge	736213	46.26%	189

Table 4.18.: Comparison of the amount of modified data between the Gedit, SublimeText, and DiskForge approach.
Note: 100% is equal to the number of the most changed bytes (absolut).
The Tool with the lowest number of changed bytes is marked in green.

As Table 4.18 illustrates, the Sublime Text test case resulted in the most significant number of byte changes. This is primarily due to Sublime Text creating a buffer with disk image memory modifications. In contrast, the Gedit and DiskForge test cases exhibited smaller data alterations compared to Sublime Text and notable similarities. The relocation of text lines containing events resulted in the movement of numerous lines by one line. Consequently, all the moved lines are now positioned further forward in the memory and are therefore displayed as having changed. Next, we compared the checksums of the resulting Syslog files, as seen in Listing 4.18.

Listing 4.18: Comparison of the Syslog Hash values of the different test cases
Equivalent Values are colored the same

```
Gedit:
aef7aa02207c1ed48af5cac0fe8bc9d69f1af4a3dceb584887dfa5b9a837324a
Sublime Text:
aef7aa02207c1ed48af5cac0fe8bc9d69f1af4a3dceb584887dfa5b9a837324a
DiskForge:
ebc6d40ab2f51946d2d606fdcaed5fdf2b777d0dd45e50a2851af5993553f1f3
```

As can be seen in Listing 4.18, the hash values of the Gedit and Sublime Text test case Syslogfiles match, but the DiskForge Syslog file does not. To determine what was different, we compared the Sublime Text test Syslog file to the DiskForge test Syslog file. As shown in Listing 4.19, the Gedit and Sublime Text approaches have two entries that are not in the correct position. The DiskForge approach resorted all the entries, placing the entry in the correct position. We looked at the base Syslog file to verify that we did not make a mistake while running these test cases. Here, the two entries are also not in the correct position. This explains the difference between the approaches' Syslog files. Finally, we compared the other data that was modified during the execution of the test case. The DiskForge approach only modified the data blocks of the Syslog file and neither accessed nor modified any other data. The Sublime Text test case modified the accessed timestamps of the root, /var, and /var/log directories. In addition to the directories

that were accessed by the Sublime Text test case, the Gedit test case also accessed various library and binary directories. In addition, the Gedit and Sublime Text left the original file contents in the disk image blocks and marked them as unallocated. On the contrary, DiskForge modified the file in place and overwrote the original file contents with the new ones.

Listing 4.19: The differing entries of the Syslog file of the Gedit and DiskForge test case: **Entry in the SublimeText Syslog File**, **Entry in the DiskForge SyslogFile**

```
First Diff:
 Jun 18 22:52:59 kassensystem-orlandoerstube tseconnector.bat[770]: TSE not mounted yet - check
                 again in some seconds...
+Jun 18 22:53:09 kassensystem-orlandoerstube tseconnector.bat[770]: TSE not mounted yet - check
                 again in some seconds...
 Jun 18 22:53:15 kassensystem-orlandoerstube kernel: [  218.272311] audit: type=1400 audit
                 (1687121595.710:56): apparmor="DENIED" operation="open" class="file" profile=
                 "snap.firefox.firefox" name="/etc/fstab" pid=2682 comm="firefox" requested_mask=
                 "r" denied_mask="r" fsuid=1000 ouid=0
-Jun 18 22:53:09 kassensystem-orlandoerstube tseconnector.bat[770]: TSE not mounted yet - check
                 again in some seconds...
 Jun 18 22:53:15 kassensystem-orlandoerstube dbus-daemon[696]: [system] Activating via systemd:
                 service name='org.freedesktop.hostname1' unit='dbus-org.freedesktop.hostname1.
                 service' requested by ':1.129' (uid=1000 pid=2682 comm="/snap/firefox/2356/usr/
                 lib/firefox/firefox " label="snap.firefox.firefox (enforce)")

Second Diff:
 Jun 19 16:13:48 kassensystem-orlandoerstube gnome-shell[2010]: Window manager warning: Ping serial
                 77907 was reused for window W1, previous use was for window W2.
+Jun 19 16:13:48 kassensystem-orlandoerstube gnome-shell[2010]: Window manager warning: Ping serial
                 77907 was reused for window W1, previous use was for window W2.
 Jun 19 16:13:49 kassensystem-orlandoerstube kernel: [   78.761848] usb 1-1: new high-speed USB
                 device number 2 using ehci-pci
 ...
 Jun 19 16:13:49 kassensystem-orlandoerstube kernel: [   79.139880] usb 1-1: SerialNumber:
                 1101331689514DA4
-Jun 19 16:13:48 kassensystem-orlandoerstube gnome-shell[2010]: Window manager warning: Ping serial
                 77907 was reused for window W1, previous use was for window W2.
 Jun 19 16:13:49 kassensystem-orlandoerstube mtp-probe: checking bus 1, device 2: "/sys/devices/
                 pci0000:00/0000:00:0b.0/usb1/1-1"
```

4.6. Database Manipulation

The Database Manipulation test case employs the Firefox browser's places.sqlite[26] file. The objective is to modify the *last_visit_date* timestamp in the *moz_places* table of all entries containing the '*tagesschau*' keyword in their *url* field by '+12 minutes'. Table 4.19 illustrates the relevant entries. The subsequent sections will examine the SQLiteBrowser and the DiskForge framework.

ID	URL	Visit Count	Last Visit Date
8	http://tagesschau.de/	1	**686353552894655**
9	https://www.tagesschau.de/	1	**1686353552976993**
10	https://www.tagesschau.de/ausland/amerika/usa-trump-anklageschrift-100.html	1	**1686353589205587**
11	https://www.tagesschau.de/ausland/europa/grossbritannien-johnson-ruecktritt-100.html	1	**1686353971060872**
12	https://www.tagesschau.de/inland/heizungsgesetz-scholz-habeck-100.html	1	**1686354099879706**

Table 4.19.: The entries in the unmodified places.sqlite file which match our search criteria '*Tagesschau*'. The unmodified values are marked in **green**.

ID	URL	Visit Count	Last Visit Date
8	http://tagesschau.de/	1	**1686353552894655**
9	https://www.tagesschau.de/	1	**1686353552976993**
10	https://www.tagesschau.de/ausland/amerika/usa-trump-anklageschrift-100.html	1	**1686353589205587**
11	https://www.tagesschau.de/ausland/europa/grossbritannien-johnson-ruecktritt-100.html	1	**1686353971060872**
12	https://www.tagesschau.de/inland/heizungsgesetz-scholz-habeck-100.html	1	**1686354099879706**

Table 4.20.: The entries in the modified places.sqlite file which match our search criteria '*Tagesschau*'. The values in the *Last Visit Date* column have been changed in according to the test case instructions and are marked in **orange**.

4.6.1. SQLiteBrowser

For this approach, we selected the tool SQLiteBrowser[27], an open-source database browser. This software enables the user to analyze the database in a graphical interface and modify entries.

Satisfyability

The tool can access and modify the database file for this test case. Therefore, we do not need to change the test case parameters, and the test can run unchanged.

[26] Full Path: home/kassensystem/snap/firefox/common/.mozilla/firefox/ij9d8zia.default/places.sqlite
[27] `https://sqlitebrowser.org/` (*Accessed On: 18.05.2024*)

Execution

First, the disk image was mounted as described in Section 4.3.1. Then, the SQLiteBrowser was started. Within the GUI, we navigated to the table *moz_places* and filtered the *url* column for the keyword '*tagesschau*', which returned the entries in Figure 4.19. Subsequently, the timestamps of the *last_visit_date* were extracted and converted into a human-readable format using the EpochConverter[28]. The converted timestamps were then shifted by 12 minutes and converted back to the timestamp format of the last_visit_date column. These new values were inserted into the respective fields, and the database was saved. An illustration of the process can be found in Appendix E, Table E.9.

Analysis

A comparison of the base and modified disk image revealed that 168485 bytes in 72 blocks have changed. Appendix E, Table E.10, provides an overview of the modified blocks and their usage. Table 4.21 presents a timeline of the modified metadata, with a higher time precision available in Appendix E, Table E.11. This indicates that two inodes were created and subsequently deleted. Unfortunately, the use of these inodes cannot be further analyzed. Furthermore, the Accessed timestamp of the places.sqlite file and the Modified and Changed timestamps of the file and its parent directories have been altered. In the following section, we will examine a few selected blocks in greater detail.

Time	Action	File/Directory
15:56	Accessed	places.sqlite, inode: 1311745, inode: 1311842
	Created	inode: 1311745, inode: 1311842
16:00	Modified, Changed	places.sqlite, firefox/ij9d8zia.default/, inode: 1311745, inode: 1311842
	Deleted	inode: 1311745, inode: 1311842

Table 4.21.: Timeline of the Metadata changes during the Databases SQLiteBrowser testcase

Fragments of places.sqlite (Blocks 12229 - 12244):
It should be noted that - according to *blkstat*[29] - these blocks are allocated neither in the unmodified nor the modified disk image. They belong to group 0 and are part of the data blocks of the group. Upon examination of the blocks, it became evident that portions of the file contents of the places.sqlite file could be found here.

places.sqlite file contents (Blocks 5162245, 5162430, 5162437):
These three blocks contain the modifications to the file contents of the places.sqlite file. Blocks 5162245 and 5162437 contain the alterations to the last_visit_date timestamps of the individual events. Annotated hexdumps of these blocks can be found in Appendix E, Table E.9 and E.10. In addition, 8 bytes were altered in block 5162430, the purpose of which remains unclear. An annotated hexdump for this block can be found in Appendix E, Table E.11.

Zeroed Blocks (Blocks 5319680 - 5319687):
These blocks are neither allocated in the unmodified nor the modified disk image. They belong

[28] `https://www.epochconverter.com/` (*Accessed On: 18.05.2024*)
[29] `http://www.sleuthkit.org/sleuthkit/man/blkstat.html` (*Accessed On: 18.05.2024*)

to group 162 and are part of the data blocks of the group. When examining the blocks, we noticed that the contents of these blocks were overwritten with zero bytes in the modified disk image. Unfortunately, it was not possible to find out what kind of data was originally stored there, as the analysis of the hexdump revealed no information.

Verdict

The SQLiteBrowser approach successfully modified the places.sqlite file. The approach's most extensive trace is the temporary buffer created in blocks 12229 to 12244 and the updated metadata values.

4.6.2. DiskForge

We used the *Database* module of DiskForge with the *ShiftDB* functionality for the Database test case.

Satisfyability

The DiskForge tool can access and modify the places.sqlite file to complete the test case. So, we ran the test case without changing the parameters.

Execution

Since the DiskForge tool works on unmounted disk images, we started directly with the test case and constructed the command shown in Listing 4.20 to execute the test case. The parameters of the DiskForge command are found and explained in Listing 4.20. The execution of the test case resulted in the 'Error' message shown in Listing 4.21. The cause of this error is that the *places.sqlite* file of the disk image references block number 0 as a data block. Since this block is not a valid address for storing file data, we implemented a fail-safe mechanism that prevents the superblock from being overwritten. In this case, the algorithm checks if any 'relevant' information would come after the end of the available space and returns the warning shown in Listing 4.21. A more in-depth explanation can be found in Section 3.4.2.

Analysis

The comparison between the base image and the DiskForge manipulated disk image resulted in 42 changed bytes in 3 blocks. An overview of the changed blocks and their usage can be found in Appendix E, Table E.12.

places.sqlite File Contents (Blocks 5162245, 5162430 & 5162437):
The block changes match those found during the Database - SqliteBrowser test case in Section 4.6.1.

Verdict

The DiskForge approach uses the same number of bytes as the Databases-SQLiteBrowser test case to alter the data blocks in the places.sqlite file. However, it only modifies these blocks and

Listing 4.20: DiskForge command utilized for the Database Manipulation Test Case and description of the parameters.
Note: Line breaks inserted to enhance readability

```
python3 ./TimeModNew.py --func timestomp --target database --mode shift --db_name firefox
 --img <Path to Disk Image> --file <Path to File> --partnum 6
 --url tagesschau --shift_type minute --shift_time 12
```

The parameters are explained in the following:

- *func*: Functionality we want to use
- *target*: The type of data we want to manipulate
- *mode*: How we want to manipulate the data
- *img*: Path to the disk image file
- *file*: File we want to manipulate
- *partnum*: Partition on which the file resides
- *db_name:* Database name/type
- *shift_type:* What unit should be shifted (day, hour, minute, second)
- *shift_time:* How many shift_type units we want to shift
- *url:* For which URL should be filtered

Listing 4.21: Error returned by DiskForge

```
Error the file you wanted to write was to large
But nothing broke cause there were only Zero Bytes
```

not any other files, directories or metadata. Thus, it leaves fewer traces than the other test case.

4.6.3. Comparison

Both approaches were successful in completing the test case without any modifications. However, upon analysis of the results in Table 4.22, it became evident that there were enormous differences in the number of altered bytes. We can see that the SQLiteBrowser approach changed 4153.76 times more bytes than the DiskForge tool. Next, we compared the places.sqlite files created by both approaches.As Listing 4.22 shows, the *places.sqlite* file of both approaches match. All other changes the SQLiteBrowser approach makes are potential traces for a forensic investigator, which could hint at the manipulation. The SQLiteBrowser did not only modify the file but also stored a buffer inside the disk image. Furthermore, the tool (partially) zeroed 8 blocks.

In conclusion, the DiskForge approach is better suited for our use case because it leaves fewer traces and modifies only the file contents and no other data on the disk image.

	Changed Bytes (absolut)	Changed Bytes (percent)	Changed Blocks
SQLiteBrowser	174458	100%	75
DiskForge	42	0.024%	3

Table 4.22.: Comparison of the amount of modified data between the SQLiteBrowser and DiskForge approach.
Note: 100% equals the number of the most changed bytes (absolute).
The Tool with the lowest number of changed bytes is marked in green.

Listing 4.22: Comparison of the places.sqlite hash values of the different test cases.
Same values are colored the same.

```
SQLiteBrowser:
f87e09b7ca038033176a4d5a3a183e7d69a2a9fec9ed367819b476f531b78dd5
DiskForge:
f87e09b7ca038033176a4d5a3a183e7d69a2a9fec9ed367819b476f531b78dd5
```

5 General Timestomping Traces

As seen in Section 4, the DiskForge tool leaves no explicit traces during our test cases. Therefore, this section will explore some possible traces that timestomping approaches could leave, although some do not apply to our tool. The section is organized as follows: First, we will look at traces that could result from metadata changes and then at logfiles and databases. The following aspects were determined during the framework creation process, in addition to those mentioned in relevant scientific papers, e.g., Wicher Minnaard [16].

5.1. Metadata

We will start with the traces created by manipulating file metadata.

5.1.1. Checksums

Most structures in ext4, such as inodes, have checksums stored in them to verify the correctness of the structure's contents. When modifying the metadata of files, it is crucial to understand how to modify the checksum correctly. Ext4 uses a modified crc32c algorithm in many places. Despite what the documentation states, the stored value is not the output of the crc32c algorithm but the result of the following equation:

$$\text{Checksum} = \text{0xFFFFFFFF} - \text{crc32c}\,(\text{UUID+Inode\_Num+Generation+Inode}) \tag{5.1}$$

In order to prevent any residual traces of inode manipulation, the script generate_crc32c.py[1] was developed for DiskForge to update the inode checksums following a successful manipulation. However, it should be noted that checksums are not solely employed in ext4 inodes. Indeed, crc32c checksums are also utilized in ext4 superblocks and bitmaps, among other structures. For a comprehensive overview of the checksummed structures, refer to the ext4 wiki[2]. Remembering the checksums when making manual changes to these structures is of the utmost importance to avoid leaving any unnecessary traces.

5.1.2. Sub-Second Precision

As illustrated in Section 4.4.1, tools such as Touch set the microseconds field to 0 when modifying the Accessed and Modified timestamps. In reality, this value is rarely naturally generated in this field. Based on these timestamps set to 0, it can be hypothesized that they have been manipulated. DiskForge allows the user to set the microsecond fields as desired to circumvent this issue. In the absence of user-specified data for the microseconds, the original microseconds of the timestamp are retained, thus giving the appearance of organic timestamps.

5.2. Mismatching Timestamps of Files and their Parent Folders

Inspired by the work of Wicher Minnaard [16], we analyzed the correlation between the timestamps of a file and its parent directory. We found a possible clue when comparing the timestamps of a folder and a timestamped file. Something has most likely been manipulated if the file's Changed value is older than the folder's Created timestamp. The Created, Modified, and Accessed timestamps can be older than the folder's Created timestamp because the file could have been moved into the folder and existed before the folder was created. However, when the file is moved to the folder, the file's Changed timestamp is updated. Therefore, a file's Changed timestamp cannot be older than the directories Created timestamp.

5.3. Log Files

Next, traces created by manipulating file metadata are discussed.

5.3.1. Extra Time Information in Entries

When modifying the timestamps of a log file entry, we usually only need to modify the timestamp located at the beginning of the event. However, the message body may contain a timestamp corresponding to the primary timestamp. If we only change the leading timestamp, the timestamp in the message will still point to the original time.

In the case of the Syslog file, we found the keyword 'audit' followed by a timestamp in Unix time, as seen in the Listing 5.1. Listing 5.2 provides an additional example of implicit temporal data, as found in the *apt/history.log* file.

[1] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/Checksumming/generate_crc32c.py` (*Accessed On: 18.05.2024*)

[2] `https://ext4.wiki.kernel.org/index.php/Ext4_Metadata_Checksums` (*Accessed On: 18.05.2024*)

Listing 5.1: Excerpt from a Syslog file showing the extra timestamps highlighted in green

```
Feb 11 00:00:20 Niclas-Rechner kernel: [42360.174574] audit: type=1400 audit(
    1707606020.557:1687430): apparmor="DENIED" ...
Apr 14 07:05:42 Niclas-Rechner NetworkManager[1377]: <info> [1713071142.8193] dhcp4 (wlp3s0): state
     changed new lease, address=192.168.178.26
```

Listing 5.2: Excerpt from a apt/history.log file showing the implicit time information in green

```
Start-Date: 2024-04-05 06:13:56
Commandline: /usr/bin/unattended-upgrade
Upgrade: xserver-xorg-core:amd64 (2:21.1.4-2ubuntu1.7~22.04.8, 2:21.1.4-2ubuntu1.7~22.04.9)
End-Date: 2024-04-05 06:13:57
```

As seen in Figure 5.2, the history.log entries contain not one but two timestamps, one for the start and one for the end of the event, respectively. Changing both the *Start-Date* and the *End-Date* and keeping the difference between these two timestamps consistent is important. DiskForge solves this problem in the following way: We calculate the difference between the *Start-Date* and the *End-Date*. When changing the timestamps, we set the *Start-Date*, and for the *End-Date*, we set the *Start-Date* plus the difference.

5.3.2. Chronological Order of Events

The log files analyzed in this thesis all follow the same ascending chronological order. The oldest event is at the beginning of the logfile, and the newest is at the end. This pattern is produced by the fact that new events are appended to the existing log file. If the timestamp of an event is changed exclusively, the chronological order of the file may be violated. Therefore, sorting the events after a successful manipulation is essential to eliminate these traces. DiskForge has implemented the less-than operator for the different log types[3] to be able to sort the events after the manipulation chronologically.

It should be noted that during our evaluation in Section 4.5.4, we found two events not in the right places in the unmodified log file of the base image file. Currently, it is unclear what the cause of this behavior is. One possible explanation is that the process submitted its event some seconds after it was created. Nevertheless, this behavior should be the subject of further research.

5.3.3. Events in Wrong Log File

As mentioned above, Ubuntu usually does not store a large log file; instead, it splits the log files into multiple parts. For example, a system may have a 'syslog' file, a 'syslog.1' file, and a 'syslog.2.gz' file. In this case, the former contains the most recent entries, the 'syslog.1' file contains older entries, and the latter contains the oldest. If an event's timestamp changes significantly, it may happen that the event is now in the correct chronological place in its log file but in the wrong log file.

DiskForge cannot resolve this particular issue. However, due to the framework's extensibility, this functionality can be incorporated in the future. One potential approach would be to parse

[3] An example for this feature can be seen on GitLab:: `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge/-/blob/main/DiskForge/Utility/LogEntryStructures/sys_log.py` (*Accessed On: 18.05.2024*)

all log files of a specific type and identify the log file in which the event fits. The event can then be inserted into the new file, and the lengths of the original and new files can be adjusted as necessary. Furthermore, new blocks must be allocated if necessary, unused blocks should be deallocated, and checksums must be updated.

5.3.4. Related Entries

The last type of traces we found in logfiles are related entries. Listing 5.3 shows the occurrence of such entries in the Syslog file. The events in the log file belong together because they are all part of a crash report. If individual events were moved to other places by timestomping, a trace of the manipulation of the log file would be recognizable from the context of the events.

At present, DiskForge cannot recognize the context of log file events. For future work in this area, an approach in the direction of machine learning similar to the approach described by Alji et al. [1] could be used to understand the context of log entries and thus move related events together.

Listing 5.3: Excerpt from a Syslog file showing the crash report of VirtualBox. Here, all the events are related to one another, as the same crash created them and contain information of the crash report.

```
Apr 7 08:45:22 Niclas-Rechner systemd[1]: Started Tool to automatically collect and submit kernel
               crash signatures.
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.878876] vboxdrv: loading out-of-tree module taints
               kernel.
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.878882] vboxdrv: module verification failed: signature
               and/or required key missing - tainting kernel
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895046] vboxdrv: Found 4 processor cores
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895247]
    ================================================================================
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895251] UBSAN: array-index-out-of-bounds in /var/lib/
               dkms/virtualbox/6.1.50/build/vboxdrv/common/log/log.c:1728:38
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895255] index 1 is out of range for type 'uint32_t [1]'
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895258] CPU: 1 PID: 1891 Comm: modprobe Tainted: G OE
               6.5.0-26-generic #26~22.04.1-Ubuntu
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895262] Hardware name: ASUSTeK COMPUTER INC. X756UXK/
               X756UXK, BIOS X756UXK.311 09/19/2017
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895264] Call Trace:
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895270] <TASK>
Apr 7 08:45:22 Niclas-Rechner kernel: [ 18.895272] dump_stack_lvl+0x48/0x70
```

5.4. Databases

Lastly, we focus on traces in databases and, in particular, the topic of inconsistent entries. Upon examination of the *places.sqlite* file associated with the Firefox browser, it was determined that the *id* field serves as the primary key for the *moz_places* table. A primary key is a unique identifier for an entry in a database, implying that no other entry with the same primary key exists. Our testing revealed that revisiting a website does not alter the *id* of the entry but only the *last_visit_date* timestamp and the *visit_count* of the entry. However, although the date of the first website access is not saved, inconsistencies can occur, which will be explained below. The two events, *Event 1* with *id 1* and *Event 2* with *id 2*, serve as examples for this explanation.

In the first example in Table 5.1, we can deduce from the given information that *Event 1* was visited for the first time previous to *Event 2*, as the *id* of *Event 1* is lower. The higher

	id	visit_count	last_visit_date
Event 1	1	2	15
Event 2	2	3	12

Table 5.1.: Example extract of places.sqlite without conflicting implicit time information

last_visit_date timestamp of *Event 1* compared to *Event 2* is no problem in this context. Therefore, we assume that such a construction of events can occur in this way.

	id	visit_count	last_visit_date
Event 1	1	1	15
Event 2	2	1	12

Table 5.2.: Example extract of places.sqlite without conflicting implicit time information

However, the situation is different for the data in Table 5.2. Again, *Events 1* and *2* have the same *id* and *last_visit_date* timestamp as in Table 5.1, but their *visit_counts* are now different. In this example, both websites were only accessed once. This means that the time of the *last_visit_date* and the time of the first access are equivalent. However, the comparison of the timestamps of the *last_visit_date* indicates that *Event 2* was accessed before *Event 1*. This is contradicted by the fact that *Event 1* has a higher *id* than *Event 2*, which indicates that *Event 1* was accessed first. This contradiction suggests an inconsistency between those events, which may hint at database manipulation. Another potential explanation for the discrepancy could be that the user has altered the system time.

At this time, DiskForge is unable to detect such discrepancies. However, due to the framework's high extensibility, this functionality can be added in the future, allowing users to be warned when their changes would result in discrepancies.

6

Limitations & Future Work

Due to both the scope of the thesis and ethical concerns, the DiskForge tool has some limitations. We first want to review the ethical considerations we had taken into account when implementing DiskForge and then discuss some of the tool's limitations due to time constraints. Some aspects were already addressed in Chapter 5, and for the sake of completeness, they will be mentioned only briefly below.

6.1. Detectability of DiskForge due to Ethical Considerations

As evidenced by the evaluation section, DiskForge generates virtually no traces. The traces that indicate its use are mainly those based on implicit time information. Our tool's intended use case is manipulating disk images to teach digital forensics. However, since the tool is publicly available, it must be noted that it can also be used for other purposes that do not correspond to our use case. Consequently, we have determined that limiting the non-detectability of our framework is necessary. To this end, we have implemented a simple watermarking algorithm that sets the last four bytes of block 0, i.e., a portion of the buffer following the superblock, to the message '*DiskForge was here: <CurrentDate>*'. It is important to note that a skilled user can easily circumvent this watermarking algorithm. Therefore, we recommend implementing a more sophisticated watermarking approach in the future.

6.2. Scope Limitations

The project's scope was limited as it was developed as part of a master's thesis. We aimed to develop an extensible disk image manipulation tool that works on unmounted disk images. The

goal was successfully achieved, and we implemented functionality for further expansion, such as extracting and writing back files.

Some of the limitations of our tool will be discussed in the following section.

6.2.1. General

We first discuss the general limitations of our framework.

File Length Extension

The write-back functionality is currently limited to files of equal or shorter length or larger files that do not exceed the last allocated block of the file. This limitation was imposed because the allocation of blocks and the resulting necessary changes to various parts of the file system metadata were beyond the scope of this work. If a write operation exceeds the allocated blocks, our tool checks whether data other than zero bytes would be lost. The framework returns a warning and stops the writing process depending on the result. This is done to ensure the integrity of the file system.

More Checksumms

DiskForge has implemented an algorithm to adjust the checksums of an inode, which is sufficient for the current implementation status, to the best of our knowledge. However, as soon as one wishes to allocate or deallocate blocks, one requires the checksumming algorithms for further structures. These are not yet implemented in DiskForge and must, therefore, be implemented so that no traces are left behind during manipulation.

Unknown file encoding

We ran into a problem when modifying files, as there is no sure way to determine the encoding used for a raw text file. While this may not be relevant in most contexts, we experienced problems with special characters in the German language. In particular, we had trouble with differentiating UTF-8 and ISO-8859-1, also known as Latin1, encoded files. The main difference between UTF-8 and Latin1 encoding is that UTF-8[1] uses two bytes to encode characters, such as ‘ä’, ‘ö’, ‘ü’, while Latin1[2] uses only one byte for all characters. This can cause problems when characters encoded in one format are interpreted as the other.

For example, the character ‘ü’ is represented as 0xC3 0xBC in UTF-8, but in Latin1, 0xC3 is interpreted as ‘Ã’, and 0xBC is interpreted as ‘¼’, resulting in an incorrect representation. If the ‘Ü’ character is in Latin1 and is interpreted as UTF-8, it can cause other problems. In Latin1, the hexadecimal representation is 0xDC. However, if we try to print 0xDC in UTF-8, it becomes an unprintable character.

Our solution to mitigate the problem as much as possible is as follows: When opening a file not created during the execution, we check to see if the file breaks UTF-8 encoding. If it does, we set the encoding to ISO-8859-1; otherwise, it will be UTF-8.

[1] A more in-depth explanation on the encoding procedure of UTF8 can be found here: `https://de.wikipedia.org/wiki/UTF-8#Kodierung` (*Accessed On: 18.05.2024*)

[2] The encoding table for Latin1 can be found here: `https://de.wikipedia.org/wiki/ISO_8859-1#ISO/IEC_8859-1` (*Accessed On: 18.05.2024*)

Listing 6.1: Code for checking the encoding

```
def checkIfUTF8(filename):
    command = "iconv -f utf8 {} -t utf8 -o /dev/null".format(filename)
    result = run(command, stdout=PIPE, stderr=PIPE, universal_newlines=True, shell=True)
    if not result.stderr == "":
        return "ISO-8859-1"
    return "UTF-8"
```

As shown in Listing 6.1, we use the iconv[3] command to convert the file from UTF-8 to UTF-8. If the file is UTF-8 encoded, this should work fine. Otherwise, if it fails, our testing has concluded that the most likely encoding is ISO-8859-1. This is not a perfect solution and could fail if other encoding formats were used. However, this approach is sufficient for our use case, as we only encountered UTF-8 and ISO-8859-1 encoded files while exploring the sample disk images. In contrast, while testing other approaches, such as using the Python3 *chardet*[4] library, the execution time increased from a few seconds to more than a minute and in most cases, the results were wrong.

If disk images with logfiles encoded in any other codec are used with our tool, most likely, this algorithm will break, and changes to the *checkIfUTF8* function in *TimeTool/Utility/TerminalCommands.py* are necessary.

Zero Bytes in the File Content

When extracting files with *icat*, the extracted file may contain zero bytes. While zero bytes may have a specific meaning in some file types (e.g. *firefox's places.sqlite* contains zero bytes that - when removed - leave a corrupted file), we assume they serve no purpose in text files as they are not mapped to any printable characters. To avoid problems when parsing logfiles, we removed these zero bytes during the manipulation process. Another solution would be to use the *-h* option of *icat*, but this does not guarantee the removal of zero bytes as shown in our tests. Since we fill the remaining unused memory block with zero bytes, all the zero bytes are now at the end of the file. This means the file will be changed even if we do not manipulate any events. To avoid this, we can write the file back only when a manipulation occurs. If we want to remove these bytes from the file entirely, we need to update the file length in the inode metadata. In the future, it would be advantageous to ascertain the precise origin of these zero bytes and to determine in which instances they possess a special significance in which they do not. Once this information has been obtained, the implementation of DiskForge can be adapted accordingly.

6.2.2. Metadata

Next, limitations concerning metadata manipulation will be discussed.

Year Limitation

As noted in Section 2.3.3, in ext4, the date can be extended using two bits of the extra fields. Since this is not particularly interesting for our use case (as we do not need dates after 2038),

[3] `https://linux.die.net/man/1/iconv` (*Accessed On: 18.05.2024*)
[4] `https://pypi.org/project/chardet/` (*Accessed On: 18.05.2024*)

we decided not to implement this feature due to time constraints. In the future, this feature can be implemented in the same place as the update of the extra field. Referring to the ext4 wiki's layout page[5], the functionality can be added by doing an addition with the values 0 to 3 depending on the timestamp.

Inconsistent Timestamp between Files and Parent Directories

As previously outlined in Section 5.2, discrepancies may arise between files and their parent directories. DiskForge is currently unable to identify these. It is conceivable that in the future, when file timestamps are modified, the timestamps of the folder above will be simultaneously checked to ascertain the presence of inconsistencies and—if necessary—implement corrective measures.

6.2.3. Logfiles

Next, limitations concerning log file manipulation will be discussed.

Logfile Manipulation over more than one file

Our approach to logfile manipulation sorts the event to the correct location in the current file. Since the operating system uses not just one file but multiple files to store even older events, an event may belong to a different file. We need functionality to allocate blocks to do this efficiently since moving content from one file to another would change the file size. If the functionality to allocate space is added, the easiest way to implement multi-file logfile timestomping would be to extract the events of all files and note the number of events in each file. Next, the events should be manipulated, and then all the events should be sorted chronologically. These sorted events should be reinserted into the files according to the number of events stored in each file. Finally, when writing back the files, if the file size has changed, the inode file size should be updated and blocks allocated if necessary.

Related Events

As illustrated in Listing 5.3, certain events within a log file may be contiguous. Currently, DiskForge cannot recognize these events as contiguous, which results in the generation of traces during manipulation. It is potentially feasible to employ a machine learning approach to identify such contiguous events and notify the user that their proposed change will result in the creation of traces, with the option of moving the entire event block.

6.2.4. Databases

At last, limitations concerning databse manipulation will be discussed.

[5] `https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout` *Accessed On 22.05.2024*

Inconsistent IDs

As demonstrated in Section 5.4, the combination of the *last_visit_date* and the *visit_count* values can be employed to ascertain the time of the first visit and verify the ID's logic. The current implementation of DiskForge is unable to identify and rectify inconsistencies that may arise due to this phenomenon. In the future, it may be feasible to identify the affected entries and - if necessary - modify the event IDs to restore chronological order.

Inconsistent Tables

In addition, it is possible that not only the IDs can be used to conclude but also that other tables in the database contain information that contradicts the manipulations. DiskForge is also unable to recognize this. In future work, it would be necessary to examine the databases used more closely to see whether contradictory values can be found there. Then, change these as part of the manipulation so that the values are consistent between the tables.

7 Conclusion

This thesis investigated the possibility of manipulating temporal data on unmounted disk images. During this work, we created an extensible ext4 disk manipulation framework capable of modifying file metadata, log files, and databases without leaving any explicit traces. Finally, this chapter summarizes the results of the thesis and possible future work in this area.

The DiskForge tool has fulfilled all the objectives initially set for this thesis in Chapter 1. It has been developed as a means of efficiently manipulating unmounted disk images. The timestomping functionalities have been accurately implemented by the tool, and according to our analysis, this approach leaves far fewer traces than commonly used tools. In contrast to other approaches, which create a multitude of traces, DiskForge bypasses these changes by modifying the raw disk image data directly.

Realizing that DiskForge leaves minimal traces, we explored the potential of utilizing implicit time information to identify manipulation in Chapter 5. Additionally, we examined the tool's ethical implications in the previous chapter.

This framework was developed as an extensible disk image manipulation tool, leaving room for future work. As the chapter on limitations and future work outlines, the timestomping functionality has only been implemented for specific log files. These can be extended to allow the modification of more logfiles or to manipulate logfiles that are not stored as plain text. Future work could also target other points of interest inside a disk image, such as the content of different files. Finally, the tool can be used to create forgeries, which can be used to train an algorithm for detecting manipulations in ext4 disk images.

As part of this thesis, we provide the framework's source code, documentation, and a dataset of manipulated disk images on Gitlab[1].

[1] `https://faui1-gitlab.cs.fau.de/lena.voigt/diskforge` (*Accessed On: 18.05.2024*)

8

Bibliography

[1] Mohamed Alji and Khalid Chougdali. "Detection of Timestamps Tampering in NTFS using Machine Learning." In: *The 10th International Conference on Emerging Ubiquitous Systems and Pervasive Networks (EUSPN 2019) / The 9th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2019) / Affiliated Workshops, Coimbra, Portugal, November 4-7, 2019*. Ed. by Elhadi M. Shakshuki, Ansar-Ul-Haque Yasar, and Haroon Malik. Vol. 160. Procedia Computer Science. [Accessed 14-04-2024]. Elsevier, 2019, pp. 778–784. DOI: 10.1016/J.PROCS.2019.11.011. URL: https://doi.org/10.1016/j.procs.2019.11.011.

[2] *bsi.bund.de*. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/Lageberichte/Lagebericht2023.pdf?__blob=publicationFile&v=7. [Accessed 18-03-2024]. 2023.

[3] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, 2005. ISBN: 0321268172.

[4] Srivathsa Dara. *Understanding Ext4 Disk Layout, Part 1*. https://blogs.oracle.com/linux/post/understanding-ext4-disk-layout-part-1. [Accessed 22-03-2024]. 2023.

[5] Srivathsa Dara. *Understanding Ext4 Disk Layout, Part 2*. https://blogs.oracle.com/linux/post/understanding-ext4-disk-layout-part-2. [Accessed 22-03-2024]. 2023.

[6] Mark Debinski, Frank Breitinger, and Parvathy Mohan. "Timeline2GUI: A Log2Timeline CSV parser and training scenarios." In: *Digit. Investig.* 28.Supplement (2019), pp. 34–43. DOI: 10.1016/J.DIIN.2018.12.004. URL: https://doi.org/10.1016/j.diin.2018.12.004.

[7] *digital forensics - Glossary | CSRC — csrc.nist.gov*. https://csrc.nist.gov/glossary/term/digital_forensics. [Accessed 12-04-2024].

[8] Lisa Marie Dreier, Céline Vanini, Christopher J. Hargreaves, Frank Breitinger, and Felix Freiling. "Beyond Timestamps: Integrating Implicit Timing Information into Digital Forensic Timelines." accepted at: Proceedings of the Digital Forensics Research Conference USA (DFRWS USA). 2024.

[9] *Ext4 Disk Layout - Ext4 — ext4.wiki.kernel.org*. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout. [Accessed 12-04-2024].

[10] Felix C. Freiling and Leonhard Hösch. "Controlled experiments in digital evidence tampering." In: *Digit. Investig.* 24 Supplement (2018). [Accessed 14-04-2024], S83–S92. DOI: 10.1016/J.DIIN.2018.01.011. URL: https://doi.org/10.1016/j.diin.2018.01.011.

[11] Simson Garfinkel. "Anti-Forensics: Techniques, Detection and Countermeasures." In: *ICIW 2007 2nd International Conference on i-Warfare and Security*. 2007, pp. 77–84. URL: https://simson.net/ref/2007/iciw07-cd.pdf.

[12] Thomas Göbel and Harald Baier. "Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding." In: *Digit. Investig.* 24 Supplement (2018). [Accessed 14-04-2024], S111–S120. DOI: 10.1016/J.DIIN.2018.01.014. URL: https://doi.org/10.1016/j.diin.2018.01.014.

[13] Sebean Hsiung. *Learn more about EXT4*. https://www.datarecoverytools.co.uk/2009/11/16/learn-more-about-ext4/. [Accessed 15-04-2024].

[14] *Indicator Removal: Timestomp*. https://attack.mitre.org/techniques/T1070/006/. [Accessed 14-03-2024].

[15] Anu Jain and Gurpal Singh Chhabra. "Anti-forensics techniques: An analytical review." In: *2014 Seventh International Conference on Contemporary Computing (IC3)*. 2014, pp. 412–418. DOI: 10.1109/IC3.2014.6897209.

[16] Wicher Minnaard. *Timestomping NTFS*. https://www.os3.nl/_media/2013-2014/courses/rp2/p48_report.pdf. [Accessed 21-04-2024]. 2014.

[17] *Phrack Magazine*. http://phrack.org/issues/59/6.html#article. [Accessed 20-04-2024]. 2002.

[18] Niclas Pohl. "Linux Timestamp Forensics on the Application Layer." Available at https://github.com/NiclasPohl/Linux-Timestamp-Forensics-on-the-Application-Layer/blob/70664d6dc9a782efff268b03f6f130aab416b38e/Linux%20Timestamp%20Forensics%20on%20the%20Application%20Layer.pdf. Bachelor's Thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, July 2021.

[19] Hal Pomeranz. *Understanding EXT4 (Part 1): Extents*. https://www.sans.org/blog/understanding-ext4-part-1-extents/. [Accessed 01-02-2024].

[20] Mark Scanlon, Xiaoyu Du, and David Lillis. "EviPlant: An efficient digital forensic challenge creation, manipulation and distribution solution." In: *CoRR* abs/1704.08990 (2017). [Accessed 18-03-2024]. arXiv: 1704.08990. URL: http://arxiv.org/abs/1704.08990.

[21] Joakim Schicht. *GitHub - jschicht/SetMace: Manipulate timestamps on NTFS — github.com*. https://github.com/jschicht/SetMace. [Accessed 12-04-2024]. 2014.

[22] Janine Schneider, Linus Düsel, Benedikt Lorch, Julia Drafz, and Felix C. Freiling. "Prudent design principles for digital tampering experiments." In: *Digit. Investig.* 40.Supplement (2022), p. 301334. DOI: 10.1016/J.FSIDI.2022.301334. URL: https://doi.org/10.1016/j.fsidi.2022.301334.

[23] Janine Schneider, Julian Wolf, and Felix C. Freiling. "Tampering with Digital Evidence is Hard: The Case of Main Memory Images." In: *Digit. Investig.* 32 Supplement (2020), p. 300924. DOI: 10.1016/J.FSIDI.2020.300924. URL: https://doi.org/10.1016/j.fsidi.2020.300924.

[24] Aurélien Thierry and Tilo Müller. "A systematic approach to understanding MACB timestamps on Unix-like systems." In: *Digit. Investig.* 40.Supplement (2022). [Accessed 11-04-2024], p. 301338. DOI: 10.1016/J.FSIDI.2022.301338. URL: https://doi.org/10.1016/j.fsidi.2022.301338.

[25] Denis Yablochkin. *GitHub - DenisNovac/EXT4Utils: Python utilities for reading and searching data in EXT4 — github.com*. https://github.com/DenisNovac/EXT4Utils. [Accessed 26-03-2024]. 2019.

A

Calculations

A.1. Example Inode Offset Calculations

For this example we want to calculate the offset of the inode of the 'var/log/syslog' file.

1. **Extract Inode Number with fls:**
   We first need to figure out, which inode number corresponds to the file name. To do this we utilize fls:

   Listing A.1: Fls output

   ```
   r/r 1184404:    var/log/syslog
   ```

2. **Extract Inodes per Group with fsstat:**
   Next we extract the number of inodes which get stored in each group:

   Listing A.2: Fsstat output

   ```
   Inodes per group: 8192
   ```

3. **Calculate Block Group of Inode:**
   Next we need to calculate the block group of the inode to find out in which inode table we need to search:

$$\begin{aligned} \text{Inode Group} &= \left\lfloor \frac{(\text{fls} - 1)}{\text{Inodes per Group}} \right\rfloor \\ &= \left\lfloor \frac{(1184404 - 1)}{8192} \right\rfloor \\ &= 144 \end{aligned} \tag{A.1}$$

4. **Extract Group 144 Inode Table begin block:**
   Now that we have the group number, we can extract the begin block of the inode table using fsstat:

Listing A.3: Fsstat output for group 144

```
Group: 144:
  Block Group Flags: [INODE_ZEROED, 0x0808]
  Inode Range: 1179649 - 1187840
  Block Range: 4718592 - 4751359
  Layout:
    Data bitmap: 4718592 - 4718592
    Inode bitmap: 4718608 - 4718608
    Inode Table: 4718624 - 4719135
    Data Blocks: 4726816 - 4751359
  Free Inodes: 21 (0%)
  Free Blocks: 2282 (6%)
  Total Directories: 467
  Stored Checksum: 0x6BEF
```

5. **Calculate Offset of Begin Block:**
   Now that we know that the begin block of the inode table 4718624 is, we can calculate its offset:
   a) **Calculate Partition Offset:**
      To calculate the offset of the partition, we need two values from mmls:

Listing A.4: Mmls output

```
GUID Partition Table (EFI)
Offset Sector: 0
Units are in 512-byte sectors

      Slot      Start        End          Length       Description
000:  Meta      0000000000   0000000000   0000000001   Safety Table
001:  -------   0000000000   0000002047   0000002048   Unallocated
002:  Meta      0000000001   0000000001   0000000001   GPT Header
003:  Meta      0000000002   0000000033   0000000032   Partition Table
004:  000       0000002048   0000004095   0000002048
005:  001       0000004096   0001054719   0001050624   EFI System Partition
006:  002       0001054720   0062912511   0061857792
007:  -------   0062912512   0062914559   0000002048   Unallocated
```

      With this values we can use the following formula:

$$\begin{aligned} \text{Offset Partition} &= \text{Begin Partition} \cdot \text{Sector Size} \\ &= 1054720 \cdot 512 \\ &= 540016640 \end{aligned} \tag{A.2}$$

   b) **Calculate Block Offset in Partition:**
      Next we can calculate the offset between the begin of the partition and the block. For this we first need the block size from fsstat:

Listing A.5: Fsstat output

```
Block Size: 4096
```

Next we can utilize the following formula:

$$\begin{aligned} \text{Offset Block in Partition} &= \text{Block Number} \cdot \text{Block Size} \\ &= 4718624 \cdot 4096 \\ &= 19327483904 \end{aligned} \tag{A.3}$$

c) **Calculate Full Block Offset:**
With all this information we can now calculate the full block offset:

$$\begin{aligned} \text{Full Block Offset} &= \text{Offset Partition} + \text{Offset Block in Partition} \\ &= 540016640 + 19327483904 \\ &= 19867500544 \end{aligned} \tag{A.4}$$

6. **Calculate Index of Inode:**
Next we need to calculate the index of the inode inside the inode table:

$$\begin{aligned} \text{Index Inode} &= (\text{Inode Number} - 1) \mod \text{Inodes per Group} \\ &= (1184404 - 1) \mod 8192 \\ &= 4755 \end{aligned} \tag{A.5}$$

7. **Calculate Offset between Inode and Inode Table:**
Now that we have the index, we can calculate the offset between the inode and the inode table:

$$\begin{aligned} \text{Offset Inode to Inode Table} &= \text{Index Inode} \cdot \text{Inode Size} \\ &= 4755 \cdot 256 \\ &= 1217280 \end{aligned} \tag{A.6}$$

8. **Calculate Full Inode Offset:**
Now we have everything the calculate the full offset of the inode:

$$\begin{aligned} \text{Full Offset} &= \text{Full Offset Inode Table} + \text{Offset Inode to Inode Table} \\ &= 19867500544 + 1217280 \\ &= 19868717824 \end{aligned} \tag{A.7}$$

B

Code

B.1. Implementation

In this Appendix Chapter we present the source code corresponding to the Pseudocodes utilized in this thesis.

B.1.1. Writeback

Listing B.1: Modify Binary Function for writing back Timestamps

```
def modify_binary(inode_offset, timestamp_offset, image_location, what_timestamp,
                  timestamps_original, verbose=False, time=None):
    pagesize = get_pagesize()
    with open(image_location, "r+b") as f:
        # Calculate the Offset for mmap
        adress = inode_offset + timestamp_offset
        usable_offset = adress // pagesize * pagesize
        usable_index = adress % pagesize
        mm = mmap.mmap(f.fileno(), pagesize, offset=usable_offset)

        #Convert the time to Hexadecimal
        t2h = time_to_hex(time, verbose)
        hextime = t2h[0]

        #Convert the hexadecimal time to little endian bytes
        bytetime = bytes.fromhex(hextime[2:])
        # Convert to little Endian
        littleEndianTime = bytetime[::-1]

        #Write time and save the file
        mm[usable_index:usable_index + 4] = littleEndianTime
        mm.flush()
        mm.close()
        f.close()
```

Listing B.2: Writeback Function for Files (Initialization)

```
    def writeback(image_location, partition_to_use, inode_number, file_path, verbose, update):
        #Extract Metadata
        file_system_metadata = extract_values_fsstat(...)
        inode_data = istat_extract_inode(...)

        #Convert to Hexdump and Extract the Hexbytes
        command.hexdump(file_path, "newfile")
        bytes = extractBytesFromHex("./newfile", verbose=verbose)

        #Compare Saved Length vs Actual Length
        if (inode_data.size == len(bytes)):
            # Inode Len and File Size Match
        elif (inode_data.size < len(bytes)):
            # Inode Len is smaller than File Size
            # We need to check if the write is successful
        else:
            # Inode Len is larger than File Size

        modify_binary(blocks=inode_data.blocks, bytes_array=bytes, image_location=image_location,
            fs_meta=file_system_metadata, sector_size=partition_to_use.sector_size,partition_start
            =partition_to_use.start, verbose=verbose)
        return 0
```

Listing B.3: Writeback Function for Files (Writeback)

```
def modify_binary(blocks, bytes_array, image_location, fs_meta, sector_size, partition_start,
    verbose=False):
    pagesize = get_pagesize()
    byte_num = 0
    with (open(image_location, "r+b") as f):
```

```
        blocksize = int(fs_meta.block_size)
        for block in blocks:
            # Edgecase: Block Number 0 (Superblock) from istat
            if (block == 0):
                break
            blockaddress = calculate_offset_block(...)
            usable_adress = blockaddress // pagesize * pagesize
            mm = mmap.mmap(f.fileno(), pagesize, offset=usable_adress)
            bytestart = byte_num
            zero_byte_counter = 0
            wrote_byte = False
            for i in range(blocksize):
                if (byte_num >= len(bytes_array)):
                    zero_byte_counter += 1
                    mm[i] = 0
                else:
                    wrote_byte = True
                    mm[i] = bytes_array[byte_num]
                byte_num += 1
            mm.flush()
            mm.close()
        f.close()
```

Listing B.4: Writeback Function for Files (Data Loss Check)

```
        was_usefull_data_lost = False
        for i in range(byte_num, len(bytes_array)):
            if not bytes_array[i] == 0:
                was_usefull_data_lost = True
        if (byte_num < len(bytes_array)):
            if was_usefull_data_lost:
                # Print Error that Data was lost
            else:
                # No data was lost
```

B.1.2. Logfiles

Listing B.5: Syslog class values and init

```
        class syslog_event:
            timestamp = ""
            year = ""
            timestamp_unix = ""
            message = ""
            audit = False

            def __init__(self, line):
                line = line.split(" ")
                self.timestamp = "{} {} {}".format(line[0], line[1], line[2])
                self.message = ' '.join(line[3::])
                self.message.replace("\n","")
                if self.message.__contains__("audit("):
                    self.audit = True
```

Listing B.6: Syslog class change and shift functions

```
            def change(self, new_ts):
                date, time = new_ts.split()
                old_unix_time = str(self.timestamp_unix)
```

```
4                 self.timestamp_unix = date_and_time_2_unixtime(date, time)
5                 year, month, day = date.split("-")
6                 month = months[int(month) - 1]
7                 self.timestamp = "{} {} {}".format(month, day, time)
8                 self.message.replace(old_unix_time, str(self.timestamp_unix))
9
10            def shift(self, seconds):
11                old_unix_time = str(self.timestamp_unix)
12                self.timestamp_unix += seconds
13                ts = unixtime2apttime(self.timestamp_unix)
14                date, time = ts.split()
15                year, month, day = date.split("-")
16                month = months[int(month) - 1]
17                self.timestamp = "{} {} {}".format(month, day, time)
18                self.message.replace(old_unix_time, str(self.timestamp_unix))
```

Listing B.7: Syslog class simple year generator

```
1             def year_generator_simple(parsedLogs, image_location, inode_number, partition_to_use,
                  verbose=False):
2                 inode_data = istat_extract_inode(imagepath=image_location, inode_number=
                      inode_number,
3                                                  offset=str(partition_to_use.start), verbose=
                                                       verbose)
4                 last_modified = inode_data.file_modified
5                 last_modified_year = int(last_modified[0:4])
6                 last_month = 0
7                 parsedLogs.reverse()
8
9                 for i in range(len(parsedLogs)):
10                    if i == 0:
11                        parsedLogs[i].year = str(last_modified_year)
12                        last_month = months2[parsedLogs[i].timestamp[0:3]]
13                        parsedLogs[i].timestamp_unix = generate_epochtime(parsedLogs[i].timestamp,
                              parsedLogs[i].year)
14                    else:
15                        this_month = months2[parsedLogs[i].timestamp[0:3]]
16                        if this_month > last_month:
17                            last_modified_year -= 1
18                        last_month = this_month
19                        parsedLogs[i].year = str(last_modified_year)
20                        parsedLogs[i].timestamp_unix = generate_epochtime(parsedLogs[i].timestamp,
                              parsedLogs[i].year)
21                parsedLogs.reverse()
```

Listing B.8: Syslog class complex year generator

```
1             def year_generator_complex(parsedLogs):
2                 commando.remove("./registrar.csv")
3                 commando.psteal("./Ubuntu Syslog", "registrar.csv")
4                 commando.find_delete("plaso")
5
6                 csv_data = CSVParser.parse_csv("../../registrar.csv")
7
8                 for i in range(len(csv_data)):
9                     timestamp = ParseTimestamps.log2timeline_2log(csv_data[i]["datetime"])
10                    message = csv_data[i]["message"]
11                    closing_bracket = message.find(']')
12                    message = message[closing_bracket + 1:].strip()
13                    temp = get_indexes_by_timestamp(parsedLogs, timestamp[0], message)
14                    for j in range(len(temp)):
```

```
                            parsedLogs[temp[j]].year = timestamp[1]
                            parsedLogs[temp[j]].timestamp_unix = generate_epochtime(parsedLogs[temp
                                [j]].timestamp,
                                                                                    parsedLogs[temp
                                                                                        [j]].year)
                    return parsedLogs
```

B.1.3. Checksumming

Listing B.9: Ext4 Inode Checksumming Algorithm

```
def read_inode(full_offset, disk_image):
    # Read the inode data from the disk
    # For demonstration purposes, let's assume inode data is read from a file
    with open(disk_image, 'rb') as f:
        # Seek the position of the inode on the disk
        f.seek(full_offset)
        # Read the inode data (assuming inode_size bytes per inode)
        inode_data = f.read(124)

        # Insert Zero Bytes at the place of the Checksum field
        inode_data = inode_data + bytes(2)
        f.seek(full_offset + 126)
        inode_data = inode_data + f.read(4)

        # Insert Zero Bytes at the place of the Checksum field
        inode_data = inode_data + bytes(2)

        f.seek(full_offset + 132)
        inode_data = inode_data + f.read(256 - 132)
    return inode_data


def read_uuid(offset_partition, disk_image):
    # Skips over Buffer in Block 0 and then jumps to UUID field
    total_offset = offset_partition + 1024 + 104
    with open(disk_image, 'rb') as f:
        f.seek(total_offset)
        uuid = f.read(16)
    return uuid


def read_generation(total_offset, disk_image):
    with open(disk_image, 'rb') as f:
        # Seek the position of the inode on the disk
        full_offset = total_offset + 100

        f.seek(full_offset)
        # Read the generation data
        generation_data = f.read(4)
    return generation_data


def serialize_inode(inode_data, inode_number, file system_uuid, generation):
    serialized_inode = file system_uuid + inode_number.to_bytes(4, byteorder='little') + generation
         + inode_data
    return serialized_inode


```

```
48 def convert_crc32c(crc32c_val):
49     inverter = 0xFFFFFFFF
50     inverted_crc32c = inverter - crc32c_val
51     inverted_crc32c = hex(inverted_crc32c)
52     inverted_crc32c = inverted_crc32c[2:]
53     converted_crc32c = bytearray.fromhex(inverted_crc32c)
54     converted_crc32c.reverse()
55     return converted_crc32c
56
57
58 def compute_inode_crc32c(inode_number, offset_partition, offset_inode, disk_image):
59     # Read inode data from the disk
60     inode_data = read_inode(offset_inode, disk_image)
61     generation_data = read_generation(offset_inode, disk_image)
62
63     file system_uuid = read_uuid(offset_partition, disk_image)
64
65     # Serialize inode data along with inode number and file system UUID
66     serialized_inode = serialize_inode(inode_data, inode_number, file system_uuid, generation_data)
67     # Compute CRC32C checksum
68     crc32c_checksum = convert_crc32c(crc32c.crc32c(serialized_inode))
69     return crc32c_checksum
```

Listing B.10: Code for finding an specific event in the logfile

```
def filter_events(events, timestamp, message):
    new_events = []
    old_events = []

    for event in events:
        if (timestamp is None or event.timestamp == timestamp) and (
                message is None or event.message.__contains__(message)):
            new_events.append(event)
        else:
            old_events.append(event)
    return (new_events, old_events)
```

C

Hash Sums

C.1. Baseline

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

C.2. Test Case - Mounting Artifacts

Listing C.1: Hash sum of the copied disk image of the mount test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.2: Hash sum of the modified disk image

```
bfa4768fea0ed283cd7d9b862fdefb0646fd32b6f24ff4767ccaf0df24ad4c92
```

C.3. Test Case - Metadata Manipulation

C.3.1. Touch

Listing C.3: Hash sum of the copied disk image of the Metadata-Touch test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.4: Hash sum of the modified disk image of the Metadata-Touch test case

```
d11f72074a932c378e3526883ce8410594c9b4555648eea2e770690ae4aed1ee
```

C.3.2. Debugfs

Listing C.5: Hash sum of the copied disk image of the Metadata-Debugfs test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.6: Hash sum of the modified disk image of the Metadata-Debugfs test case

```
780566143941151b0435872ce916f1e744accc0afdd1deaa2fd5b23282e3fcc7
```

C.3.3. DiskForge

Listing C.7: Hash sum of the copied disk image of the Metadata-DiskForge test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.8: Hash sum of the modified disk image of the Metadata-Debugfs test case

```
780566143941151b0435872ce916f1e744accc0afdd1deaa2fd5b23282e3fcc7
```

C.4. Test Case - Logfile Manipulation

C.4.1. Gedit

Listing C.9: Hash sum of the copied disk image of the Logfile-Gedit test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.10: Hash sum of the altered disk image of the Logfile-Gedit test case

```
ea0ff3e22de9c45ca183ad7d56e1709752bf8a7faf9827d0705670dc69158200
```

Listing C.11: Hash sum of the Syslog File of the Logfile-Gedit test case

```
aef7aa02207c1ed48af5cac0fe8bc9d69f1af4a3dceb584887dfa5b9a837324a
```

C.4.2. Sublime Text

Listing C.12: Hash sum of the copied disk image of the Logfile-SublimeText test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.13: Hash sum of the modified disk image of the Logfile-SublimeText test case

```
c1569e16d51120e5bc763d5e99b59c1664f09378bcd2bfed3f37fdaeb34a9390
```

Listing C.14: Hash sum of the Syslog File of the Logfile-SublimeText test case

```
aef7aa02207c1ed48af5cac0fe8bc9d69f1af4a3dceb584887dfa5b9a837324a
```

C.4.3. DiskForge

Listing C.15: Hash sum of the copied disk image of the Logfile-DiskForge test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.16: Hash sum of the modified disk image of the Logfile-DiskForge test case

```
c0c179647ad17fbf05c977e1fb49b5abad0a9024f8f1be8551f70f699f8f40d8
```

Listing C.17: Hash sum of the Syslog File of the Logfile-DiskForge test case

```
ebc6d40ab2f51946d2d606fdcaed5fdf2b777d0dd45e50a2851af5993553f1f3
```

C.5. Databases

C.5.1. SQLiteBrowser

Listing C.18: Hash sum of the copied disk image of the Database-SQLiteBrowser test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.19: Hash sum of the modified disk image of the Database-SQLiteBrowser test case

```
9713ac8b3771ce304ed3be5a48a77db3aa0c12d66bfd179de6373ae71ff6e858
```

Listing C.20: Hash sum of the places.sqlite File of the Database-SQLiteBrowser test case

```
f87e09b7ca038033176a4d5a3a183e7d69a2a9fec9ed367819b476f531b78dd5
```

C.5.2. DiskForge

Listing C.21: Hash sum of the copied disk image of the Database-DiskForge test case

```
be69419b766c950de25d66e97f899fe30dbb58b33d6702f21e034686211868e5
```

Listing C.22: Hash sum of the modified disk image of the Database-DiskForge test case

```
0db63f71ffddf62504039fdd387a2d658f96c70b8bf1586d9e83ae30c59b6cd9
```

Listing C.23: Hash sum of the places.sqlite File of the Database-DiskForge test case

```
f87e09b7ca038033176a4d5a3a183e7d69a2a9fec9ed367819b476f531b78dd5
```

D

Commands

In this Appendix chapter we present some of the commands used in this thesis.

D.1. General

Listing D.1: The compare command used in this thesis

```
cmp -l <Path to Base Image> <Path to Modified Image>
```

Listing D.2: The blkcat command used in this thesis

```
blkcat -o <Partition_Offset> <Path_To_Disk_Image> <Block_Number>
```

Listing D.3: The command to open gedit

```
sudo gedit <Path to Syslog>
```

D.2. Mounting Commands

Listing D.4: Creation of the mount point

```
  sudo mkdir <Mount Point>
```

Listing D.5: Mounting command

```
sudo mount -o loop,offset=<Byte Offset> <Path to Disk Image> <Mount Point>
```

Listing D.6: Checking if the partition was mounted correctly

```
df -H
```

Listing D.7: Unmount command

```
sudo umount /mnt/master
```

E

Evaluation

In this section additional material of the Evaluation Chapter is presented.

E.1. Mounting Artifacts

E.2. Metadata - Touch Testcase

Listing E.1: Commands used to execute the Touch test case

```
touch <Mount Point><Path to File>/places.sqlite
touch -t 202401020102.03 -a <Mount Point><Path to File>/places.sqlite
touch -t 202502030102.03 -m <Mount Point><Path to File>/places.sqlite
```

H

Listing E.2: Output of the parsing tool for the Metadata - Touch test case

```
How many Bytes have been changed: 13167
How many Blocks have been changed: 13
Block Numbers:
[0, 3702784, 3702785, 3702786, 3702787, 3702788, 3702789, 3702790, 3702791, 3702792, 3702793,
    3702794, 5242978]
```

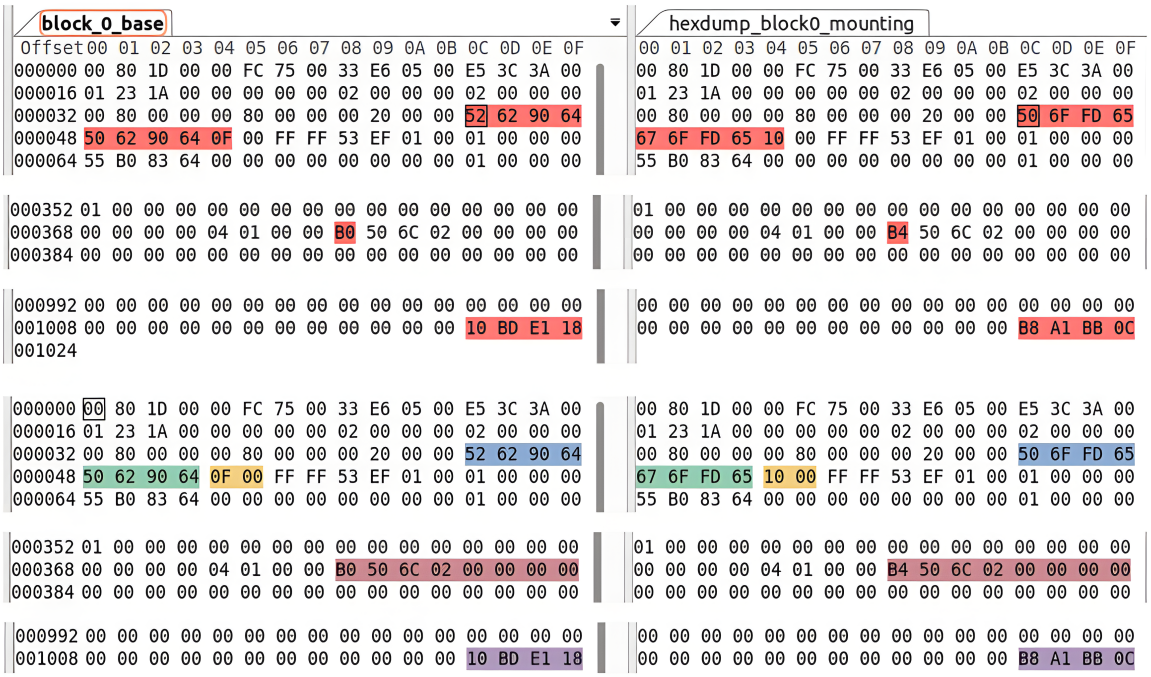

Figure E.1.: Difference of block 0 between the base image (left) and the modified image (right) after the mounting test case
The upper figure shows a diff between the base and the modified image
The lower annotates the changed structs: Last Modification Time, Last Write Time, Mount Count, Snapshot Inode Number, Checksum

Blocks	Base	Modified
0	Superblock	Superblock
3702784-3702794	Journal	Journal
5242978	Group 160 Inode Table	Group 160 Inode Table

Table E.1.: List of modified blocks during the Metadata - Touch test case and their usage

H

Listing E.3: Istat output for the places.sqlite file of the Metadata Touch Test Case

```
Inode Times:
Accessed:       2024-01-02 01:02:03.000000000 (CET)
File Modified:  2025-02-03 01:02:03.000000000 (CET)
Inode Modified: 2024-03-24 16:37:51.772748242 (CET)
File Created:   2023-06-10 01:32:18.332726057 (CEST)
```

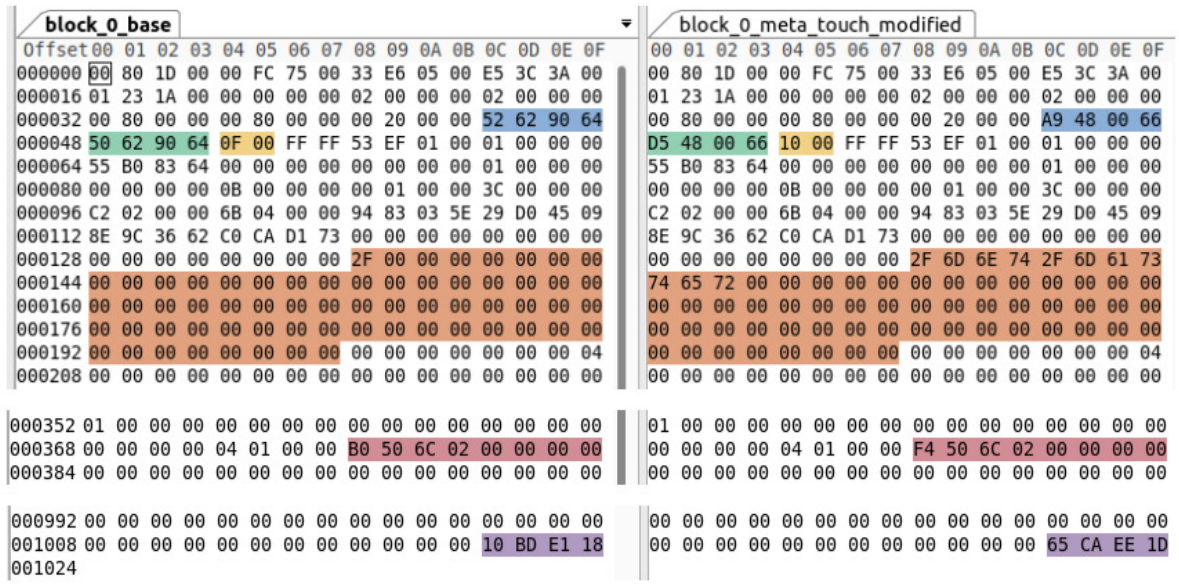

Figure E.2.: Difference between block 0 on base (left) and modified (right) disk image
Last Mount Time, Last Write Time, Mount Count, Last Mount Path, Snapshot Inode Number, Superblock Checksum

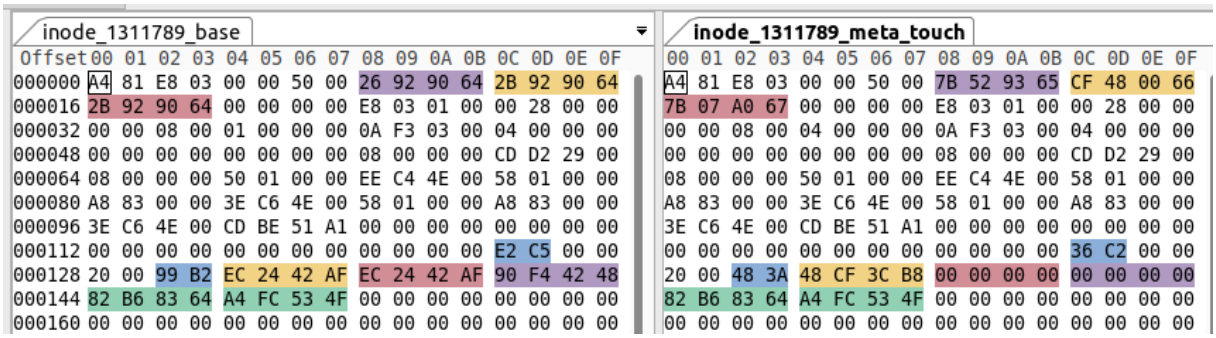

Figure E.3.: Layout of inode 1311789 on the base image and modified image
Checksum, Created timestamp, Changed Timestamp, Modified Timestamp, Accessed Timestamp

E.3. Metadata - Debugfs Testcase

Listing E.4: Command used to create loop device

```
sudo losetup -P -o <Offset> <Loop Device> <Path to Disk Image>
```

Listing E.5: Command to start debugfs

```
sudo debugfs -w <Loop Device>
```

Listing E.6: Debugfs Commands

```
debugfs: cd home/kassensystem/snap/firefox/common/.mozilla/firefox/ij9d8zia.default/
debugfs: set_inode_field ./places.sqlite atime 20240102010203
debugfs: set_inode_field ./places.sqlite mtime 20250203010203
debugfs: set_inode_field ./places.sqlite ctime 20260304010203
debugfs: set_inode_field ./places.sqlite crtime 20270405010203
debugfs: set_inode_field ./places.sqlite atime_extra 493827156
debugfs: set_inode_field ./places.sqlite mtime_extra 493827156
debugfs: set_inode_field ./places.sqlite ctime_extra 493827156
debugfs: set_inode_field ./places.sqlite crtime_extra 493827156
debugfs: quit
```

Listing E.7: Command to free the loop device

```
sudo losetup -d <Loop Device>
```

Listing E.8: Istat output for the Metadata - Debugfs test case

```
Inode Times:
Accessed: 2024-01-02 02:02:03.123456789 (CET)
File Modified: 2025-02-03 02:02:03.123456789 (CET)
Inode Modified: 2026-03-04 02:02:03.123456789 (CET)
File Created: 2027-04-05 03:02:03.123456789 (CEST)
```

Listing E.9: Debugfs Commands

```
debugfs: cd home/kassensystem/snap/firefox/common/.mozilla/firefox/ij9d8zia.default/
debugfs: set_inode_field ./places.sqlite atime 20240102000203
debugfs: set_inode_field ./places.sqlite mtime 20250203000203
debugfs: set_inode_field ./places.sqlite ctime 20260304000203
debugfs: set_inode_field ./places.sqlite crtime 20270404230203
debugfs: set_inode_field ./places.sqlite atime_extra 493827156
debugfs: set_inode_field ./places.sqlite mtime_extra 493827156
debugfs: set_inode_field ./places.sqlite ctime_extra 493827156
debugfs: set_inode_field ./places.sqlite crtime_extra 493827156
debugfs: quit
```

Listing E.10: CompareParser output for Debugfs

```
How many Bytes have been changed: 36

How many Blocks have been changed: 1
Block Numbers:
[5242978]
```

Blocks	Base	Modified
5242978	Group 160 Inode Table (1311789)	Group 160 Inode Table (1311789)

Table E.2.: Overview of changed blocks and their usage

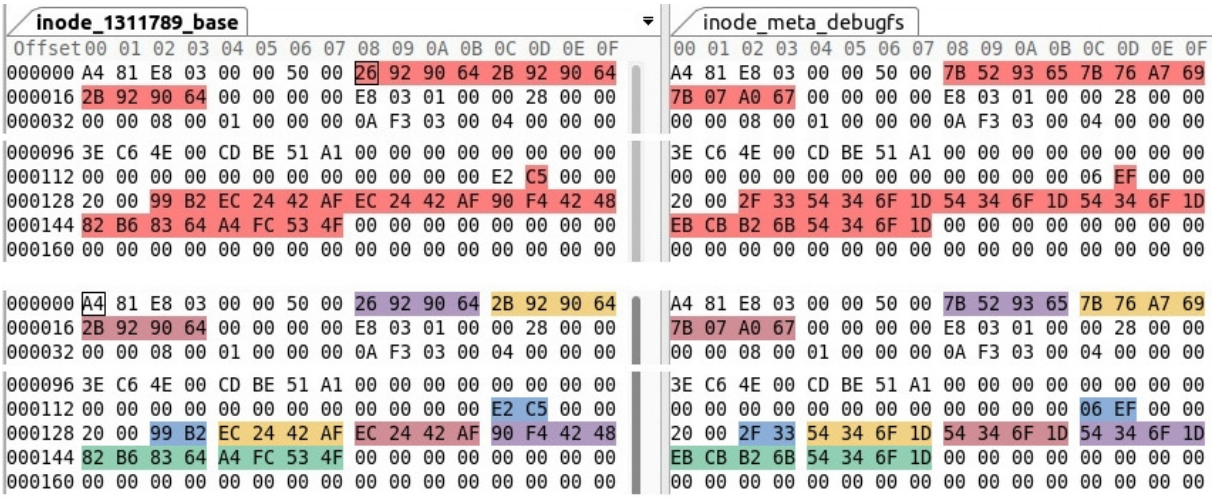

Figure E.4.: Inode 1311789 on the base image (left) and modified image (right)
Accessed Timestamp, Changed Timestamp, Modified Timestamp, Checksum, Created timestamp

E.4. Metadata - Diskforge Testcase

Blocks	Base	Modified
5242978	Group 160 Inode Table (1311789)	Group 160 Inode Table (1311789)

Table E.3.: Overview of changed blocks and their usage

E.5. Logfiles - Gedits Testcase

Table E.4.: Overview of the Changed Blocks of the Logfiles - Gedit Testcase

Blocks	Base	Modified
0	Superblock	Superblock
1	Group 0 - Metadata - Group 1 Descriptor Table	Group 0 - Metadata - Group 1 Descriptor Table
3	Group 0 - Metadata - Group 144 Descriptor Table	Group 0 - Metadata - Group 144 Descriptor Table
1030	Group 1 Data Bitmap	Group 1 Data Bitmap
1061	Group 0 Metadata - Group 0 Inode Table (multiple files)	Group 0 Metadata - Group 0 Inode Table (multiple files)

1062	Group 0 Metadata - Group 0 Inode Table (multiple files)	Group 0 Metadata - Group 0 Inode Table (multiple files)
34048-34236	Unused	Inode 1184217 File Contents
1048576	Metadata - Group 32 Data Bitmap (Syslog Data Blocks)	Metadata - Group 32 Data Bitmap
1048577	Metadata - Group 33 Data Bitmap (Syslog Data Blocks)	Metadata - Group 33 Data Bitmap
1048580	Metadata - Group 36 Data Bitmap (Syslog Extent Block)	Metadata - Group 36 Data Bitmap
3702784-3702825	Journal File Contents	Journal File Contents
4194336	Metadata - Group 128 Inode Table (/var Directory)	Metadata - Group 128 Inode Table (/var Directory)
4197415	Group 134 - Inode Table (/var/lock Directory)	Group 134 - Inode Table (/var/lock Directory)
4718608	Group 144 Metadata - Inode Bitmap (Syslog Inode)	Group 144 Metadata - Inode Bitmap
4718655	Group 144 Metadata - Inode Table (multiple files)	Group 144 Metadata - Inode Table (multiple files)
4718691	Group 144 Metadata - Inode Table (multiple files)	Group 144 Metadata - Inode Table (multiple files)
4718692	Group 144 Metadata - Inode Table (var/log/auth.log.1)	Group 144 Metadata - Inode Table (var/log/auth.log.1)
4718693	Group 144 Metadata - Inode Table (var/log/vboxadd-setup.log.1)	Group 144 Metadata - Inode Table (var/log/vboxadd-setup.log.1)
4718909	Group 144 Metadata - Inode Table (Orphan File)	Group 144 Metadata - Inode Table (Syslog File)
4718921	Metadata - Group 144 Inode Table (Syslog File)	Metadata - Group 144 Inode Table (Orphan File)
4721011	Metadata - Group 148 Inode Table (/var/log directory)	Metadata - Group 148 Inode Table (/var/log directory)
4721577	Metadata - Group 149 Inode Table (multiple files)	Metadata - Group 149 Inode Table (multiple files)
4733203	Group 144 - Data Block (/var/log directory)	Group 144 - Data Block (/var/log directory)

Table E.5.: Timeline of the Metadata changes during the Logfiles Gedit testcase

Time	**Action**	**File/Directory**
2024-03-31 16:44:23.950783793	Accessed	root
2024-03-31 16:44:24.122785376	Accessed	bin (Symlink)
2024-03-31 16:44:24.142785561	Accessed	lib (Symlink)
2024-03-31 16:44:24.174785855	Accessed	lib32 (Symlink)
2024-03-31 16:44:24.202786113	Accessed	lib64 (Symlink) sbin (Symlink)
2024-03-31 16:44:24.262786665	Accessed	libx32 (Symlink)
2024-03-31 16:44:26.298805410	Accessed	/var/

2024-03-31 16:44:30.418843340	Accessed	/var/log/
2024-03-31 16:44:33.942875782	Accessed	Syslog (old)
2024-03-31 16:56:23.222103436	Accessed	var/log/syslog.1
2024-03-31 16:56:23.234103553	Accessed	var/log/kern.log.1
2024-03-31 16:56:23.278103985	Accessed	var/log/dmesg.0
2024-03-31 16:56:23.286104063	Accessed	var/log/dmesg
2024-03-31 16:56:23.298104180	Accessed	var/log/auth.log.1 var/log/vboxadd-setup.log.1
2024-03-31 16:56:23.322104415	Accessed	var/log/faillog
2024-03-31 16:56:23.338104572	Accessed	var/log/lastlog
2024-03-31 16:56:23.342104611	Accessed	var/log/wtmp
2024-03-31 16:56:58.378447177	Created	Syslog (new)
2024-03-31 16:56:58.834451629	Modifed Changed Deleted Modifed	Syslog (old) Syslog (new)

E.6. Logfile Sublime Text Testcase

Table E.6.: Overview of the Changed Blocks of the Logfiles - Sublime Text Testcase

Blocks	Base	Modified
0	Superblock	Superblock
1	Group 0 - Metadata - Group 1 Descriptor Table	Group 0 - Metadata - Group 1 Descriptor Table
1030	Group 1 Data Bitmap	Group 1 Data Bitmap (Syslog Data Blocks)
1061	Group 0 Metadata - Group 0 Inode Table (Root Directory)	Group 0 Metadata - Group 0 Inode Table (Root Directory)
34048-34236	Unused	Syslog File Contents
34304-34492	Unused	Sublime Text Buffer
1048576	Metadata - Group 32 Data Bitmap (Syslog Data Blocks)	Metadata - Group 32 Data Bitmap
1048577	Metadata - Group 33 Data Bitmap (Syslog Data Blocks)	Metadata - Group 33 Data Bitmap
1048580	Metadata - Group 36 Data Bitmap (Syslog Extent Block)	Metadata - Group 36 Data Bitmap
3702784-3702813	Journal File Contents	Journal File Contents
4194336	Metadata - Group 128 Inode Table (/var Directory)	Metadata - Group 128 Inode Table (/var Directory)
4718921	Metadata - Group 144 Inode Table (Syslog File)	Metadata - Group 144 Inode Table (Syslog File)
4721011	Metadata - Group 148 Inode Table (/var/log directory)	Metadata - Group 148 Inode Table (/var/log directory)

Listing E.11: Result of the comparison between the base and the gedit modified disk image

```
How many Bytes have been changed: 839757
How many Blocks have been changed: 252
Block Ranges:
[(0, 1), (3, 3), (1030, 1030), (1061, 1062), (34048, 34236), (1048576, 1048577), (1048580, 1048580)
    , (3702784, 3702825), (4194336, 4194336), (4197415, 4197415), (4718608, 4718608), (4718655,
    4718655), (4718691, 4718693), (4718909, 4718909), (4718921, 4718921), (4721011, 4721011),
    (4721577, 4721577), (4733203, 4733203)]
```

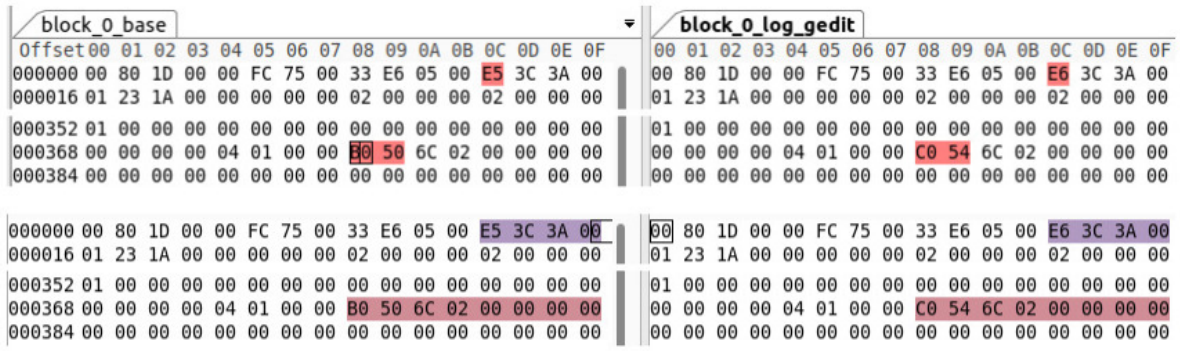

Figure E.5.: Differences between block 0 on the base (left) and modified (right) disk image
Number of free blocks, Number of KiB written over lifetime

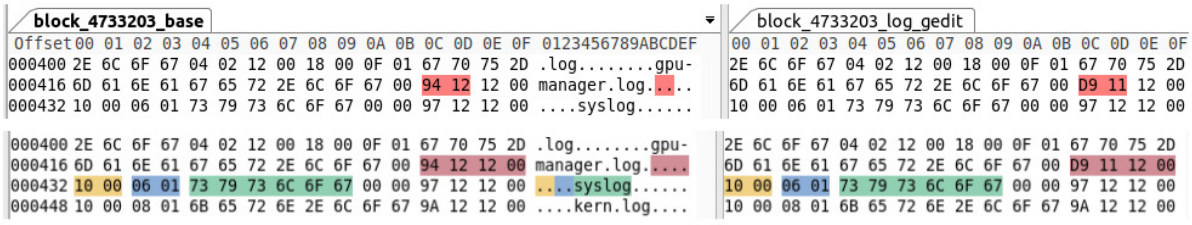

Figure E.6.: Differences between block 4733203 on the base (left) and modified (right) disk image
Inode Number, Entry Length, File Name Length, File Name

Table E.7.: Timeline of the Metadata changes during the Logfiles Sublime testcase

Time	Action	File/Directory
2024-03-31 17:42:39.926669998	Accessed	root
2024-03-31 17:42:40.990679496	Accessed	/var
2024-03-31 17:42:42.230690566	Accessed	/var/log
2024-03-31 17:42:43.854705063	Accessed	Syslog
2024-03-31 17:44:13.119503190	Modifed Changed	Syslog

E.7. Logfiles - DiskForge Testcase

Blocks	Base	Modified
1059591 - 1059728	Group 32 Data Block	Group 32 Data Block (Syslog)

Table E.8.: Overview of the Changed Blocks of the Logfiles - DiskForge Testcase

E.8. Databases SQLiteBrowser Testcase

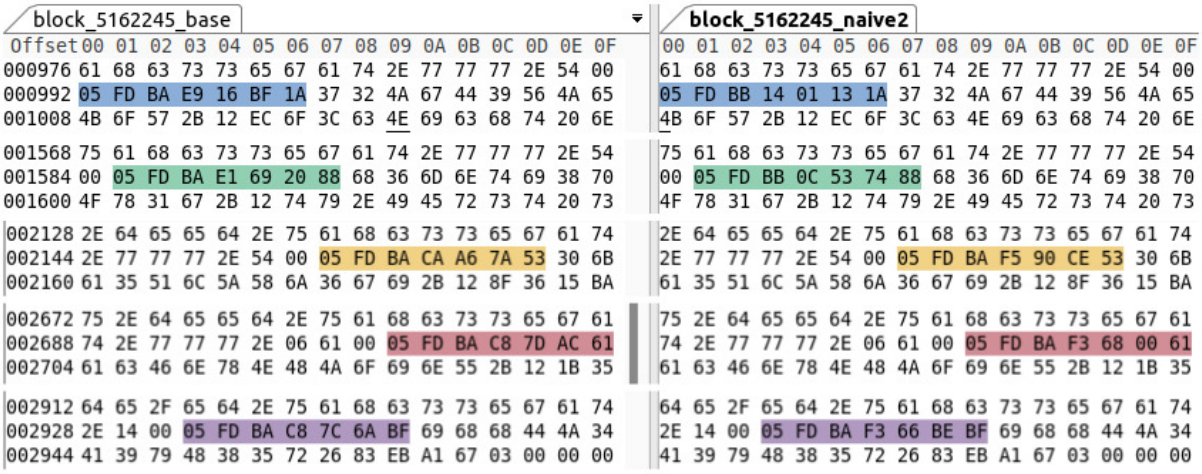

Figure E.9.: Comparison of the block 5162245 between the base disk image (left) and the modified disk image (right)
The changes show the modification of the *last_visit_date* of the corresponding events
Events: Event ID 12, Event ID 11, Event ID 10, Event ID 9, Event ID 8

E.9. Databases DiskForge

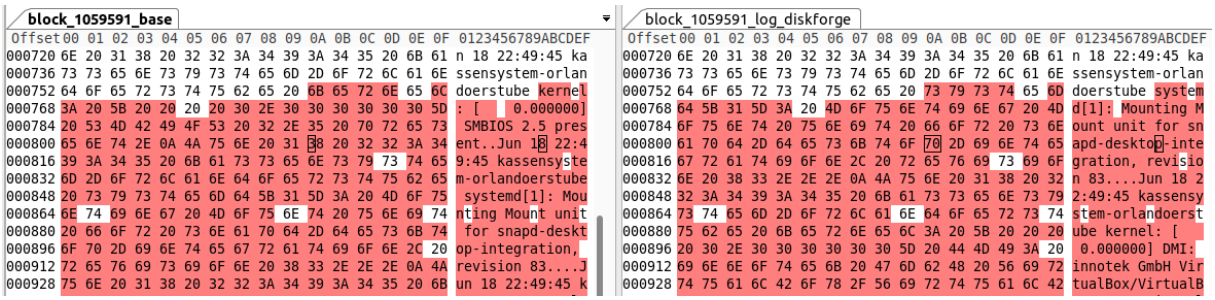

Figure E.7.: Difference between block 1059591 on the base (left) and the modified (right) disk image

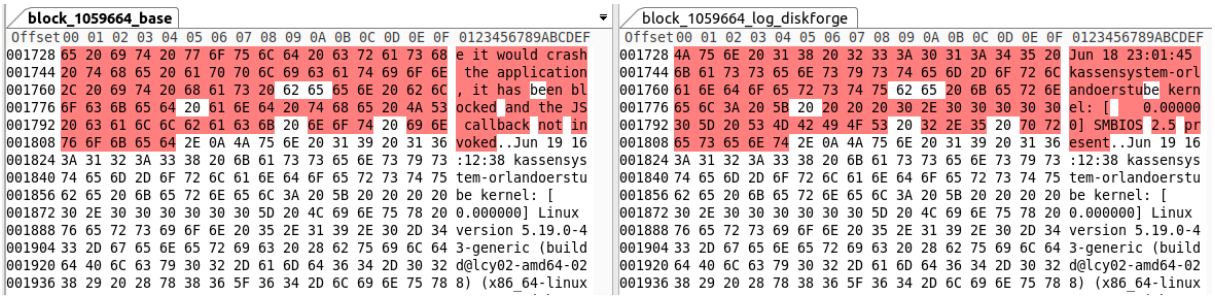

Figure E.8.: Difference between block 1059664 on the base (left) and the modified (right) disk image

ID	last_visit_date	Timestamp	Shifted	Converted
8	1686353552894655	Friday, 9. June 2023 23:32:32.894	Friday, 9. June 2023 23:44:32.894	1686354272894655
9	1686353552976993	Friday, 9. June 2023 23:32:32.976	Friday, 9. June 2023 23:44:32.976	1686354272976993
10	1686353589205587	Friday, 9. June 2023 23:33:09.205	Friday, 9. June 2023 23:45:09.205	1686354309205587
11	1686353971060872	Friday, 9. June 2023 23:39:31.060	Friday, 9. June 2023 23:51:31.060	1686354691060872
12	1686354099879706	Friday, 9. June 2023 23:41:39.879	Friday, 9. June 2023 23:53:39.879	1686354819879706

Table E.9.: Conversion and Shifting of the *last_visit_date* timestamps

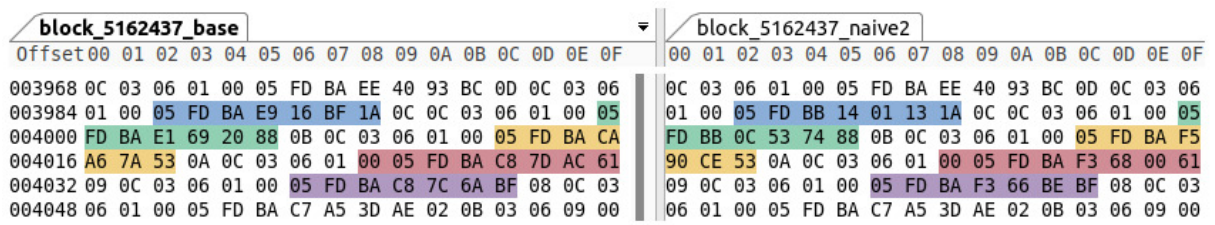

Figure E.10.: Comparison of the block 5162245 between the base disk image (left) and the modified disk image (right)
The changes show the modification of the *last_visit_date* of the corresponding events
Events: Event ID 12, Event ID 11, Event ID 10, Event ID 9, Event ID 8

Blocks	Base	Modified
0	Superblock	Superblock
12229 - 12244	Unallocated PNG	Temporary Buffer
3702784 - 3702822	Inode 8	Inode 8
5162245	Group 157 Data Block (places.sqlite)	Group 157 Data Block (places.sqlite)
5162430	Group 157 Data Block (places.sqlite)	Group 157 Data Block (places.sqlite)
5162437	Group 157 Data Block (places.sqlite)	Group 157 Data Block (places.sqlite)
5242975	Inode 1311729	Inode 1311729
5242976	OrphanFile-1311745	OrphanFile-1311745
5242978	Inode Data (places.sqlite)	Inode Data (places.sqlite)
5242982	OrphanFile-1311842	OrphanFile-1311842
5319680 - 5319687	Unallocated	Nulled
5320192	Unallocated	Partially Nulled

Table E.10.: Overview of the Changed Blocks of the Database - SQLiteBrowser Testcase

Time	**Action**	**File/Directory**
2024-04-07 15:56:35.813059702	Accessed	inode: 1311789
2024-04-07 15:56:35.817059742	Created	inode: 1311745
	Created	inode: 1311842
2024-04-07 15:56:35.821059781	Accessed	inode: 1311842
2024-04-07 16:00:18.435231298	Accessed	inode: 1311745
	Modified Changed	inode: 1311789
2024-04-07 16:00:18.447231415	Modified Changed	Inode 1311729
	Modified Changed Deleted	inode: 1311745
	Modified Changed Deleted	inode: 1311842

Table E.11.: Timeline of the Metadata changes during the Databases SQLiteBrowser testcase

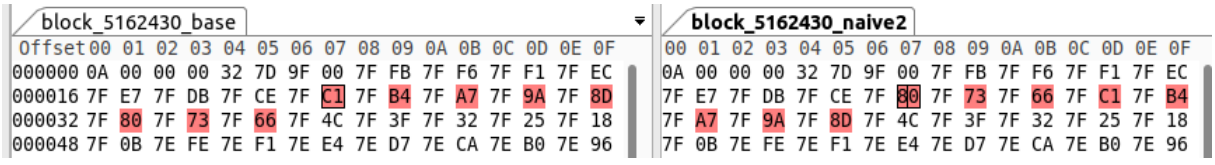

Figure E.11.: Comparison of the block 5162245 between the base disk image (left) and the modified disk image (right)
Differing Bytes between the two images

Blocks	Base	Modified
5162245	Group 157 Data Block (places.sqlite (Enries))	Group 157 Data Block (places.sqlite (Enries))
5162430	Group 157 Data Block (places.sqlite)	Group 157 Data Block (places.sqlite)
5162437	Group 157 Data Block (places.sqlite (Timestamps))	Group 157 Data Block (places.sqlite (Timestamps))

Table E.12.: Changed blocks of the Database Diskforge testcase