Lehrstuhl für Informatik 1
Friedrich-Alexander-Universität
Erlangen-Nürnberg

BACHELOR THESIS

Linux Timestamp Forensics on the Application Layer

Niclas Pohl

Erlangen, July 1, 2021

Examiner: Prof. Dr.-Ing. Felix Freiling
Advisor: Dr. Aurélien Thierry
Dr. Tilo Müller

b

Eidesstattliche Erklärung / Statutory Declaration

Hiermit versichere ich eidesstattlich, dass die vorliegende Arbeit von mir selbständig, ohne Hilfe Dritter und ausschließlich unter Verwendung der angegebenen Quellen angefertigt wurde. Alle Stellen, die wörtlich oder sinngemäß aus den Quellen entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

I 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.

Der Friedrich-Alexander-Universität, vertreten durch den Lehrstuhl für Informatik 1, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Arbeit einschließlich etwaiger Schutz- und Urheberrechte eingeräumt.

Erlangen, July 1, 2021

[Niclas Pohl]

Zusammenfassung

Das Forschungsgebiet der digitalen Forensik gewinnt in den letzten Jahren immer weiter an Bedeutung. Durch die wachsende Nutzung von digitalen Geräten in allen Bereichen des täglichen Lebens und in der Industrie wird diese Entwicklung immer weiter mit angetrieben. Durch diesen Anstieg gibt es immer mehr Daten die forensisch ausgewertet werden müssen, sei es für Strafverfolgungsbehörden um Verbrechen aufklären zu können, oder aber für die Behandlung von Sicherheitsverstößen in einem industriellen Kontext, um mit diesem wissen besser auf Gefahren reagieren zu können.

Die sogenannte Zeitstempelforensik, welche sich auf die Zeitstempel von Dateien fokussiert, bietet eine Möglichkeit durch ihre Erkenntnisse das Rekonstruieren von Abläufen zu ermöglichen.

Das Ziel dieser Bachelorarbeit ist es, ein größeres Verständnis für das Verhalten von Dateieditoren unter *Linux* zu generieren. Um dies zu erreichen, wurde ein Programm geschrieben welches automatisch die Zeitstempelveränderungen durch Editoren testen kann. Die daraus resultierenden Daten wurden dann händisch ausgewertet, verglichen und gefundene Auffälligkeiten wurden weiter untersucht.

Abstract

The importance of digital forensics as a research field grew more and more in the last years. This is due to the rising usage of digital technology in all aspects of the daily life and in the industrial sector, which drives forward this development immensely. Due to this development the amount of data which needs to be forensically examined grew enormously. Two of the main usage areas for digital forensics are law enforcement in order to solve crimes and incident response in order to better handle and detect for example a ransomware attack. reconstruct which actions left which traces on the system.

The so called timestamp forensic is a sub field of digital forensic and deals with the timestamp of files and what changes them. With knowledge about what processes change what timestamps, it is possible to reconstruct events.

The aim of the bachelor thesis is to develop a deeper understanding of the behavior of different file editors under *Linux*. To achieve this a program was developed, which is able to test automatically which timestamp modifications are caused by different actions of file editors. The resulting data was then manually analysed, compared and different anomalies where further investigated.

Contents

1

INTRODUCTION

"The ubiquity of digital devices means that digital evidence may be present in almost every crime. This offers new opportunities for police investigations. However, the proliferation of devices is increasing demand for digital forensic techniques" [32]. As this excerpt from a research briefing of the *UK Parliamentary Office of Technology & Research* shows, the importance of digital forensics grew more and more in the last years.

This is, because the amount of technology used by today's population is increasing rapidly as the figures 1.1 and 1.2 shows.

Figure 1.1: Usage of technology in the population [33]

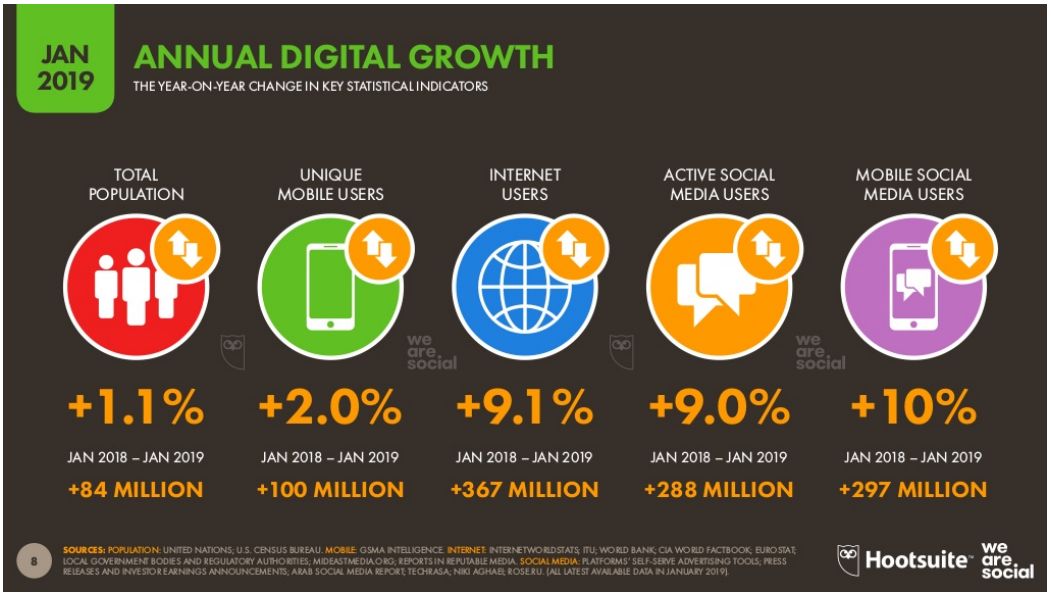

Figure 1.2: Annual growth of technology usage [33]

Due to this increase in world wide technology usage, the relevance of digital forensics in today's world increases rapidly.

The term "*Digital Forensics*" isn't really defined by a standard, but the *UK Forensic Science Regulator* defines it as the following: "the process by which information is extracted from data storage media (e.g. devices, systems associated with computing, . . .), rendered into a usable form, processed and interpreted for the purpose of obtaining intelligence for use in investigations, or evidence for use in criminal proceedings" [32].

Another definition is given by the *BSI*, the *Bundesamt für Sicherheit in der Informationstechnik*, in its "*Leitfaden IT-Forensik*" the term *IT-Forensic* is defined as the "strict methodical data analysis of disks

and computer networks for the reconnaissance of incidents under inclusion of the possibilities of strategic preparation from the view of a IT system operator". [22] Two of the main fields, which use digital forensics are "*Incident Response*" and "*Law Enforcement*".

Incident Response is the way a corporation, agency etc. responses to a to an attack, in this context to a cyber-attack. The normal way of acting can be broken down into six steps as shown by Patrick Kral in "*The incident handlers handbook*" [28]:

Step	Description
Preparation	Knowledge of assets, infrastructure, and preparation of defenses
Identification	Detecting of attacks and identification of compromised assets
Containment	Collection of evidence and mitigation of impact
Eradication	Blocking of relevant artifacts and re-imaging machines
Recovery	Regain operational capabilities and ensure that no risk for reinfection is given
Lessons Learned	Updating of techniques

Law Enforcement: The international police organisation *Interpol* says , that "the main goal of digital forensics is to extract data from the electronic evidence, process it into actionable intelligence and present the findings for prosecution. All processes utilize sound forensic techniques to ensure the findings are admissible in court" [25]. As seen here, digital forensic isn't only used for companies to conduct incident response, but also used by law enforcement to gather additional evidence or to reconstruct a sequence of events.

1.1 Motivation

A part of the research field digital forensics are timestamp forensics. The focus lies on file timestamps and how they get changed, by not only the OS but also programs. The importance of timestamps in the context of digital forensics is explained by Aurelien Thierry in his work on "*Mac(b) timestamps across POSIX implementations*". "File timestamps are crucial forensics artifacts when investigating a machine during a security incident, they are regularly modified and can provide both primitive information and inferred information"[20]. As shown in this quote, file timestamps play an important role, when investigating a security incident and can be very helpful when recreating events or in order to find suspicious activity on a system.

As a resource on timestamps, the "*Windows Time Rules*" from SANS [1] gives a great overview on how timestamps work on the operation system *Windows*. But unfortunately there isn't an equivalent for the operation system *Linux*.

This thesis is part of a bigger project which aims to do exactly this. Research has already be done by Dr. Aurelien Thierry on the Operation System Layer by analysing Linux, Free-BSD and Open-BSD and by Berenger Temgoua Dibanda on the Graphical Middle-ware/Libraries. The goal now is to research file timestamps on the application layer by analysing the behavior of various file editors.

1.2 Task

The thesis will focus on timestamp forensics and how different applications affect these timestamps. It will explore how different text editors change the MACB timestamps of files.

Timestamp	Description
M(odify)	Time when the file was last modified
A(ccess)	Time when the file was last accessed
C(hange)	Time when the status of the file was last changed
B(irth)	Time when the file was created

Table 1.1: Timestamps

In order to accomplish this, a program needs to be created which is used to acquire a data set of these timestamp modifications. The programs goals are to simulate user input as close as possible and to automate the gathering of the data. For this program, four different use cases were chosen to simulate user input as shown in table 1.2:

Test-name	Description
Access Test	Opening and closing a file without modifying its contents
Modify Test	Opening, modifying and saving a file
Save without Modification Test	Opening, saving and closing a file without modifying its contents
Modification without Saving Test	Opening, modifying and closing a file without saving the modifications

Table 1.2: Description of the test-cases

Furthermore three different file sizes where chosen as testfile sizes, to test the editors on as seen in table 1.3:

Size Name	File size
Empty	0 Bytes
Small	2 Bytes
Large	$\sim$6.9 MB

Table 1.3: Description of the file sizes

One of the challenges hereby lies in how to control different graphical applications which often require unique instructions.
After the acquisition of the aforementioned data, the results need to be analysed in order to detect abnormal patterns.

1.3 Related Work

In the "Related Work" chapter of this thesis a short overview of the preceding work in this project done by Dr. Aurelien on "*MAC(B) Timestamps across POSIX implementations (Linux, OpenBSD, FreeBSD)*"

[20] and Berenger Temgoua Dibanda thesis on "*Timestamp Forensics on graphical middleware/libraries on Linux*" [21] will be given. Furthermore the paper "*Forensic Application-Fingerprinting based on File System Metadata*" [27] by Freiling et al. is compared to this thesis as it gives an interesting approach to event reconstruction through timestamp forensics. The aim is to point out some similarities and differences between this thesis and the related works in order to give a more broad view on the topic at hand.

1.4 Results

In the context of this thesis, a program was build, which is able to execute 4 test-cases for 15 different text editors (+different configurations) on 3 different file sizes. Through the usage of the aforementioned program, the behavior of the different file editors regarding timestamp modification was analysed.
Through this analysis some interesting patterns have been found:

- The behavior of the *codeblocks* file editor when working on an empty file while executing the *Access*, *Save No Modify* and *Modify No Saving* Tests, were contrary to all other editors, no timestamps got modified.
- The behavior found in the *Modify Test* can be grouped in editors which either modify the existing file, or create a new file in which the old file and the changes get written into.
- The behavior found in the *Save No Modification Test* can be put into two different groups. File editors, which save the not existing modification and exhibit while saving the same behavior as in the *Modify* test. The other group only updates the *Access* timestamp.
- The behavior found in the *Modification but not Saving* test was in all editors an update of the *Access* timestamp, except for the *codeblocks* on an empty file test, where no timestamp was updated.

1.5 Outline

The remaining thesis is structured as follows.
In Chapter 2, some background knowledge about *Syscalls*, *Ext4* and *timestamps* is presented. Next, the structure of the project code and some key components of the implementation get highlighted and explained. The implementation is followed by an evaluation of the results gained through the aforementioned code and interesting behavior is discussed. In Chapter 5 an overview about different relevant scientific work is given and their relation to this thesis is discussed. Last but not least, the thesis is concluded in Chapter 6 and an insight into possible future work is given.

1.6 Acknowledgments

I would like to thank Dr. Aurelien Thierry for all the time he sacrificed and all the help and knowledge he provided me in order to create this thesis.
Another big thank you goes to my uni colleagues who helped me when I was stuck at problems and gave me constructive feedback on my progress.

2 BACKGROUND

In order to fully understand what the results are and what's discussed in the evaluation chapter of this thesis, the essential background knowledge is provided in this chapter. Section 2.1 will explain what a so called *system call* is and describes what different system calls are used for the evaluation in chapter 4. The next part, section 2.2, gives a short introduction on the file system *ext4* and what its features are. This part is followed with a short overview on what timestamps there are and what their meaning is in section 2.3. The chapter is concluded in section 2.4 which explains, how an application is able to trigger a system call and how it uses a library to accomplish this.

2.1 System Calls

A "system call is the fundamental interface between an application and the Linux kernel" [2]. Most of the time a program doesn't directly make a system call, but rather invokes it, by calling a library function. When a user program runs on a host machine, it runs as unprivileged on the privilege layer 3. The problem is, the kernel works on the privilege layer 0 as seen on 2.1, so the system call is used to make the user program able to request a service from the kernel. This is done mostly for two reasons.
Firstly it improves the fault tolerance of a given system.
Furthermore this hierarchy is able to prevent some malicious behavior executed by different applications and thus makes the system more secure.
To be able to use the resources provided by a more privileged layer, so called *call gates* need to be used. This only works in a predefined manner and thus if configured properly prevents unwanted access of system resources. [31][30][29]

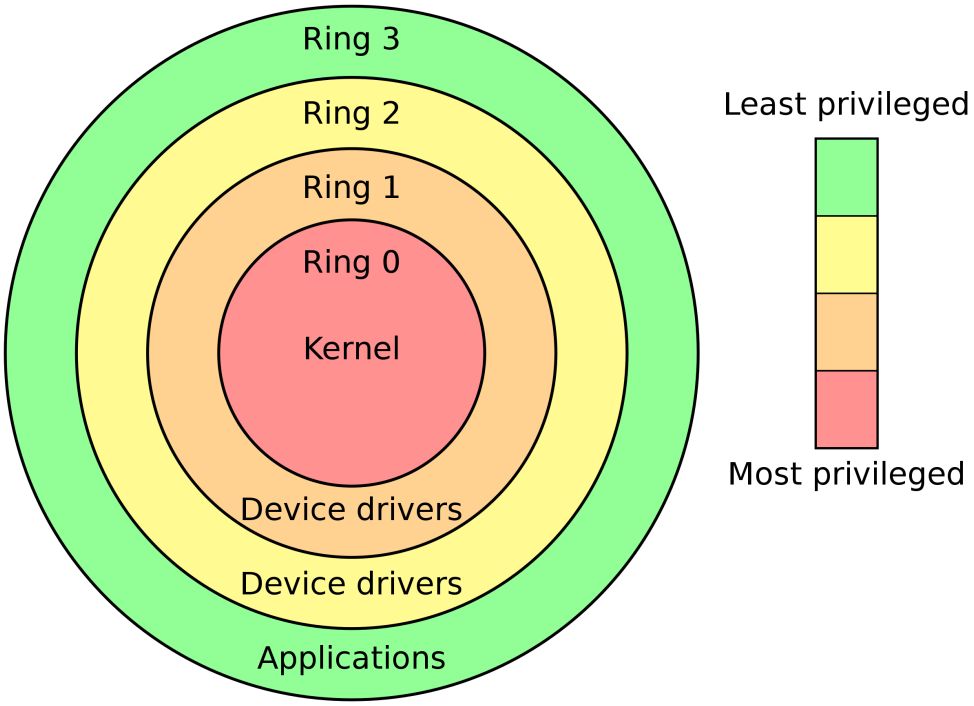

Figure 2.1: Privilege Layers [10]

As seen in figure 2.1 there are also the ring layers 1 & 2. The Ring Layer 1 is reserved for *device drivers*, as is Ring Layer 2. These two exist, but are most of the time not used by *Linux*.
There are many syscalls used by the operation system *Linux*, for example at the time of writing this thesis there are 332 syscalls for the x86_64 *Linux Kernel*, and it would take to much time and space to list all of them. Most of the time an explanation of a syscall can be found in the *man-pages* of the system. Because of this, in the following the syscalls, which are typically involved in the modification of files, are explained to better understand the results of the evaluation.

2.1.1 open

The *open* syscall opens a specified file and returns the file descriptor of the opened file. If the file isn't existing at the time of the call and the "*O_CREAT*" flag is set, a new file with the specified name is created. On success the *open* syscall typically returns a small non negative number which is used in following syscalls to refer to this opened file [9].

2.1.2 openat

The *openat* syscall functions in nearly the same way as the open syscall. The main difference is, "If the path-name given in path-name is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd*". On success the *openat* syscall typically returns a small non negative number which is used in following syscalls to refer to this opened file [9].

2.1.3 write

The *write(fd, *buf, count)* syscall writes from a specified buffer to a specified file. The number of bytes written doesn't exceed the number of bytes specified in count. It is worth noting, that "POSIX requires that a read(2) that can be proved to occur after a write() has returned will return the new data" [19]. On a success, the number of bytes which have been written is returned, but if a failure occurred, -1 is returned.

2.1.4 read

The *read* syscall reads from a specified file and writes it to a specified buffer. The number of bytes read doesn't exceed the number of bytes specified in the *count* variable at the call time of the syscall. If the *read* syscall is executed successfully, the number of bytes read is returned. If an error occurred, -1 is returned. [12]

2.1.5 unlink

The *unlink* syscall deletes a specified name from the filesystem. If this was the last link to the file, the file then gets deleted. If an error occurred, -1 is returned otherwise 0. [18]

2.1.6 rename

The *rename* syscall changes the name of the specified file. Hard-links to the file are not affected by this. If it was executed successfully, 0 is returned and on an error -1 is returned. [14]

2.2 Ext4

Ext4 (forth extended file system) is the fourth revision of the extended file-system which was developed for the Linux kernel and is backwards compatible for *ext2* and *ext3*. It was developed as a successor to *ext3* and features a 16TB file size limitation and nanosecond resolution timestamps. The default inode size of ext4 is 256 byte. Due to this size it is possible to store timestamps with a higher resolution and also it is possible to store small files in the inode. If a smaller inode size is used it can happen, that the timestamps are getting saved in a smaller resolution or that the birth timestamp isn't saved at all. [4][3]

2.3 Timestamps

Most file systems support the three timestamps

Timestamp	Description
mtime	time of last file modification
atime	time of last file access
ctime	time of last file change

Table 2.1: mtime, atime and ctime

which stand for *modification time*, *access time* and *change time*. But *ext4* and the *xfs* file system support a fourth timestamp

Timestamp	Description
btime	time of creation of the file

Table 2.2: btime

which stands for *birth time*. In the following section, these different timestamps will be explained. [23][15][5]

2.3.1 Modify

The *modify* timestamp represents the point in time, when the contents of the file have last been changed. It will be changed, if the contents of the file get changed.

2.3.2 Access

The *access* timestamp represents the point in time when the file was last accessed. Reading the file contents or opening the file will update this timestamp.

2.3.3 Change

The *change* timestamp represents the point in time when the status or the metadata of the file have last been modified. Every time when the *Modify* timestamp gets updated, the *Change* timestamp will also be updated. Further possibilities for a modification of the timestamp are:

- Change of the file permissions
- Change of the file ownership
- Change of the file location

2.3.4 Birth

The *birth* timestamp represents the point in time, when the file was first created on the local file system. This means, that if a file is created and modified on another file system and then copied to the local file system, that the *birth* timestamp can be on a later point in time than the *access* and the *modify* timestamp.

2.4 How Applications trigger Syscalls

But how exactly does a program use a syscall? Most of the time when developing applications, you don't need to know how exactly a specific action is going to be executed with syscalls, because a library will do the work for you and knows how to perform specific actions with syscalls. One of these libraries is "*glibc*", which stands for "*GNU C Library*". It provides the programmer with many critical APIs and functions, for example "*open, read, write, malloc, printf, getaddrinfo, dlopen, pthread_create, crypt, login, exit*" and many more.
So how does now the user program get the kernel to execute its command. The user program calls the library function which has the functionality coded into it. The library function then escalates the call to the kernel. [31][7][30][29]

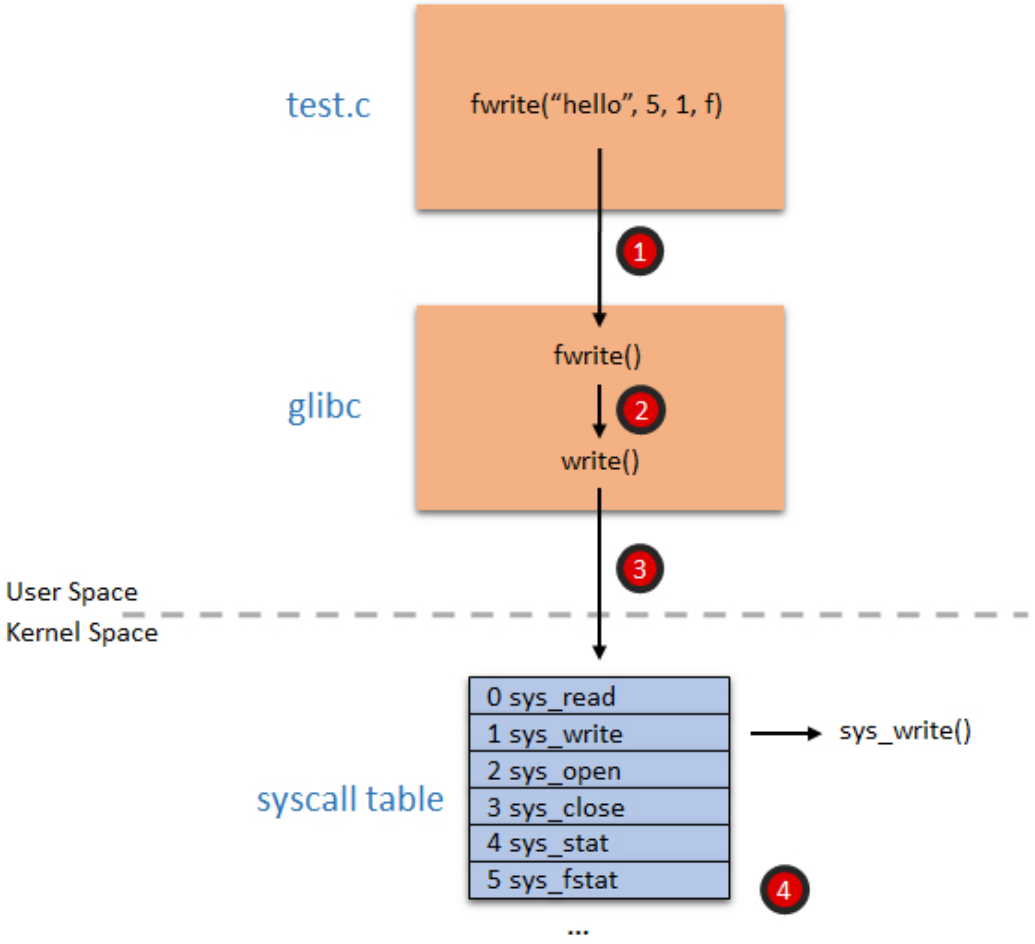

Figure 2.2: Execution of fwrite to syscall [30]

As seen in 2.2, the "*fwrite()*" function call in "*test.c*" gets mapped to the "*fwrite()*" function in "*glibc*". This function then gets mapped to the corresponding syscall through looking it up on the syscall table. Between the *glibc* and *syscall table* the change from user space to kernel space is seen as this is the location, where the change from Layer 3, the Application Layer, to Layer 0, the Kernel Layer occurs.

3 IMPLEMENTATION

This chapter aims at explaining the structure and highlighting different key components of the project. To get a deeper understanding of the project code, the code is documented and self explaining names for functions and variables are used wherever possible.

Listing 3.1: Example of commented code

```
1 def vim_test(mode):
2     #Checks if the program is installed
3     if (doesProgramExist("vim") == -1):
4         notFoundEditors.append("VIM")
5         return
6     x = vimVersion()
7     print("Vim Version: " + x)
8     #prints the testname
9     print("vim --clean")
10    #runs the tests
11    vim_Access(sleeptime, verbose, mode)
12    vim_Modify(sleeptime, verbose, mode)
13    vim_safe_no_mod(sleeptime, verbose, mode)
14    vim_mod_no_save(sleeptime, verbose, mode)
15    print()
```

The whole code of the project is found under `https://github.com/QuoSecGmbH/os_timestamps`. The first section will document what different libraries were used in this project. After that, section 3.2 will give a short overview on what different files the project consists of and what their purpose is. In the fol-

lowing section, the possible run configurations are shown and explained. In section 3.4 the structure of the test-cases is highlighted and briefly explained. The following section shows the creation of new files and what different modes there are. Lastly in section 3.6 the acquisition of timestamp data is shown and some design choices are explained.

3.1 Framework

The code for the automated test cases was written in *Python 3.8* with the usage of the following libraries:

Library Name	Usage in project
os	Used for fast calling CLI commands
sys	Piping the stdout into a file
subprocess	For calling background process and getting timestamps in background process
dataclasses	Building the Timestamp object
pyautogui	Simulating all user input in editors
time	Used for the sleep function call
datetime	Timing the execution time of the tests
argparse	Used to parse arguments at program start

Table 3.1: Used libraries

On a fresh install of python3 it is possible that the library *pyautogui* is not already installed. If this is the case in order to install *pyautogui* some other applications need to be installed for it to work. The following commands should install everything needed:

Listing 3.2: Commands needed to fully install pyautogui

```
1 sudo apt-get install scrot
2 sudo apt-get install python3-tk
3 sudo apt-get install python3-dev
4
5 python3 -m pip install pyautogui
```

3.1.1 pyautogui library

The most used library in this project is *pyautogui*. The library is created to let python scripts control the systems keyboard and mouse, in order to automate tasks. As stated on the *readthedocs.io* page of *pyautogui* some of its features are [13]:

- Moving the mouse and clicking or typing in the windows of other applications
- Sending keystrokes to applications (for example, to fill out forms)
- Take screenshots, and given an image (for example, of a button or checkbox), find it on the screen
- Locate an application's window, and move, resize, maximize, minimize, or close it (Windows-only, currently)

- Display message boxes for user interaction while your GUI automation script runs

It was used to simulate the user input that controls the file editors, through either direct input of keystrokes or the usage of shortcuts.
But the usage of this library had several downsides. For example some keystrokes got wrongly mapped on the german "*QWERTZ*" keyboard layout and thus some commands couldn't get executed. To counter this, different commands were used to achieve the same results. Furthermore a English "*QWERTY*" keyboard layout could get used, in order to circumvent the problem.
Another problem originated from the missing of the window control under *Linux*. Because of this problem under some systems the windows didn't focus correctly and thus the keyboard input was send to the wrong application. This resulted in a complete failure in executing the project code and could possibly alter the contents of other files or possible crash other programs. On tests under *Ubuntu* this didn't happen.
One of the upsides of using the library was the option to hold down keys. This enabled the possibility to use shortcuts like *CTRL + S*, *CTRL + Q* or *CTRL + W*.

3.2 Project Scope

The project consists of 7 different files which are all needed in order to execute the code.

File	Description
editor_test.py	The file of the project, which gets executed. Responsible for starting the test cases and parsing the arguments
tests_editors.py	Stores all different test-cases for the editor tests
versions.py	Stores all functions for getting the version numbers of the editors
utility.py	Stores functions for code that's often used in the project and makes the code more readable
shortcuts.py	Stores keyboard shortcuts which are executed via pyautogui
config_vim	Stores a config file for the vim set nowritebackup test-cases
prototype_file_ts	Used for getting the statx functionality

Table 3.2: Project Files

If one of the files is missing, it is not possible to run the code. The file *prototype_file_ts* can be found in a not compiled form in the GIT repository and from there be compiled by the user.

3.3 Running the Code

The project code was designed, so that the way it runs is as much configurable as possible. To achieve this, the argparse library was used. With this library it is possible to easily parse the arguments given by user. Below in listing 3.3 all the possible arguments are listed.

Listing 3.3: Program run options

```
1 python3 editor_test.py --help
2 usage: editor_test.py [-h] [-a] [-v] [-t TIME] [--vim] [--vimown]
```

```
3                       [--vimchoose VIMCHOOSE] [--nano] [--gedit] [--kate]
4                       [--sublime] [--geany] [--atom] [--emacs] [--codeblocks]
5                       [--bluefish] [--texstudio] [--code] [--leafpad]
6                       [--notepadqq] [--jed] [--empty] [--small] [--large]
7
8  optional arguments:
9    -h, --help            show this help message and exit
10   -a, --all             runs all tests
11   -v, --verbose         prints timestamp before and after test as well as
12                         inode number
13   -t TIME, --time TIME  set sleep timer manually
14   --vim                 runs the vim tests
15   --vimown              runs the vim tests with system used vimrc
16   --vimchoose VIMCHOOSE
17                         runs the vim tests with choosen config file
18   --nano                runs the nano tests
19   --gedit               runs the gedit tests
20   --kate                runs the kate tests
21   --sublime             runs the sublime tests
22   --geany               runs the geany tests
23   --atom                runs the atom tests
24   --emacs               runs the emacs tests
25   --codeblocks          runs the codeblocks tests
26   --bluefish            runs the bluefish tests
27   --texstudio           runs the texstudio tests
28   --code                runs the visualstudiocode tests
29   --leafpad             runs the leafpad tests
30   --notepadqq           runs the notepadqq tests
31   --jed                 runs the jed tests
32   --empty               runs the tests with empty files
33   --small               runs the tests with small files
34   --large               runs the tests with large files
```

When running the code without any arguments, the *verbose*, *vimown* and *vimchoose* configurations are disabled. Furthermore if one or more editors are specified in the run configurations, only these specified editors will be tested.
It has to be noted, that for some of the editors, the *large* file test cannot be executed, because the text editors simply can't work with such big files efficiently and thus would increase the execution time of the program drastically. On a slower system, the sleep time can be increased via the *–time* configuration or in the python code by increasing the value of the *sleeptime* variable. This is recommended, because otherwise weird behavior can occur, where the program wants to execute keystrokes while the editor isn't fully loaded or closed yet.

3.4 Test Case Structure

The different test cases are the heart piece of the project. As seen in listing 3.4 all different test-cases follow a similar structure. For each test-case a new file gets created and its size is depending on which mode is used to execute the test-case. After each test, all created files get deleted to prevent cluttering.

Listing 3.4: Test Case Structure

```
1  def testcase(name, sleeptime, verbose, mode):
2    create_file(mode)
3    timestamp1 = statextract("testfile")
4
5        #MAKE THE TEST
6
7    timestamp2 = statextract("testfile")
8    compareTimestamps(str(name).upper() + " TESTNAME: ", timestamp1, timestamp2)
9    if (verbose):
10     print_timestamps(timestamp1, timestamp2)
11     print()
12   os.system("rm testfile")
```

How exactly each test is executed and what steps are needed to be simulated highly depends on the used editor, because of that no specific test scenario is shown in listing 3.4.
If the verbose setting is activated when starting the test-case, the file timestamps before and after the test-case are printed to the file like seen in listing 3.5:

Listing 3.5: Verbose output

```
1  NANO ACCESS TEST:  |A| | |
2  Before:
3  Modify: Fri Jun 18 10:11:50 2021 - ns: 624398777
4  Access: Fri Jun 18 10:11:50 2021 - ns: 624398777
5  Change: Fri Jun 18 10:11:50 2021 - ns: 624398777
6  Birth:  Fri Jun 18 10:11:50 2021 - ns: 624398777
7  Inode:  921047
8  After:
9  Modify: Fri Jun 18 10:11:50 2021 - ns: 624398777
10 Access: Fri Jun 18 10:11:50 2021 - ns: 816399302
11 Change: Fri Jun 18 10:11:50 2021 - ns: 624398777
12 Birth:  Fri Jun 18 10:11:50 2021 - ns: 624398777
13 Inode:  921047
```

When creating a new test-case, the given structure can be used. Only the behavior which is need by the editor to execute the test-case needs to be looked up in order to make it work.

3.5 Creating new files

The function *create_file* which is found in the *utility.py* file is responsible for creating new files. When creating a new file for this test-cases, it is important to be able to create different sizes of files. As seen in listing 3.6, two different terminal commands are used to accomplish this. To create an empty file, the *touch* command is used, which creates an empty file. When creating a non empty file, the seq command is used, which creates a sequence of numbers (1-Specified Number) and writes each one on a separate line. This sequence is then piped into the file. If the file doesn't exist at this point in time, it gets automatically created.
The choice was made to create *.c* files, because some file editors need fewer user interaction when opening a file-type which resembles code and thus speeds up the tests.

Listing 3.6: Creating a new file

```
1 def create_file(mode):
2     if (mode == 0):
3         os.system("touch testfile.c")
4     if (mode == 1):
5         os.system("seq 1 > testfile.c")
6     if (mode == 2):
7         os.system("seq 1000000 > testfile.c")
```

3.6 Getting the timestamps

In order to compare the different file timestamps, system utilities needed to be used. A problem arises when trying to use the *stat* command, because it can't display the birth timestamp under ext4. Because of this, the program *prototype_file_ts* by Aurelien Thierry is used, which uses the *statx* syscall which is able to correctly display the Birth timestamp. The *stat* command was only used to acquire the inode number of a given file.

Listing 3.7: Acquiring the timestamps

```
1  def statextract(testfile):
2      Access = ""
3      Modify = ""
4      Change = ""
5      Birth = ""
6      Inode = ""
7
8      output = str(subprocess.check_output(['./prototype_file_ts', testfile]))
9      beginOutput = output.find("M:")
10     output = output[beginOutput + 3::]
11
12     findAccess = output.find("A:")
13     Modify = output[:findAccess - 2]
14     output = output[findAccess + 3:]
15
```

```
16      findChange = output.find("C:")
17      Access = output[:findChange - 2]
18      output = output[findChange + 3:]
19
20      findBirth = output.find("B:")
21      Change = output[:findBirth - 2]
22      Birth = output[findBirth + 3:len(output) - 3]
23
24      output = str(subprocess.check_output(['stat', testfile]))
25      findInode = output.find("Inode")
26      output = output[findInode + 7::]
27      i = 0
28      while output[i] != " ":
29          Inode += output[i]
30          i += 1
31
32      ts = TimeStamp(Access, Modify, Change, Birth, Inode)
33      return ts
```

It may be that there is a way to replicate the behavior of *prototype_file_ts* in Python 3 and thus not needing an external file, but because the program already existed and this project comes bundled in the same git as the *prototype_file_ts*, it was decided to use this program.

4 EVALUATION

The implementation and execution of the project gave a data-set which will be analysed in this chapter. In the first section, the used tools, *strace* and *ldd* are shortly explained. The next section details, what different editors have been used and what version of the file editors has been used for the analysis. Section 4.3 defines an *expected behavior* for each of the four different test-cases in order to make the comparison and analysis of the gained results easier. The next section analysis the observed behavior and is structured in subsection for each test-case. At last in section 4.5 an analysis of *Vim* is discussed on the topic how the editor behaves, when hard-links exists and if the links break or not.

A short version of the findings discussed in this chapter is found in the appendix of the thesis.

The system used, for executing these test is a laptop running the Linux distribution Ubuntu. It has to be noted, that the system is mounted with the *strictatime* option. The exact specs are:

Component / Software	Specifications
Processor	Intel® Core™ i5-7200U CPU @ 2.50GHz × 4
Hard-disk	1TB SSD
RAM	12GB DDR4
BIOS	American Megatrends Inc., v. X756UXK.311
Operating System	Ubuntu 20.04.2 LTS
Partition	378GB, ext4

Table 4.1: System specifications

4.1 Used Tools

In this section, the tools used to gain additional knowledge are shortly discussed and explained.

4.1.1 strace

strace is a command line tool, which offers the tracing of signals and system calls of a given application. To do this, strace acts as an interface between application and system kernel. Each line of the output contains first the name of the executed syscall, which is followed by a list of arguments encapsulated in parenthesis. At the end of the line the return value of the syscall is printed. [26][16][17]
The syscall monitored for the evaluation in this thesis are:

- open
- openat
- close
- read
- write
- rename
- unlink

To achieve this, the following command is used:

Listing 4.1: Used strace command

```
1 strace -o <OutputFileName> -e trace=open,close,read,write,openat,rename,unlink <EditorName>
        <Configs> <Filename>
```

The -o option lets the user specify, to which file the output should be written and the -e option is used for specifying, which syscall and/or signals should be monitored.

4.1.2 ldd

ldd is a command line tool, which is used to get the names of the libraries used by a specified application. In the context of this thesis it is used to get the used libraries of different editors, especially to distinguish, which use *gio*, *Qt* or self written code for the saving of files. To achieve this the following command is used [8]:´

Listing 4.2: Used ldd command

```
1 ldd <applicationname>
```

4.2 Used Editors

To be able to accurately represent the findings, it is necessary to know, which version of which file editor was used. This is particularly helpful, when different version of the same editor are getting tested. The used

editors and the number of the installed version can be found in table 4.2.

Editor	Version
Vim	8.1
Nano	4.8
Gedit	3.36.2
Kate	19.12.3
Atom	1.55.0
Emacs	26.3
Geany	1.36
Sublime Text	3211
Codeblocks	20.03
Bluefish Text	2.2.11
Texstudio	2.12.22
Visual Studio Code	1.56.2
Leafpad	0.8.18.1
Notepadqq	1.4.8
Jed	0.99.19

Table 4.2: List of used editors and their version numbers

The *Vim* editor is used in two different test-cases. To have a better understanding on how also different settings of the *vimrc* affect the behavior of the file editor, one test is done with the *–clean* run configuration, which sets the editor in a state similar to a fresh install without any modifications. The other configuration is the *set nowritebackup* configuration, where the line *set nowritebackup* is added to the *vimrc* file of this editor. This should prevent *Vim* from modifying another file and because of this directly modify the testfile.

4.3 Expected Behaviour

For easier comparison of the different results, an expected behavior is defined for each of the four different test-cases.
For the *Access Test* the editor will open the file and then close it. Because of that, the expected behavior is that only the *Access* timestamp is modified.
The next test-case is the *Modify Test*. In this test, a file gets opened by the editor and some text gets written into it. After that the editor performs the save action and closes the file. For this case the expected behavior is the modification of *Modify*,*Access* and *Change* timestamps. The *Change* timestamp gets updated every time the *Modify* timestamp gets updated and because of that it is modified in this test-case.
For the *Save no Modify Test*, the text editor will open a file, perform the save action and close the file. Because there is no new data to be saved, the file contents shouldn't be modified. Therefore the expected result is a modification of the *Access* timestamp.
The last test is the *Modify no Save Test*. For this test, the editor will open the file, write some text into the file but then doesn't save the modification and closes the file. Like in the *Save no Modify Test*, no modification to the file contents should have occurred and because of that the only timestamp modified should be the *Access* timestamp.

Test name	Modify	Access	Change	Birth	Inode
Access Test		X			
Modify Test	X	X	X		
Save no Modify Test		X			
Modify no Save Test		X			

Table 4.3: Expected Timestamp Modifications

As seen in table 4.3 the expected behavior is the same for the *Access Test*, *Save no Modify Test* and *Modify no Save Test*. Furthermore it is interesting that the defined *expected behavior* sees no modification of the *Birth* timestamp and no changes to the *Inode* number as well.

4.4 Found Behaviour

The following part is structured in the different test cases. The data shown is generated with the editors and their respective version shown in table 4.2 and on a system with the specifications shown in table 4.1. If a different system or different versions of the editors are used, it is possible that the gained results will differ from the results discussed in the following. Furthermore it has to be noted that the results were gained when using the setting *verbose* and all editors and file sizes were tested in one run of the code.

4.4.1 Access

Editor	Modify	Access	Change	Birth	Inode
vim –clean		X			
vim set nowritebackup		X			
nano		X			
gedit		X			
kate		X			
atom		X			
emacs		X			
geany		X			
sublime		X			
codeblocks non empty		X			
bluefish		X			
texstudio		X			
visual studio code		X			
leafpad		X			
notepadqq		X			
jed		X			
codeblocks empty					

Table 4.4: Access timestamp modifications

When looking at table 4.4 it is easy to see, that almost all file editors exhibit the expected behavior and have modified only the *Access* timestamp. The only abnormality is the behavior of the *codeblocks* editor when running its *Access Test* on an empty file.

4.4.1.1 Codeblocks on an empty file

As seen before in table 4.4 the *codeblocks* editor exhibits an alternating behavior depending on the size of the file. The *Access Test* on a non empty file exhibits the normal behavior, but on an empty file, no timestamps, not even the *Access* timestamp got modified. When running the test-case with the *–verbose* configuration the timestamps shown in table 4.5 can be gathered.

Before:		**After:**	
Modify:	Tue May 25 17:05:28 2021 - ns: 707626252	Modify:	Tue May 25 17:05:28 2021 - ns: 707626252
Access:	Tue May 25 17:05:28 2021 - ns: 707626252	Access:	Tue May 25 17:05:28 2021 - ns: 707626252
Change:	Tue May 25 17:05:28 2021 - ns: 707626252	Change:	Tue May 25 17:05:28 2021 - ns: 707626252
Birth:	Tue May 25 17:05:28 2021 - ns: 707626252	Birth:	Tue May 25 17:05:28 2021 - ns: 707626252
Inode:	933966	Inode:	933966

Table 4.5: codeblocks empty file access test

It is really interesting to see, that it looks like the program really doesn't even access the file, because all timestamps are the same as the time of the file creation. To further understand what happens, the test-case gets executed with strace running in the background to investigate what syscalls are getting executed.
When searching through the output file for the filename *testfile.c* which was the file used in the *Access Test* it is to note, that no matches are found. Either the editor opens the file on another way or the file simply isn't accessed.
Because of this strange behavior another instance of strace executed, this time without a filter on the used syscalls and signals. For better readability, in the listing 4.3 only the syscalls which use the file *testfile.c* are shown.

Listing 4.3: codeblocks empty access test

```
1 execve("/usr/bin/codeblocks", ["codeblocks", "./testfile.c"], 0x7ffff2d39478 /* 59 vars */)
      = 0
2 ...
3 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
      S_IFREG|0664, st_size=0, ...}) = 0
4 ...
5 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
      S_IFREG|0664, st_size=0, ...}) = 0
6 ...
7 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
      S_IFREG|0664, st_size=0, ...}) = 0
8 ...
9 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
      S_IFREG|0664, st_size=0, ...}) = 0
10 access("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", W_OK) = 0
```

```
11 ...
12 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
       S_IFREG|0664, st_size=0, ...}) = 0
13 ...
14 access("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", W_OK) = 0
15 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
       S_IFREG|0664, st_size=0, ...}) = 0
16 ...
17 stat("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c", {st_mode=
       S_IFREG|0664, st_size=0, ...}) = 0
18 ...
```

As seen in listing 4.3, there are three different syscalls using the file *testfile.c*.

Syscall	Description
execve	Executes a program, which is defined by a given path name
stat	Displays the file status
access	Checks if a given process can access the file defined in path-name

It's interesting to see that the file gets only used for *access* and *stat* and nowhere else. It could be that codeblocks launches a separate process, which does something with the testfile.
In order to either prove or disprove this assumption, the strace command gets run again, this time with the flag *-f* which also follows child processes. This time multiple other calls using *testfile.c* can be found. Most importantly this time there are *openat* syscalls using *testfile.c*. Below is a excerpt of the *strace* output shown in listing 4.4.

Listing 4.4: codeblocks empty access with strace -f

```
1 openat(AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c",
       O_RDONLY) = 15
2 fstat(15, {st_mode=S_IFREG|0664, st_size=0, ...}) = 0
3 read(15, "", 0)                       = 0
4 close(15)                             = 0
```

As seen in listing 4.4, the *openat* syscall opens the file and gives it the number 15. Afterwards *fstat* is used on *testfile.c* and in the extracted data the file size can be seen. Thereafter the *read* syscall is used. It's interesting to see that read gets the correct size of the file and thus attempts to read 0 bytes. For comparison, the same part has been extracted from the *codeblocks access small file test* and is presented in figure 4.5.

Listing 4.5: codeblocks small access with strace -f

```
1 openat(AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c",
       O_RDONLY)              = 20
2 fstat(20, {st_mode=S_IFREG|0664, st_size=2, ...}) = 0
3 read(20, "1\n", 2)        = 2
4 close(20)                             = 0
```

It is to note, that this snippet is altered in order to make it more readable. Normally the *strace* output would have been not consecutive and other syscalls would have interrupted the syscalls shown in figure 4.5.

As seen in this excerpt, the file size is 2 bytes as signaled by the *st_size=2* argument in line 2. In this previous output file this part was 0 because the file size was 0 bytes. Now in line 3, the *read* syscall has a count of 2 and thus reads the file and modifies its access timestamp.
So now it has been cleared that the behavior of *codeblocks* differs between file sizes, because it one time executes the *read* syscall with a count of 0 and one time with a count of 2. How *read* handles this two different cases will be explained below. The next question is why does *codeblocks* behavior differs from the other editors. To answer this, the *strace* output of the *vim* editor gets analysed with the *–clean* configuration, which is shown an excerpt of in listing 4.6.

Listing 4.6: vim –clean empty access with strace -f

```
1 openat(AT_FDCWD, "./testfile", O_RDONLY) = 3
2 ...
3 read(3, "", 8192)                 = 0
4 close(3)                          = 0
```

As seen in listing 4.6 the *vim* editor doesn't look up the file size and tries to read 2^{13} bytes of the file.
So the difference between at least the behaviors of *vim* and *codeblocks* is, that *codeblocks* tries to read the whole file at once and *vim* tries to read the file in chunks, thus performing the *read* syscall on an empty file with a count higher than 0.
It is important to note, that the behavior of the *write* syscall changes depending on the number of Bytes which should be read. As stated in the man pages "a read() with a count of 0 returns zero and has no other effects" [12]. Furthermore the POSIX specification states "Upon successful completion, where nbyte is greater than 0, read() shall mark for update the last data access timestamp of the file, and shall return the number of bytes read." [24].
Most of the time, when an application reads a file, it happens in chunks. In the case of *vim* in chunks of the size of 8192 bytes. When running the command *stat -f .* on the evaluation system the output shows, that the block size is 4096 byte. So *vim* reads 2 blocks of the size 4096 bytes until the whole file has been read.

4.4.2 Modify

Editor	Modify	Access	Change	Birth	Inode
vim –clean	X	X	X	X	X
gedit	X	X	X	X	X
emacs	X	X	X	X	X
geany	X	X	X	X	X
codeblocks empty	X	X	X	X	X
codeblocks non empty	X	X	X	X	X
bluefish	X	X	X	X	X
texstudio	X	X	X	X	X
jed	X	X	X	X	X
vim set nowritebackup	X	X	X		
nano	X	X	X		
kate	X	X	X		
atom	X	X	X		
sublime	X	X	X		
visual studio code	X	X	X		
leafpad	X	X	X		
notepadqq	X	X	X		

Table 4.6: Modify timestamp modifications

After Evaluating the Output of the test, two kinds of editors could be found. One type, which edits the original file and another type, which builds a new file out of a swap-file. Some editors can do both, depending on their configuration. Namely this is the case for the vim text editor, who in its standard configuration (or run with "–clean") creates a swap-file and build with this swap-file the new file. If vim is run with a "set nowritebackup" configuration, no swap-file is created and thus vim modifies the original file.

When looking at the output of the "Modify" test, a pattern can be observed. Either the Modify, Access and Change timestamps, or the Modify, Access, Change, Birth and Inode timestamps get modified by the editor. For some editors, this behavior is dictated by a used library, often *glib* or *Qt*, and in some case it depends on its own code.
An example for a program which uses self written code and none of these libraries is vim. Because we have both behaviors, depending on the config, it is a good choice to explore the behavior on a syscall level.

4.4.2.1 Comparing vim –clean and vim set nowritebackup of empty files based on syscalls

To be able to compare what syscalls each vim configuration makes, the tool "strace" is utilized. For the upcoming output snippets, the syscalls read, write, openat, rename and unlink got monitored. Below in listing 4.7 the output of the strace command for *vim –clean* is shown and in table 4.7 the corresponding timestamp modifications are shown.´

Listing 4.7: vim –clean

```
1 ...
2 openat(AT_FDCWD, "testfile", O_RDONLY)  = 3
3 openat(AT_FDCWD, ".testfile.swp", O_RDONLY) = -1 ENOENT (No such file or directory)
4 openat(AT_FDCWD, ".testfile.swp", O_RDWR|O_CREAT|O_EXCL, 0600) = 4
5 openat(AT_FDCWD, ".testfile.swpx", O_RDONLY) = -1 ENOENT (No such file or directory)
6 openat(AT_FDCWD, ".testfile.swpx", O_RDWR|O_CREAT|O_EXCL, 0600) = 5
7 ...
8 unlink(".testfile.swpx")                = 0
9 ...
10 unlink(".testfile.swp")                = 0
11 ...
12 unlink("testfilz~")                    = -1 ENOENT (No such file or directory)
13 rename("testfile", "testfilz~")        = 0
14 ...
15 openat(AT_FDCWD, "testfile", O_WRONLY|O_CREAT, 0664) = 3
16 write(3, "lorem ipsum\n", 12)          = 12
17 close(3)                               = 0
18 ...
19 unlink("testfilz~")                    = 0
20 ...
```

Before:		**After:**	
Modify:	Tue May 25 23:16:46 2021 - ns: 453171258	Modify:	Tue May 25 23:16:53 2021 - ns: 217149314
Access:	Tue May 25 23:16:46 2021 - ns: 453171258	Access:	Tue May 25 23:16:53 2021 - ns: 217149314
Change:	Tue May 25 23:16:46 2021 - ns: 453171258	Change:	Tue May 25 23:16:53 2021 - ns: 221149301
Birth:	Tue May 25 23:16:46 2021 - ns: 453171258	Birth:	Tue May 25 23:16:53 2021 - ns: 217149314
Inode:	920996	Inode:	920998

Table 4.7: vim –clean timestamp modifications

As seen in line 12, the program checks if a file with the name "testfilz~" exists, if now such file exists, the original file "testfile" gets renamed to "testfilz~". The rename here works like a moving of the file out of the way to make place for creating a new file. In line 15 comes an "openat" syscall with the "O_CREAT" flag, which tries to open "testfile" and if it doesn't exists, creates a new file with the given name. After that, the text gets written in the new file and the "old" file gets unlinked.
Below in listing 4.8 the same test case monitored as in listing 4.7, but this time for the *vim set nowritebackup* configuration. Table 4.8 shows the corresponding timestamp modifications.

Listing 4.8: vim set nowritebackup

```
1 ...
2 openat(AT_FDCWD, "testfile", O_RDONLY)  = 3
3 openat(AT_FDCWD, ".testfile.swp", O_RDONLY) = -1 ENOENT (No such file or directory)
4 openat(AT_FDCWD, ".testfile.swp", O_RDWR|O_CREAT|O_EXCL, 0600) = 4
5 openat(AT_FDCWD, ".testfile.swpx", O_RDONLY) = -1 ENOENT (No such file or directory)
6 openat(AT_FDCWD, ".testfile.swpx", O_RDWR|O_CREAT|O_EXCL, 0600) = 5
7 ...
```

```
 8 unlink(".testfile.swpx")                = 0
 9 ...
10 unlink(".testfile.swp")                 = 0
11 ...
12 openat(AT_FDCWD, "testfile", O_WRONLY|O_CREAT, 0664) = 3
13 write(3, "lorem ipsum\n", 12)           = 12
14 close(3)                                = 0
15 ...
```

Before:		**After:**	
Modify:	Tue May 25 23:17:13 2021 - ns: 417090509	Modify:	Tue May 25 23:17:20 2021 - ns: 237072881
Access:	Tue May 25 23:17:13 2021 - ns: 417090509	Access:	Tue May 25 23:17:13 2021 - ns: 581090073
Change:	Tue May 25 23:17:13 2021 - ns: 417090509	Change:	Tue May 25 23:17:20 2021 - ns: 241072871
Birth:	Tue May 25 23:17:13 2021 - ns: 413090521	Birth:	Tue May 25 23:17:13 2021 - ns: 413090521
Inode:	920996	Inode:	920996

Table 4.8: vim set nowritebackup timestamp modifications

As seen in listing 4.8, if the "set nowritebackup" configuration is used, the original file isn't renamed and thus the text gets directly written to the original file.

4.4.2.2 Swap-files vs. direct modification

As stated by the "*Linux Weekly News LWN*" in its article "Ghosts of Unix past, part 4: High-maintenance designs" from 2010, "Editors need to take special care of linked files. It is generally safer to create a new file and rename it over the original rather than to update the file in place. When a file has multiple hard links it is not possible to do this without breaking that linkage, which may not always be desired." [6] This could explain the difference in behaviour, experienced in the "Modify Test" as this is the exact behavior observed on the syscall layer.

In the end it boils down to a assessment of priorities. Not updating the file in place and renaming it to build it from a new file may be a more robust solution, but it comes at the cost of temporary higher space usage, while on the other hand the update in place uses less space, but is less robust.
Below in listing 4.9 a part of the strace output of the *Modify Test* on Geany is shown.

Listing 4.9: strace geany

```
1 ...
2 openat(AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c",
      O_WRONLY|O_CREAT|O_EXCL|O_CLOEXEC, 0666) = -1 EEXIST (File exists)
3 openat(AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c",
      O_WRONLY|O_CREAT|O_NOFOLLOW, 0666) = 18
4 ...
5 openat(AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/.goutputstream-
      ELP740", O_WRONLY|O_CREAT|O_EXCL, 0666) = 19
6 ...
7 close(18)                                = 0
```

```
 8 ...
 9 write(19, "lorem ipsum dolor\n", 18)    = 18
10 fsync(19)                              = 0
11 rename("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/.goutputstream-ELP740", "/
       home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.c") = 0
12 fstat(19, {st_mode=S_IFREG|0664, st_size=18, ...}) = 0
13 close(19)                              = 0
14 ...
```

For example as seen in listing 4.9, *Geany* creates a new file called "*.goutputstream-ELP740*" to which it writes the changes. Afterwards, the file gets renamed to "*testfile.c*" effectively overwriting the old file.
In comparison, the behavior of *vim –clean* as seen in listing 4.7 is really close to that. Instead of directly overwriting the "old file" like *geany* does, the "old file" gets moved out of the way through a *rename* syscall, the new file is getting the original name of the "old file" and the "old file" than gets unlinked.
A completely other behavior is shown by *vim set nowritebackup*. As seen in listing 4.8 the original file is preserved and the changes get directly written to it. No other files get used to save the changes and no files get renamed in the process.

4.4.2.3 Comparing which editors used GIO and Qt

Now it is also possible to compare the observed behavior and the used libraries to see if there is a pattern to be found. Through the usage of the Linux command line tool ldd, for some of the editors the used libraries can be found.

Editor	Library	Modify	Access	Change	Birth	Inode
bluefish	gio	X	X	X	X	X
codeblocks	gio	X	X	X	X	X
emacs	gio	X	X	X	X	X
geany	gio	X	X	X	X	X
gedit	gio	X	X	X	X	X
texstudio	qt	X	X	X	X	X
kate	qt	X	X	X		

Table 4.9: Behavior matched with used library

As seen in Table 4.9 all editors using the GIO library show the same behavior regarding file timestamp changes, by changing all timestamps and creating a new file when saving changes made to a file.
Surprisingly, the two editors found to use the Qt library don't seem to exhibit the same behavior.

In the following the difference in behavior between *texstudio* and *Kate* is explored in order to find the reason for the differing behavior.

Firstly the behavior of *texstudio* is analyzed. When looking with strace at what texstudio does when modifying a file a rather interesting behavior can be found.
Below in listing 4.10 an excerpt of the output can be found.

Listing 4.10: strace texstudio modify

```
1 openat(AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code", O_RDWR|
      O_CLOEXEC|O_TMPFILE, 0600) = 31
2 ...
3 write(31, "lorem ipsum", 11)      = 11
4 ...
5 linkat(AT_FDCWD, "/proc/self/fd/31", AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/
      bachelor-thesis/Code/testfile", AT_SYMLINK_FOLLOW) = -1 EEXIST (File exists)
6 linkat(AT_FDCWD, "/proc/self/fd/31", AT_FDCWD, "/home/niclas/PycharmProjects/Thesis/
      bachelor-thesis/Code/testfile.KpmIxH", AT_SYMLINK_FOLLOW) = 0
7 close(31)                          = 0
8 rename("/home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile.KpmIxH", "/home/
      niclas/PycharmProjects/Thesis/bachelor-thesis/Code/testfile") = 0
```

When looking at listing 4.10, in the first line a temporary file gets created in the directory */home/niclas/PycharmProjects/Thesis/bachelor-thesis/Code* with the usage of the *0_TMPFILE* flag. As written in the man pages *O_TMPFILE* "creates an unnamed temporary regular file" [9]. The newly created file gets the file descriptor 31. Afterward the modification gets written by the *write* syscall to the newly created file. But now, the program tries to create a hard link between the file descriptor 31 and the original file. This fails, because the file already exists. Thereafter the editor tries it again, but this time between the file descriptor 31 and a new file called "*testfile.KpmIxH*" which works, because the file doesn't exist. Now a hardllink between the new file and the save point of the original file exists. Afterwards the *close* syscall is executed. Now the same behavior can be observed as seen in listing 4.9. The new file gets renamed to the name of the old file and thus overwrites it. In this case, the name gets changed from "*testfile.KpmIxH*" to "*testfile*". This overwriting of the old file replaces all the timestamps with the timestamps of the new file and because of this a new *Birth* timestamp can be observed.
Now when looking into the source code of *texstudio* the function *QEditor::saveCopy* is used, which can either use *QSaveFile* or *QEditor::writeToFile*. When looking at the description of *QSaveFile*, the Qt Documentation states "While writing, the contents will be written to a temporary file, and if no error happened, commit() will move it to the final file. This ensures that no data at the final file is lost in case an error happens while writing, and no partially-written file is ever present at the final location" [11].
This exactly describes the behavior of what has been observed with the strace of *texstudio* in listing 4.10.
When looking into the behavior of *Kate*, Berenger states in his thesis, that *Kate* uses for saving files not the *Qt* framework but the *KIO* library which is part of the *KDE* framework [21].
Because of this reason the behavior of the *texstudio* file editor and *Kate* differ from each other.

4.4.3 Save no Modify

Below the results of the *Save no Modify* are shown in table 4.10.

Editor	Modify	Access	Change	Birth	Inode
vim –clean	X	X	X	X	X
gedit	X	X	X	X	X
bluefish	X	X	X	X	X
vim set nowritebackup	X	X	X		
nano	X	X	X		
kate	X	X	X		
atom	X	X	X		
sublime	X	X	X		
visual studio code	X	X	X		
notepadqq	X	X	X		
emacs		X			
geany		X			
codeblocks non empty		X			
texstudio		X			
leafpad		X			
jed		X			
codeblocks empty					

Table 4.10: Save no Modify timestamp modifications

As seen in table 4.10 there are two groups of behavior to notice.
One group handles the test-case as if a modification has occurred and therefor exhibits the same behavior, the editor showed in the Modify test-case.
The other group doesn't perform a save action. Therefore only an Access gets noticed and thus only the access timestamp gets updated.

4.4.4 Modify don't Save

Below the results of the *Modify don't Save Test* are shown in table 4.11.

Editor	Modify	Access	Change	Birth	Inode
vim –clean		X			
vim set nowritebackup		X			
nano		X			
gedit		X			
kate		X			
atom		X			
emacs		X			
geany		X			
sublime		X			
codeblocks non empty		X			
bluefish		X			
texstudio		X			
visual studio code		X			
leafpad		X			
notepadqq		X			
jed		X			
codeblocks empty					

Table 4.11: Modify dint Save timestamp modifications

Looking at the results of the Modify don't Save test, the only unexpected result is again the behavior of the editor codeblocks on an empty file.

4.5 Testing linkage breaks through different modification strategies

To test how different modification strategies behave when there are hard-links, the vim editor was chosen as a test subject, because it features settings in which the two different modification strategies are exhibited. The used settings are on the one hand "–clean" and the "set nowritebackup" configuration. Via the "touch" command, a new file was created and with the "ln" command, the hard-link between the two files was created.
Thereafter, 10 modifications per file and configuration where executed. After each modification, the files where tested for differences in their data. The following table shows the results of this test.

Modification Number	–clean	set nowritebackup
1	no breakage	no breakage
2	no breakage	no breakage
3	no breakage	no breakage
4	no breakage	no breakage
5	no breakage	no breakage
6	no breakage	no breakage
7	no breakage	no breakage
8	no breakage	no breakage
9	no breakage	no breakage
10	no breakage	no breakage

Table 4.12: Manual breakage test on hard-linked files

As seen, the hard-link persists, even if the inode number should be changed. This is particularly interesting, because when run with the *–clean* configuration *vim* should change the inode of the file and both files shouldn't have the same inode.
To test this, a new file gets created and hard-linked to an other file. After the creation both inodes get checked. After the modification of one of the files the inode numbers get checked again.

File	Inode Before	Inode After
file	921038	921038
linked file	921038	921038

Table 4.13: Result of the Inode Test

As seen in figure 4.13 interestingly the Inode numbers don't get changed. To achieve this, *vim –clean* shouldn't create a new file, or create a new file with the same Inode number as before.
To check what is happening, the test gets repeated, but this time the timestamps get monitored with the *prototype_file_ts* program. This results in the following timestamp changes.

File	Modify	Access	Change	Birth	Inode
file	X	X	X		
linked file	X	X	X		

Table 4.14: Timestamp modification with multiple Hard-links

When comparing figures 4.14 and 4.6 the behavior of *vim –clean* has changed and now resembles the results of *vim set nowritebackup*. This means, that somewhere *vim* checks if there are hard-links linking to the file. If one or multiple hard are found, the behavior gets automatically changed.

5 Related Work

In this chapter different related scientific work is discussed, in order to put the findings of the last chapter into a scientific context.

The first work to be discussed is directly related to this thesis, because it is part of the same project. Dr. Aurelien Thierry already concluded work on the topic of "*MACB Timestamps across POSIX implementations (Linux, OpenBSD, FreeBSD)*". His work explores the behavior of file timestamps on different operation systems and if they correspond the *POSIX* specifications. Furthermore the impact of different mount option between Linux, OpenBSD and FreeBSD, the different timestamp resolutions and the behavior of common operations like *touch*, *mkdir*, *chmod* are explored. The difference to this thesis is the part of the system that is looked into. Where Dr. Thierrys looks at a operation system specific layer, this work looks at how different applications modify timestamps using different libraries and syscalls. [20]

This next scientific work is also part of the same project. Berenger Temgoua Dibanda concluded his thesis on the topic of "*Timestamp Forensics on graphical middle-ware/libraries on Linux*". In his work he looked into how the usage of different libraries, namely *GIO* and *Qt* affect the modification of file timestamps. To accomplish this, he looks into how basic behavior is executed by the different libraries and how this affects the file timestamps. Furthermore he looked into graphical applications and also analysed the behavior of the file editors *Kate* and *Gedit*, which are also analysed in this thesis. The main difference to this thesis is again the layer which is looked at. His focus lies on the middle-ware and libraries whereas this thesis looks at the applications which often use these libraries. As seen here, booth works partly analyze the same topic. [21]

The third scientific work is by Felix C. Freiling, Andreas Dewald and Sven Kälber and has the title "*Forensic*

Application-Fingerprinting based on File System Metadata". It investigates, how file system metadata can be used to reconstruct events of a system and how this data can be used to fingerprint different applications. Furthermore an approach to automatically reconstruct events is discussed. One of the key differences is the aim of the scientific works. This thesis only explores how different applications alter the file timestamps whereas the scientific work by Freiling, Dewald and Kälber discusses a much more extensive fingerprinting system. [27]

6 Conclusion and Future Work

Timestamp forensics are and will be an important part of today's computer forensic field. To be able to accurately reconstruct which events lead to the current state of a system, the knowledge about what actions in different applications produce what results is indispensable.
Therefore this thesis investigated the behavior of a broad range of file editors on how they affect the MACB timestamps of the files they are working on. To fulfill this goal, a program was developed, which is able to automatically run profiling test on the given file editors while mimicking user input as best as possible.
Through the usage of the aforementioned program a broad data set was created and analyzed in order to understand the behavior of different applications in different use cases.
After grouping all the results, a short cheat sheet was created to make the lookup of the different behaviors easy and fast.

But despite all the work and results accomplished in this thesis the research on this topic is far from finished as there are many factors that can change the observed behavior. In the following some possible future work will be highlighted in order to show what could possibly be researched.
One of the main goals of the thesis was to make it as expandable and customizable as possible, because there are many more applications which can be tested for their behavior. The easiest expansion could be made by adding support for more file editors. Furthermore it will be interesting to see how different versions of the supported editors, other operating systems or other operating system version affect the results gained in this thesis.
Another aspect that was explored in this thesis is how hard links affect the behavior of other file editors then *vim*. Further investigation in this topic could aim to explain how different editors handle the occurrence of hard links. Does the applications behavior change in order to protect the hard links or could it happen,

that an application breaks the connection between file and hard linked file and doesn't restore the hard link. Further research could also try to explore how not only the timestamps of the files which the editors modify get changed, but also the timestamps of other files, which the editors use or access to function properly like fonts, configs or other system files. With the knowledge gained from this, it could be possible to gain enough unique behavior in order to accurately fingerprint actions taken by different applications which would aid in the reconstruction of events.

6 BIBLIOGRAPHY

[1] Windows forensic analysis. URL https://www.sans.org/security-resources/posters/windows-forensic-analysis/170/download.

[2] Syscalls. URL https://man7.org/linux/man-pages/man2/syscalls.2.html.

[3] Ext4 filesystem, . URL https://static.usenix.org/event/lsf07/tech/cao_m.pdf.

[4] Ext4 filesystem, . URL https://en.wikipedia.org/wiki/Ext4.

[5] File timestamps. URL https://linuxreviews.org/File_timestamps.

[6] Ghosts of unix past, part 4: High-maintenance designs. URL https://lwn.net/Articles/416494/.

[7] The gnu c library (glibc). URL https://www.gnu.org/software/libc/.

[8] URL https://man7.org/linux/man-pages/man1/ldd.1.html.

[9] openat. URL https://man7.org/linux/man-pages/man2/openat.2.html.

[10] Protection ring. URL https://en.wikipedia.org/wiki/Protection_ring#/media/File:Priv_rings.svg.

[11] Qsavefile class. URL https://doc.qt.io/qt-5/qsavefile.html#details.

[12] read, . URL https://man7.org/linux/man-pages/man2/read.2.html.

[13] pyautogui, . URL https://pyautogui.readthedocs.io/en/latest/.

[14] rename. URL https://man7.org/linux/man-pages/man1/rename.1.html.

[15] Ext4 timestamps. URL https://www.researchgate.net/figure/Available-timestamps-in-the-ext4-inode-structure_tbl1_323926417.

[16] strace, . URL https://linuxwiki.de/strace.

[17] strace(1) — linux manual page, . URL https://man7.org/linux/man-pages/man1/strace.1.html.

[18] unlink. URL https://linux.die.net/man/2/unlink.

[19] write. URL https://man7.org/linux/man-pages/man2/write.2.html.

[20] Aurélien Thierry. Mac(b) timestamps across posix implementations (linux, openbsd, freebsd). URL https://medium.com/@quoscient/mac-b-timestamps-across-posix-implementations-linux-openbsd-freebsd-1e2d5893e4f.

[21] Berenger Temgoua Dibanda. *Timestamp Forensics on graphical middleware/libraries on Linux*. Bachelor's thesis, Friedrich Alexander Universität, Erlangen, 2020.

[22] Bundesamt für Sicherheit in der Informationstechnik. Leitfaden it-forensik. URL https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Cyber-Sicherheit/Themen/Leitfaden_IT-Forensik.pdf?__blob=publicationFile&v=2.

[23] Ext4 Timestamps. URL https://www.sans.org/blog/understanding-ext4-part-2-timestamps/.

[24] The Open Group IEEE Computer Society. Ieee standard for information technology— portable operating system interface (posix ®) base specifications, issue 7.

[25] Interpol. Digital forensics. URL https://www.interpol.int/How-we-work/Innovation/Digital-forensics.

[26] Jürgen Wolf. Linux-unix-programmierung, 2006. URL https://openbook.rheinwerk-verlag.de/linux_unix_programmierung/Kap17-006.htm.

[27] Sven Kalber, Andreas Dewald, and Felix C. Freiling. Forensic application-fingerprinting based on file system metadata. pages 98–112. doi: 10.1109/IMF.2013.20.

[28] Patrick Kral. The incident handlers handbook. 2011. URL https://www.sans.org/reading-room/whitepapers/incident/incident-handlers-handbook-33901.

[29] LinuxBnB. Adding a system call to linux (arm architecture). URL https://www.linuxbnb.net/home/adding-a-system-call-to-linux-arm-architecture/.

[30] Loris Degioanni. The fascinating world of linux system calls, 2014. URL https://sysdig.com/blog/fascinating-world-linux-system-calls/.

[31] packagecloud. The definitive guide to linux system calls. URL https://blog.packagecloud.io/eng/2016/04/05/the-definitive-guide-to-linux-system-calls/.

[32] Parliamentary Office of Science and Technology. Digital forensics and crime. URL https://researchbriefings.files.parliament.uk/documents/POST-PN-0520/POST-PN-0520.pdf.

[33] Simon Kemp. Digital 2019: Global internet use accelerates. URL https://wearesocial.com/blog/2019/01/digital-2019-global-internet-use-accelerates.

7 APPENDIX

Editor Name	Access Test	Modify Test	Save no Modify Test	Modify don't Save Test
vim –clean	A	MACBI	MACBI	A
gedit	A	MACBI	MACBI	A
bluefish	A	MACBI	MACBI	A
emacs	A	MACBI	A	A
geany	A	MACBI	A	A
codeblocks non empty	A	MACBI	A	A
texstudio	A	MACBI	A	A
jed	A	MACBI	A	A
codeblocks empty		MACBI		
vim set nowritebackup	A	MAC	MAC	A
nano	A	MAC	MAC	A
kate	A	MAC	MAC	A
atom	A	MAC	MAC	A
sublime	A	MAC	MAC	A
visual studio code	A	MAC	MAC	A
notepadqq	A	MAC	MAC	A
leafpad	A	MAC	A	A

Table 7.1: All Testcases

List of Tables

List of Figures

Listings