

Doctoral thesis

Doctoral theses at NTNU, 2024:115

Rune Nordvik

Interpretation of File System Metadata in a Criminal Investigation Context

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of
Philosophiae Doctor
Faculty of Information Technology and Electrical
Engineering
Dept. of Information Security and
Communication Technology



Norwegian University of
Science and Technology

Rune Nordvik

Interpretation of File System Metadata in a Criminal Investigation Context

Thesis for the Degree of Philosophiae Doctor

Gjøvik, April 2024

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology



Norwegian University of
Science and Technology

NTNU

Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

© Rune Nordvik

ISBN 978-82-326-7824-2 (printed ver.)
ISBN 978-82-326-7823-5 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2024:115

Printed by NTNU Grafisk senter

Abstract

The reliable reconstruction of digital events is imperative for solving criminal cases. Computers, servers, mobile and IoT devices, vehicles, and EV charging infrastructure all use either local or remote storage (cloud). The storage needs to use a file system in order to store and retrieve files. Currently, digital forensic tools implement support for the most popular file systems, either fully or partially. In order to determine what has taken place, investigators today are dependent on tools that automate much of the investigation. Unfortunately, these tools use techniques that are not necessarily published, tested or peer-reviewed, which increases the uncertainty of their results. Furthermore, investigators normally use well known artifacts from the Operating System (OS) when trying to determine what occurred, however, file system interpretation is often automated by the tools and trusted as reliable and complete by the investigators. In many cases the OS is not available, for instance, when an external storage device is seized. This means the investigator only has the file system and the file content available for investigation. We found metadata structures that may connect an external device to the computers used to create files on the device, which order these files have been created, and when the computers were booted. These findings will help investigators to identify which computers are relevant for the investigation, create timelines, and detect timestamp manipulation, but also identify which files users have created, opened, or saved. It is not unusual that external devices are damaged or reformatted with new file systems. In this context it is important to be able to recover files from the damaged file system. We were able to invent a novel and generic method to carve and identify metadata for files using equality or approximate equality to identify timestamps that are co-located, a pattern typical for file metadata structures in most file systems. Our prototype tool outperforms the other tools we tested in recovery from damaged file systems. Investigators often use timestamps to create timelines

or to limit their investigation to a particular time frame. We found that both tools and different file system drivers are implemented differently, not necessarily following the file system specifications. Even normal usage of an external USB disk on multiple operating systems may change timestamps to invalid settings, and it is imperative that investigators are able to identify such usage. This thesis will focus on interpreting the file system metadata to identify and understand the accurate meaning of structures that the digital forensic tools currently do not support or only partly support, identifying new knowledge that will increase the quality of digital investigations.

Preface

This dissertation is submitted in partial fulfillment of the requirements for the degree of Philosophiae Doctor (PhD) at the Norwegian University of Science and Technology (NTNU).

The presented work was carried out at the Faculty of Information Technology and Electrical Engineering, Department of Information Security and Communication Technology (IIK) at NTNU from 2018 until 2023, and comments from the PhD committee were amended in February 2024. The work was supervised by Prof. Dr. Stefan Axelsson, Dr. Fergus Toolan, and Associate Prof. Dr. Geir Olav Dyrkolbotn.

This research received funding from the Research Council of Norway programme IKTPLUS, under the R&D project “Ars Forensic”, grant agreement 248094/O70.

Acknowledgements

I would first like to sincerely thank my employer, the Norwegian Police University College (PHS), for allowing me to use time to complete this PhD study. I would like to thank my main supervisor Prof. Dr. Stefan Axelsson for his continues support and all the good talks, reviews, advises and contributions. I would also like to thank Dr. Fergus Toolan for his good reviews and contributions, and for being a very good colleague at PHS and a friend. In addition, I would also like to thank my supervisors, and Dr. Radina Stoykova, Dr. Kyle Porter, and Prof. Dr. Katrin Franke for the good cooperation and discussions in the process of creating and publishing scientific papers. I must mention Dr. Georgina Louise Humphries, especially for being a very good colleague at PHS and a friend. She has helped me significantly with reviews and fixed language issues of both scientific papers and for this PhD thesis.

Last, but not least I would like to thank the love of my life Veronica Tøvik for her continues support during this PhD study. She has always been there for me, no matter how busy I have been.

Publications

This thesis is based on the following publications

Paper A *Using the object ID index as an investigative approach for NTFS file systems*, Rune Nordvik, Fergus Toolan and Stefan Axelsson. In: *Digital Investigation Volume 28, Supplement*. April 2019, Pages S30-S39. DFRWS 2019 Europe — Proceedings of the 19th Annual DFRWS Conference. DOI: <https://doi.org/10.1016/j.diin.2019.01.013>

Paper B *Generic Metadata Time Carving*, Rune Nordvik, Kyle Porter, Fergus Toolan and Stefan Axelsson and Katrin Franke. In: *Forensic Science International: Digital Investigation Volume 33, Supplement*. July 2020, Pages 301005. DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA. DOI: <https://doi.org/10.1016/j.fsidi.2020.301005>

Note: Awarded the Best Paper Award at DFRWS USA 2020 ([DFRWS 2020](#)).

Paper C *Timestamp prefix carving for filesystem metadata extraction*, Kyle Porter, Rune Nordvik, Fergus Toolan and Stefan Axelsson. In: *Forensic Science International: Digital Investigation Volume 38*. September 2021, Pages 301266. DOI: <https://doi.org/10.1016/j.fsidi.2021.301266>

Paper D *It is about time—Do exFAT implementations handle timestamps correctly?*, Rune Nordvik and Stefan Axelsson. In: *Forensic Science International: Digital Investigation Volumes 42/43*. October-December 2022, Pages 301476. DOI: <https://doi.org/10.1016/j.fsidi.2022.301476>

Note: The version included in this thesis is a corrected version, including the corrigendum of the original paper ([Nordvik and Axelsson 2023](#)).

In addition the the author has also published the following that is not part of this thesis

Reverse engineering of ReFS, Rune Nordvik, Henry Georges, Fergus Toolan and Stefan Axelsson. In: *Digital Investigation Volume 30, Supplement*. September 2019, Pages 127-147. DOI: <https://doi.org/10.1016/j.diin.2019.07.004>

Reliability validation for file system interpretation, Rune Nordvik, Radina Stoykova, Katrin Franke, Stefan Axelsson and Fergus Toolan. In: *Forensic Science International: Digital Investigation Volume 37*. June 2021, Pages 301174. DOI: <https://doi.org/10.1016/j.fsidi.2021.301174>

Law enforcement educational challenges for mobile forensics, Georgina Humphries, Rune Nordvik, Harry Manifavas, Phil Cobley and Matthew Sorell. In: *Forensic Science International: Digital Investigation Volume 38*. October 2021, Pages 301129. DOI: <https://doi.org/10.1016/j.fsidi.2021.301129>

Legal and technical questions of file system reverse engineering, Radina Stoykova, Rune Nordvik, Munnazzar Ahmed, Katrin Franke, Stefan Axelsson and Fergus Toolan. In: *Computer Law & Security Review: The International Journal of Technology Law and Practice Volume 46*. September 2022, Pages 105725. DOI: <https://doi.org/10.1016/j.clsr.2022.105725>

APFS, Rune Nordvik. In: *Hummert, C., Pawlaszczyk, D. (eds) Mobile Forensics – The File Format Handbook*. Springer, Cham.. May 2022, Pages 3-39. DOI: https://doi.org/10.1007/978-3-030-98467-0_1

Ext4, Rune Nordvik. In: *Hummert, C., Pawlaszczyk, D. (eds) Mobile Forensics – The File Format Handbook*. Springer, Cham.. May 2022, Pages 41-68. DOI: https://doi.org/10.1007/978-3-030-98467-0_2

Contents

List of Tables	xxii
List of Figures	xxvii
List of Symbols	xxxiii
1 Introduction	1
1.1 Research questions	2
1.2 Summary	4
2 Background	5
2.1 Digital forensics	5
2.2 Common file system basics	6
2.2.1 Volumes	6
2.2.2 Partition systems	6
2.2.3 Volume Boot Records	6
2.2.4 Blocks or clusters	7
2.2.5 Sector	7
2.2.6 Multi byte fields	7

2.2.7	Endianness	7
2.2.8	Timestamps	8
2.2.9	Atomic write and timestamps	10
2.2.10	Timezone	12
2.2.11	Additional sources for File Systems	13
2.3	Summary	13
3	Related Research	15
3.1	Introduction	15
3.2	Validation and verification	15
3.3	Daubert Standard (criteria)	16
3.3.1	Testing	16
3.3.2	Peer-review	17
3.3.3	General acceptance	17
3.3.4	Error rates	18
3.4	Method validation in Digital Forensics	20
3.5	Dual tool verification	21
3.6	Reliability Validation Enabling Framework	21
3.6.1	Technology	22
3.6.2	Methodology	22
3.6.3	Application	22
3.7	FRED - Framework for Reliable Experiment Design	23
3.7.1	Plan	23
3.7.2	Implement	24
3.7.3	Evaluate	24
3.7.4	Repeat	24
3.7.5	Analyse	24

3.7.6	Confirm	25
3.8	File Carving	25
3.9	Metadata carving	29
3.10	Timestamps	30
3.11	Summary	31
4	Contribution and publication summaries	33
4.1	Publication A - Using the object ID index as an investigative approach for NTFS file systems	33
4.1.1	Motivation	33
4.1.2	Research contributions	34
4.1.3	Technical contributions	35
4.1.4	Limitations	36
4.1.5	Summary	36
4.2	Publication B - Generic Metadata Time Carving	37
4.2.1	Motivation	37
4.2.2	Research contributions	37
4.2.3	Technical contributions	38
4.2.4	Limitations	38
4.2.5	Summary	39
4.3	Publication C - Timestamp prefix carving for filesystem metadata extraction	39
4.3.1	Motivation	40
4.3.2	Research contributions	40
4.3.3	Technical contributions	40
4.3.4	Limitations	40
4.3.5	Summary	40

4.4	Publication D - Its about time—Do exFAT implementations handle timestamps correctly?	41
4.4.1	Motivation	41
4.4.2	Research contributions	41
4.4.3	Technical contributions	43
4.4.4	Limitations	43
4.4.5	Summary	43
4.5	Summary	44
5	Discussion	45
5.1	General weaknesses with methodology	45
5.2	General strengths with methodology	46
5.3	R1: Identifying user activity using FS metadata	46
5.3.1	Publication A: Using the object ID index as an investigative approach for NTFS file systems.	47
5.3.2	Publication D: Its about time—Do exFAT implementations handle timestamps correctly?	48
5.4	R2: Connecting storage devices to computers using FS metadata	48
5.4.1	Publication A: Using the object ID index as an investigative approach for NTFS file systems.	49
5.4.2	Publication D: Its about time—Do exFAT implementations handle timestamps correctly?	49
5.5	R3: Using FS metadata to identify deleted files	50
5.5.1	Publication B: Generic Metadata Time Carving and Publication, C: Timestamp prefix carving for filesystem metadata extraction	50
5.5.2	Publication D: Its about time—Do exFAT implementations handle timestamps correctly?	51
5.6	R4: FS tool reliability and validation	53

5.6.1	Publication A: Using the object ID index as an investigative approach for NTFS file systems	54
5.6.2	Publication B: Generic Metadata Time Carving	55
5.6.3	Publication C: Timestamp prefix carving for filesystem metadata extraction	57
5.6.4	Publication D: Its about time—Do exFAT implementations handle timestamps correctly?	58
5.7	Summary	59
6	Conclusion and further work	61
6.1	To what degree can user activity be documented from non-OS volumes (external storage devices) using only FS metadata from the file system?	61
6.2	To what degree can an external storage device be used to identify the computers it has been attached to by only assessing metadata from the file system?	62
6.3	To what extent can FS metadata reliably identify deleted files?	62
6.4	To what extent can the reliability and accuracy of file system parsing performed by current DF tools be assessed?	63
6.5	Further work	63
A	Publication A: Using the object ID index as an investigative approach for NTFS file systems	75
A.1	Introduction	76
A.2	Related work and contributions	79
A.2.1	Related work	79
A.2.2	Contributions	81
A.3	Research goals	82
A.3.1	Research questions	82
A.3.2	Automation	82

A.4	Methodology	83
A.4.1	Object ID creation	83
A.5	Results	87
A.5.1	File creation	89
A.5.2	Opening a file	89
A.5.3	Copying a file (same volume)	90
A.5.4	Copying a file (other volume)	91
A.5.5	Moving a file (same volume)	91
A.5.6	Moving a file (other volume)	91
A.5.7	Deleting a file	92
A.6	Evaluation	93
A.6.1	Feasibility	93
A.6.2	Reliability	93
A.7	Discussion	94
A.8	Conclusions and future work	95
B	Publication B: Generic Metadata Time Carving	99
B.1	Introduction	100
B.1.1	Assumptions	101
B.1.2	Objectives	102
B.1.3	Novelty of the new approach	102
B.1.4	Importance for Digital Forensics	103
B.1.5	Organization of this paper	103
B.2	Related work and contributions	103
B.2.1	Metadata carving	104
B.2.2	Evaluating recovered files	105
B.3	Method	105

B.3.1	General Potential Timestamp Algorithm Description	106
B.3.2	Practical Potential Timestamp Program Details	108
B.3.3	Semantic parsers	108
B.3.4	Experimental setup	111
B.3.5	Experiment - NTFS reformatted with exFAT	111
B.3.6	Experiment - Previous Ext4 reformatted with NTFS	112
B.3.7	Limitations	113
B.4	Results	114
B.4.1	NTFS metadata carving	114
B.4.2	Ext4 metadata carving	115
B.4.3	Commercial tools	117
B.5	Discussion	119
B.5.1	Discussion related to NTFS	119
B.5.2	Discussion related to Ext4	120
B.5.3	Addressing Our Statistics and Current Challenges	120
B.6	Conclusion and further work	121
C	Publication C: Timestamp prefix carving for filesystem metadata ex- traction	127
C.1	Introduction	128
C.2	Related Work	131
C.2.1	Metadata Carving	131
C.2.2	Related Methods of Data Retrieval	133
C.3	Methodology	133
C.3.1	Prefix-Based Potential Timestamp Carving Algorithm	133
C.3.2	Generic Metadata Time Carving and the Filesystem Spe- cific Parsers	136
C.3.3	Experimental Methodology	138

C.3.4	Precision-Recall Location-Based Data Recovery Evaluation	139
C.3.5	Specifics of NTFS Experiments	141
C.3.6	Specifics of Ext4 Experiments	142
C.3.7	Computer Specifications	142
C.4	Results	143
C.4.1	Small NTFS Image	143
C.4.2	Ext4 Samsung S8 Image	143
C.4.3	Large NTFS Image	145
C.5	Discussion	148
C.5.1	Analysis: Small NTFS Image	148
C.5.2	Analysis: Ext4 Samsung S8 Image	149
C.5.3	Analysis: Large NTFS Image	149
C.5.4	Limitations	150
C.5.5	Revisiting Research Questions	154
C.6	Conclusion and Further Work	157
C.7	Acknowledgement	158
C.8	Prefix-Based Potential Timestamp Carving Algorithm	159
D	Publication D: It is about time—Do exFAT implementations handle timestamps correctly?	165
D.1	Introduction	166
D.1.1	Background	168
D.1.2	Research problem	170
D.1.3	Organisation of this paper	172
D.2	Related Work	172
D.3	Methodology	174
D.3.1	Experiment A - Base	174

D.3.2	Experiment B - mounting and unmounting only	176
D.3.3	Experiment C - accessing selected files	176
D.3.4	Experiment D - changing the content of all files	177
D.3.5	Experiment E - changing the content of selected files	177
D.3.6	Tool Testing	178
D.3.7	Limitations and assumptions	178
D.4	Results	179
D.4.1	Experiment A - Creating files on an exFAT storage	179
D.4.2	Experiment B - Mounting exFAT storage	181
D.4.3	Experiment C - Opening files	181
D.4.4	Experiment D and E: Changing exFAT files on multiple OSes	184
D.4.5	10msIncrement fields	185
D.4.6	Tool testing	185
D.5	Discussion	189
D.5.1	Rules for updating timestamps	191
D.5.2	10ms granularity	192
D.5.3	Patterns	192
D.5.4	Challenges	193
D.5.5	Tools	194
D.6	Conclusion and Further Work	195

List of Tables

2.1	Most file systems use a volume boot record	7
2.2	Endianess in a 4 byte field	8
2.3	Epochs used by different file systems	8
2.4	Different Ext timestamps found in inodes, listed in the order they are found in the structure	9
2.5	Timestamps found in two different MFT attributes, listed in the order they are found in each attribute.	10
2.6	Timestamps found in File Directory Entries in exFAT, listed in the order they are stored.	10
3.1	Reliability Validation Enabling Framework	22
3.2	Action sequences for the simple file system. All other combinations are impossible without manipulation. The list is filtered down from Willassen (2009) original list.	30
3.3	Action sequences for a simple file system with the update on access disabled, meaning the action Read has no impact. All other combinations are impossible without manipulation	30
4.1	Summary for experiments creating files - All tested file system drivers limiting to one particular timezone.	42

5.1	File directory entry (The fields are mandatory)	52
A.1	Offset table index entry, based on Carrier (2005, pp. 386-387).	86
A.2	Experiment 1 - Test 1: File Creation.	89
A.3	Experiment 1 - Test 2: Opening a file.	90
A.4	Experiment 1 - Test 3: Copying file to the same volume.	90
A.5	Experiment 1 - Test 4: Copying file to another volume.	91
A.6	Experiment 1 - Test 5: Moving file to the same volume.	91
A.7	Experiment 1 - Test 6: Moving file to another volume.	92
A.8	Experiment 1 - Test 7: Deleting a file.	93
B.1	File Systems with timestamps co-located within metadata structures	102
B.2	Precision and Recall for finding MFT records in ntfsexfat.dd	114
B.3	Precision and Recall for finding and attributing iNode numbers for known files in expExt4Attr.dd	116
B.4	Precision of iNode classification for non-reformatted image.	116
B.5	Precision and Files Found for finding and attributing iNode num- bers for known files in Ext4AttrNowNTFS.dd	116
B.6	Precision of inode classification for reformatted image.	117
B.7	Tool testing - Carve for metadata from previous file system when reformatted with another file system.	117
C.1	Precision and recall for carving MFT records from the \$MFT from the 1 GB NTFS image's partition beginning at sector 128 with $p = 1, 2, \dots, 8$. The \$MFT had 239 Condition Positives.	144
C.2	Test Positive count over entire partition from the 1 GB NTFS im- age, where $p = 1, 2, \dots, 8$	144
C.3	Generic Metadata Time Carving performance for the entire 1 GB NTFS image with $p = 1, 2, \dots, 8$. PTS stands for "Potential Time- stamp".	145

C.4	Precision and recall for carving inodes from the inode table from the SYSTEM partition in the 59.5 GB Ext4 Samsung S8 image with $p = 1, 2, 3, 4$. The inode table had 7436 Condition Positives.	145
C.5	Test Positive count over entire SYSTEM partition from the Ext4 Samsung S8 image, where $p = 1, 2, 3, 4$.	146
C.6	Generic Metadata Time Carving performance for the entire 59.5 GB Samsung S8 image with $p = 1, 2, 3, 4$. PTS stands for “Potential Timestamp”, m for minutes, and s for seconds. Note, the Ext4 parser skips the first approximately 210 MB.	146
C.7	Precision and recall for carving MFT records from the \$MFT of the Basic Data Partition from the 476 GB LoneWolf NTFS image with $p = 1, 2, \dots, 8$. The \$MFT had 142960 Condition Positives.	147
C.8	Precision and recall for carving MFT records from the \$LogFile of the Basic Data Partition from the 476 GB LoneWolf NTFS image with $p = 1, 2, \dots, 8$. The \$LogFile had 2604 Condition Positives.	147
C.9	Test Positive count over entire Basic Data Partition from the large LoneWolf NTFS image, where $p = 1, 2, \dots, 8$.	148
C.10	Files containing the remaining Test Positives not found in the \$MFT or \$LogFile of the Basic Data Partition, where $p = 1$. The number associated to each file indicates how many MFT records were found in that particular file.	148
C.11	Generic Metadata Time Carving performance for the entire 476 GB NTFS image with $p = 1, 2, \dots, 8$. PTS stands for “Potential Timestamp”, hr for hours, m for minutes, and s for seconds.	149
D.1	File directory entry (all fields are mandatory)	171
D.2	Experiment A Results - MacOS. We can see that stored timestamps use a timezone offset with switched signs compared to the computer the experiments were executed on	179
D.3	Experiment A Results - Win10. We can see that all types of timestamps are stored using the local UTC offset of the computer the experiments were executed on, and that real time is the same as stored time.	179

D.4	Experiment A Results - Linux Ubuntu 20.04 using exFAT fuse v.1.3. We can see that the timestamps are stored using the local time of the computer the experiments were executed on, and that real time is the same as stored time. However, the <i>UTCOffset</i> fields are not in use.	180
D.5	Experiment A Results - Linux Ubuntu 20.04 using exFAT native driver. We can see that the timestamps are stored using UTC+0, not the local time of the computer the experiments were executed on. The <i>UTCOffset</i> fields are used (set to 0x80).	180
D.6	Experiment B Results - MacOS. Impact of mounting and unmounting	181
D.7	Experiment C Results. Impact of opening files	181
D.8	Experiment D and E Results - Changes in Timestamps, 10msIncrement and <i>UTCOffset</i> fields in the exFAT file directory entry when changing the files on Windows, MacOS or Linux. The C* means a special case where an invalid UTCOffset for the Created is interpreted incorrectly and then the Created timestamps is converted using the switching feature of MacOS.	185
D.9	Experiment Results - Usage of the 10ms granularity fields in the exFAT file directory entry when using Windows, MacOS or Linux.	186
D.10	Experiment Results - MacOS and Autopsy v. 4.19.3	186
D.11	Experiment Results - MacOS and FTK Imager v. 4.5.0.3	187
D.12	Experiment Results - MacOS and X-Ways Forensics v. 20.04 SR-4	187
D.13	Experiment Results - MacOS and EnCase Forensic v. 22.1	187
D.14	Rules for updating timestamps - compliance	191

List of Figures

2.1	First 32 bytes of an Ext4 inode, including timestamps.	9
2.2	Atomic Write as implemented by an App using glibc in Linux. . .	11
2.3	ExFat timestamps on a file created in MacOS Big Sur v. 11.6., where the timezone is Central European Standard Time (UTC+1).	13
2.4	ExFat timestamps on a file created in Windows 10 pro v. 2004., where the timezone is Central European Standard Time (UTC+1), we selected (UTC+1) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna as the timezone.	13
3.1	Investigation Stages (Sunde and Horsman 2021)	17
3.2	Validation Framework, based on The United Kingdom Forensic Science Regulator (2020)	21
4.1	Structure of an Object ID UUID version 1.	34
4.2	Connecting ObjID index record with NTFS MTF record.	36
4.3	Connecting inodes with directory entries.	39
5.1	Unallocated directory set after rename.	53
5.2	Allocated directory set after rename.	53
5.3	ObjID index entry.	55

5.4	Hex dump with highlights to illustrate the timestamp prefix matching search procedure. The byte sequence underlined in green represents the current candidate timestamp, and those underlined with blue are test sequences. The brackets represent the candidate timestamp's search window ($k = 24$). The red boxes represent the little-endian prefixes ($p = 4$) that are being compared for equivalency. The first two examples show matches, despite the fact the candidate timestamp does not equal the subsequent ones. If three matching timestamps are required ($h = 3$), the third example shows the advancement of the search by k bytes, and begins to repeat the entire procedure.	57
A.1	Structure of an Object ID UUID version 1.	78
A.2	Exporting the MFT table, and MFT record number 25.	84
A.3	Hex dump of the Index Allocation Attribute.	84
A.4	Exporting the Object ID Index Allocation non resident data, and show one Object ID Index Entry.	85
A.5	Hex dump of an Object Index Entry.	85
A.6	C structure of an Object ID index entry.	86
A.7	NTFSObjIDParser output. Results are split between (a) and (b).	88
B.1	Visual representation of the search procedure where three matching time stamps are searched for. The underlined byte sequence represents the current byte sequence being tested as a possible timestamp. The subsequent bytes in brackets represent the search threshold for checking matches. The bytes in grey boxes represent checks for matching byte sequences. In the second row, after a second match is found, we advance the search procedure ahead by k bytes, where the process is repeated.	107
B.2	Diagram for system deployment, used in our experiments.	108

- C.1 An abstraction of a simple disk image, partitions, and filesystems. The large encompassing rectangle is the entire disk image, the furthest left rectangle with internal lines is the partition table that points to the partitions, and the other rectangles with rounded corners are partitions. Each partition has a filesystem, where the green rectangles represent filesystem critical data structures such as the \$MFT record (and its mirror), superblock, or group descriptor table. These help keep track of the filesystem records (for example, inodes or MFT records), which are represented by the red rectangles. *Generic Metadata Time Carving* (Nordvik et al. 2020a), and our work, attempts to find the red blocks without help from the green blocks. For a more complete picture of the general filesystem structure, see the work by Carrier (Carrier 2005). 131
- C.2 For 8 byte timestamps, the *candidate timestamp* is highlighted with green, and the *test sequences* are highlighted in blue. The search window is indicated by the brackets. The *timestamp equivalency test* simply checks how many times the candidate timestamp matches the test sequences. If the number of matches is greater than or equal to the threshold $h - 1$, where h is the number of required matching timestamps within a metadata record set by the user, the candidate timestamp is deemed a *potential timestamp*. 134
- C.3 Hex dump with highlights to illustrate the timestamp prefix matching search procedure. The byte sequence underlined in green represents the current candidate timestamp, and those underlined with blue are test sequences. The brackets represent the candidate timestamp's search window. The red boxes represent the little-endian prefixes that are being compared for equivalency. The first two examples show matches, despite the fact the candidate timestamp does not equal the subsequent ones. If three matching timestamps are required ($h = 3$), the third example shows the advancement of the search by k bytes, and begins to repeat the entire procedure. . . 135
- C.4 Histogram comparing the number of Condition Positives we account for on the Basic Data Partition of the 476 GB Lone Wolf image and the number of potential timestamp (PTS) locations identified after carving for all possible prefix lengths. 152

C.5	Histogram comparing the number of Condition Positives we account for on the SYSTEM partition of the 59.5 GB Samsung S8 image and the number of potential timestamp (PTS) locations identified after carving for all possible prefix lengths.	153
D.1	ExFat set of directory entries.	169
D.2	ExFat allocation of directory entries.	169
D.3	ExFat timeszone field, from hex byte to UTC offset.	170
D.4	Overview of all experiments, and all of them have a forensic image associated	175
D.5	Changes in timestamps and <i>UTCOffset</i> fields when opening a file using Gedit in Linux Ubuntu 20.04 exFAT fuse driver. The <i>LastAccessedTimestamp</i> is changed using local time (LT), which was UTC-5, however the <i>LastAccessedUtcOffset</i> is not changed. In this case the last accessed is inaccurate.	182
D.6	Changes in timestamps and <i>UTCOffset</i> fields when opening a file using Gedit in Linux Ubuntu 20.04 exFAT native driver. The <i>LastAccessedTimestamp</i> is stored as UTC+0, however the <i>CreateUtcOffset</i> and <i>LastModifiedUtcOffset</i> are also changed to UTC+0, but not the timestamps. In this case the create and last modified timestamps are inaccurate.	182
D.7	Changes in timestamps and <i>UTCOffset</i> fields when opening a file using TextEdit in MacOS Monterey/Mojave. The <i>LastAccessedTimestamp</i> is stored as UTC+5, even though the real timezone was UTC-5. The <i>CreateUtcOffset</i> is not changed, but the <i>LastModifiedUtcOffset</i> is changed to UTC+5, trying to convert LT from UTC-5 to UTC+5. In this case the last modified timestamp is inaccurate.	183
D.8	Changes in timestamps and <i>UTCOffset</i> fields when opening a file using Notepad in Windows 10. Nothing was changed. In this case the <i>LastAccessedTimestamp</i> is inaccurate.	183
D.9	ExFat timezones using stored UTC-1 offset for Experiment3 on MacOS and the X-Ways directory listing	188
D.10	ExFat timezones using stored UTC-1 offset for Experiment3 and the X-Ways for D2022-02-24T01-53-54-tz-3-file1.txt	188

D.11 ExFat timeszones using stored UTC+3 offset for Experiment0 from the Linux Base exFAT fuse image and the EnCase	189
D.12 ExFat timeszones using Europe/Oslo timezone (UTC+1) for Experiment-3 and the Windows 10 computer.	191

List of Abbreviations

ACPO Association of Chief Police Officers

APFS Apple File System

API Application Programming Interface

ASCII American Standard Code for Information Interchange

AVI Audio Video Interleave, a container for multimedia

BE Big Endian

BVOID Birth Volume Object ID

CAB Windows Cabinet files, compressed containers often used by installation programs

CFTT Computer Forensic Tool Testing

CMD Command

COW Copy-On-Write

cPTS Tool written in c, which finds potential Timestamps

CPU Central Processing Unit

DF Digital Forensic

DFRWS Digital Forensic Research Workshop

DLL Dynamic Link Library

DOC Document files

Epoch Start number of a timestamp from a specific year 0

exFAT extensible File Allocation Table, or the exFAT file system

Ext2 Second extended file system. Normally used on Linux distributions

Ext3 Third extended file system. Normally used on Linux distributions

Ext4 Fourth extended file system. Normally used on Linux distributions

FAT File Allocation Table

FNA File Name Attribute, contains the file name and additional set of timestamps in NTFS

FRED Framework for reliable experimental design

FS File System

fte file time extractor

FTK Forensic Toolkit

GDPR General Data Protection Regulation

GiB GibiByte, or 2^{30} or 1024^3

glibc GNU library for C

GMT Greenwich Mean Time

GMTC Generic Metadata Time Carver

GPT GUID partition table system

GUID Global Unique Identifier

IMRAD Introduction, Method, Results and Discussion

Inodes File or directory metadata records

IoT Internet of Things

JPEG Joint Photographic Experts Group

LE Law Enforcement

- LE* Little Endian
- LEA* Law Enforcement Agency
- MAC* Media Access Control
- MBR* Master Boot Record, usually the first sector of the disk
- Metadata* Data describing other data
- MFT* Master File Table. All files and records have at least one record in the MFT in NTFS
- MMS* Multimedia Messaging Service
- MSB* Most Significant Byte
- MSOLE* Microsoft Object Linking and Embedding
- NIC* Network Interface Controller
- NTFS* New Technology File System
- OID* Object ID
- OOXML* Microsoft Open Office XML File Format
- OS* Operating System
- OUI* Organisation Unique Identifier, the first 3 bytes of an MAC address
- OUI* Organizationally Unique Identifier
- PID* Process Identifier
- PNG* Portable Network Graphics
- PST* Personal Storage, often used as containers for outlook e-mail (POP accounts)
- QT* Graphical application programming interface for C++
- RAM* Random Access Memory (primary memory)
- RVEF* Reliable Validation Enabling Framework
- SIA* Standard Information Attribute, contains metadata such as the main set of timestamps for a file in NTFS

SVM Support Vector Machine

TXT Text

USB Universal Serial Bus

UTC Coordinated Universal Time

UUID Universal Unique Identifier

VBR Volume boot sector is a sector describing metadata about the file system and booting features.

ZIP ZIP archive, where ZIP means move at high speed

List of Symbols

$\sqrt{\quad}$	Square Root
Σ	Sum
\bar{e}	Average of error
S_e^2	Error Variance
S_e	Error Standard Deviation

Chapter 1

Introduction

Law Enforcement (*LE*) has become tool dependent when investigating criminal cases that involve digital storage and this dependency is justified due to the large backlog (Lillis and Scanlon 2016, Scanlon 2016). Even though previous research has found errors when assessing digital forensic tool results (failure to detect anti-forensic techniques (Bhat et al. 2020), 88% of practitioners have experienced errors using forensic tools (Horsman 2019), tool developers do not guarantee error-free tools (Horsman 2018b)), the focus on the file system interpretation performed by tools are more or less missing.

Digital forensic experts need to read the files and directories from a file system (*FS*) when investigating data storage devices, and one approach is reading it using the driver provided by the file system developer. However, the driver may change the file system data based on the usage of the file system during the investigation, and is therefore not considered forensically sound. Using the original driver does not give access to deleted files or other content in the unallocated areas of the file system. Therefore, it is considered good practice to make a forensic image of the storage device, and use digital forensic tools to parse the file systems in a read only mode. Hash algorithms are utilised to preserve the integrity of the data. As long as the hash verifies, the data integrity is intact. This good practice is based on the first principle from the Association of Chief Police Officers's (*ACPO*) Good Practice Guides for Digital Evidence, which requires that the data relied upon in court should not be changed (Williams 2012). Unfortunately, it is more difficult to assure data integrity during the acquisition process from a live device, for instance a mobile phone, since the device is changing parts of the data as long as it is alive. Horsman (2020) requires that all reasonable steps are taken to ensure the data is unchanged in his new fifth principle, preserving the data integrity as far as it is

practical. This principle opens for not preserving the integrity when this is very resource demanding or impossible.

This thesis depends on already acquired data including one or more complete file system where the content is already hashed, and where it is possible to preserve the integrity of the data. [Horsman \(2020\)](#) describes in his third principle that the investigators need to identify any and all relevant potential evidence, taking into account proportionality and necessity. Unfortunately, it is not possible to decide what is relevant, proportional, and necessary without actually viewing, assessing, and classifying the content either manually or automatically by using tools. Classification without viewing the content must be based on assumptions. Without the complete file system, it is often not possible to find relevant deleted content or relevant metadata describing file content or user activity. Therefore, the suggested granular approach should be applied after this classification is performed by filtering out non-relevant data.

We focus on the importance of file system metadata in an investigation context. *Metadata* is data describing other data ([Carrier 2005](#)), and metadata describing files is imperative for any investigation. These metadata structures (*inodes*) normally contain information about timestamps, location of content, owner, privileges, name, parent directory, and are all properties of a particular file or directory ([Carrier 2005](#)).

Large digital forensic suites automate the parsing of popular file systems, and the accuracy and reliability are trusted by most digital forensic practitioners. This assumption is based on trust, not on rigorous scientific principles. [Scanlon \(2016\)](#) describes that practitioners depend on tools because of the large backlog, which means they trust the tools. [Neale et al. \(2022\)](#) describe trust in criminal investigation and warn about the negative impact trusting tools may have on the investigation. Can we trust that digital forensic tools parsing different file systems produce accurate and reliable results, and will they show all relevant data? This leads us to the following questions.

1.1 Research questions

1. To what degree can user activity be documented from non-OS volumes (external storage devices) using only FS metadata from the file system?
2. To what degree can an external storage device be used to identify the computers it has been attached to by only assessing metadata from the file system?
3. To what extent can FS metadata reliably identify deleted files?

4. To what extent can the reliability and accuracy of file system parsing performed by current DF tools be assessed?

We have mainly focused on three main file systems; *NTFS* (used on Windows OS), *Ext4* (used on Linux or Android OSes), and *exFAT* (often used on multiple OSes).

The New Technology File System (NTFS) is a commonly encountered closed source file system from Microsoft. NTFS was implemented in 1993 in Windows NT 3.1 (Baloja 2017, Fandom 1993), and it is still used as the standard file system in Windows 10 (Karresand et al. 2020b) and 11. There has been a significant amount of research in this area, and Carrier (2005) has written chapters about the structures in NTFS that are important for digital investigation and analysis. However, still there are some unknown structures that may be relevant in criminal investigations. We have especially focused on the investigative value of object ids (Nordvik et al. 2019b) and carving for metadata for file recovery purposes based on co-located timestamps within the Standard Information Attribute (*SIA*) and the File Name Attribute (*FNA*) (Nordvik et al. 2020b).

Ext4 is an open source file system that builds on the previous *Ext2* and *Ext3* versions. Even if this is an open source file system with well known structures, from an investigation perspective, it is interesting to see if we can increase the accuracy and reliability of recovered files. Traditional techniques focus on carving for un-allocated files, excluding the metadata. However, more recently other techniques have been proposed for metadata carving (Dewald and Seufert 2017a). We propose a more generic approach by carving inodes or *MFT* records based on their co-located timestamps. These timestamps are identified by their equality properties (Nordvik et al. 2020b) or by prefix equality (Porter et al. 2021). The latter approach means testing the equality of the x number of the most significant bytes in the timestamps. The two approaches are equal when x equals the number of bytes in the timestamp. These approaches identify file metadata structures, allowing more accurate recovery of files.

In our latest contribution, Nordvik and Axelsson (2022), we have performed experiments on the exFAT file system. The experiments were used to validate tools and to assess how different operating systems and exFAT file system drivers have implemented the exFAT specifications. Furthermore, we assessed if the timezone offset of the computer used could be found within the exFAT metadata. We cannot assume that different file system drivers implement the same specification identically. We observed that using a storage device on different operating systems, with different file system drivers, impact the accountability of the metadata stored (Nordvik and Axelsson 2022).

1.2 Summary

In this chapter an introduction to metadata of file systems, investigation tool dependency, evidence relevance, proportionality, and necessity, and the thesis research questions were given. In order to increase the reader target group, a basic background on file systems is introduced in chapter 2. Chapter 3 focuses on related work. Then a presentation of the thesis results are presented in chapter 4, and further discussed how the results are relevant to the overall research questions in the chapter 5. Finally, the thesis concludes in chapter 6. The remaining chapters include the papers this thesis is comprised of.

Chapter 2

Background

In this section we will introduce the reader to file system basics, because the basics are important to know in order to understand our research. We will include common file system features and others specific to selected file systems.

2.1 Digital forensics

At the earliest stage of digital forensics, about 50 years ago ([Garfinkel 2010](#)), the main focus was recovery of deleted files, which required file system knowledge or tools that utilised this knowledge. There were several research publications about file systems, and it is legitimate to ask if there could be more unanswered questions about the most common file systems. Without the knowledge of the metadata structures in file systems, we can only carve for files based on signatures or other known semantics (patterns). In order to analyse file systems, knowledge of file system metadata is essential. We depend on file system forensics in order to perform digital forensics ([Marshall and Paige 2018](#)). In the golden age of digital forensics (1999-2007) investigators with little training could use automated tools to recover deleted files and the main focus was on the Windows file systems ([Garfinkel 2010](#)).

Currently, investigators are tool dependent because of large backlogs ([Scanlon 2016](#)), and the accuracy of tools are often based on trust, not science ([Neale et al. 2022](#)). The security of devices is increased, making it harder to access file systems due to encryption ([Caviglione et al. 2017](#)). New file systems have been introduced that tools do not fully support. New file systems and operating systems are introduced for mobile phones, drones, *IoT* devices, cloud solutions, etc. As such, law enforcement practitioners must understand file system metadata structure in order to verify findings and increase the quality of the investigation.

2.2 Common file system basics

In this section we discuss concepts that are currently used by most file systems. We will not go into detail about each metadata structure for each of the selected file systems. The details about particular file systems are described in the publications of this thesis. For more information about file systems, we suggest the File System Forensic Analysis book by [Carrier \(2005\)](#) as a good reference.

2.2.1 Volumes

All file systems use the concept of volumes, for instance a file system volume. A volume can be defined as:

"A volume is a collection of addressable sectors that an Operating System (OS) or application can use for data storage." ([Carrier 2005](#))

A volume is a logical set of contiguous sectors, but these sectors do not need to be physically contiguous. Therefore, all sectors of a complete storage device is a volume (disk volume), a partition is a volume, and the set of sectors used for a file system is a volume. A file system volume may just as well merge different partitions from different physical devices into one volume. Even a file system volume may divide the volume into further volumes. For instance, the Apple File System (*APFS*) uses the concept of a container volume which is located on a partition volume, and the container will divide this container volume into sub-volumes ([Hansen and Toolan 2017](#)).

2.2.2 Partition systems

The two most popular partition systems used for storage devices are the legacy master boot record (*MBR*) and the Global Unique Identifier (*GUID*) Partition Table (*GPT*). *APFS* has, in addition to the used *GPT* system, their own internal volume system, managed by the Container volume ([Hansen and Toolan 2017](#)).

Partition systems define a number of partition volumes, their locations on the disk, and sometimes identifies the type of volume that should be present. The *MBR* can support 4 partitions if extended partition tables are not in use, and the *GPT* supports 128 partitions as standard ([Carrier 2005](#)). An *APFS* container can add their own additional volumes within the assigned partition volume, where allocation status and content blocks may be shared between internal *APFS* volumes ([Hansen and Toolan 2017](#)).

2.2.3 Volume Boot Records

A FS will normally be located within a partition. Within the FS there is a volume boot record (called superbblock in Ext), often located near the start of the FS volume.

Figure 2.1 shows that different file systems use multiple names for the same volume boot record (*VBR*) concept. This record contains metadata about the file system it self and is necessary for the FS driver to locate the different parts of the file system, e.g. location of inodes, location of allocation table, location of data blocks, etc.

Table 2.1: Most file systems use a volume boot record

Boot records	FS
Superblock	Ext2,3,4
Container Superblock	APFS
Volume Header	HFS+
VBR	NTFS, FAT, exFAT, ReFS

2.2.4 Blocks or clusters

The concept of blocks and clusters is the same, and it describes a set of sectors as an addressable unit. A block or a cluster is the smallest unit that can be assigned to a file [Carrier \(2005\)](#). No matter how small a file is, it will at a minimum contain one block or cluster if it uses non-resident data. Some file systems allow small files to be stored resident within the file metadata entry ([Ext4 \(and Ext2/Ext3\) Wiki 2019](#), [Carrier 2005](#)), using the blocks assigned to the metadata table.

2.2.5 Sector

A sector is the smallest unit the OS can write to or read from ([Carrier 2005](#)). Traditionally, a sector was 512 bytes. However, a sector may just as well be 2^x bytes, for instance 1024, 2048, 4096, etc. bytes. We have not observed smaller sector sizes than 512 bytes. We have observed 4096 bytes sectors on iOS and Android devices.

2.2.6 Multi byte fields

In a hex dump we may have some bytes that belong together, for instance a timestamp that uses 8 bytes. This is considered a multi byte field, and the endianness needs to be taken into consideration.

2.2.7 Endianness

There are two types of endianness; Little Endian (*LE*) and Big Endian (*BE*). In Little Endian the first bytes to the left have least weight, and therefore we need to read the bytes from the right to the left. In Big Endian the first bytes to the left have most weight, meaning we need to read them from left to right. Endianness is only important for multi byte fields ([Arora 2012](#)). Endianness will not impact the reading of an array of single byte fields. Table 2.2 shows an example of little- and

big-endian interpretation.

Table 2.2: Endianess in a 4 byte field

Raw Value	Little Endian	Big Endian
0x01020304	0x04030201	0x01020304

The *CPU* decides the endianess used, and some processors may have support for both types, which is called Bi-endianess (Arora 2012). The operating system is compiled with the endianess that should be used, and this endianess must be supported by the CPU. Digital forensic experts should be aware that some computer architectures will use a specific endianess, and adjust their tools accordingly.

When transferring data through the network, big endian is used (Arora 2012). Therefore, if necessary, there will be a conversion between the endianess of the network data and the host data.

2.2.8 Timestamps

Timestamps are often represented as a number since a particular date in time. This date is called the epoch. The number for the Unix epoch is representing seconds since 1970, and for NTFS and ReFS it represents the number of 100 nanoseonds intervals since 1601 (Carrier 2005, Nordvik et al. 2020b; 2019a). Different file systems may use different epochs, as shown in Table 2.3.

Table 2.3: Epochs used by different file systems

FS	Epoch	Granularity	From
Ext2,3,4	Unix Epoch	Seconds	1.1.1970 (Carrier 2005)
NTFS	FILETIME	100 nano second intervals	1.1.1601 (Carrier 2005)
REFS	FILETIME	100 nano second intervals	1.1.1601 (Nordvik et al. 2019a)
APFS	Unix Epoch	Nanoseconds	1.1.1970 (Apple 2018)

Figure 2.1 demonstrates a standard 256 byte Ext4 inode, and from relative offset 0x8 we find four 32 bit timestamps where each timestamp represents the number of seconds since 1970 (Unix *epoch*). Using a time converter, we can find that all timestamps in the hex dump with a value of 0x5A685FA8 LE¹ representing 2018-01-24 10:27:52 GMT. The entry from relative offset 0x14 is the 32 bit time of deletion, here it is 0x0 (unused). From relative offset 0x90 we find the 32 bit timestamp that represents the creation time. This timestamp also has the value 0x5A685FA8 LE (2018-01-24 10:27:52 GMT). Since three of the timestamps are

¹LE means interpreted or read as little endian.

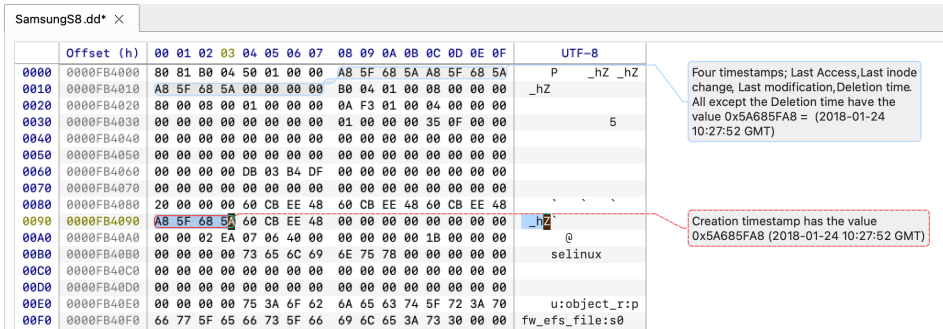


Figure 2.1: First 32 bytes of an Ext4 inode, including timestamps.

near co-located, they can be identified by our Generic Metadata Time Carving (*GMTC*) approach.

Previous versions of Ext will not have the creation time, and any tool that supports only Ext2 or Ext3 will miss the creation timestamp and will not take into consideration the new sub-second precision for all the timestamps. This is because Ext4 has additional time related metadata fields starting from offset 132 in each inode, while the Ext2 and Ext3 as standard only used 128 bytes for inodes (Dewald and Seufert 2017a).

The different timestamps in a metadata structure describing a file or directory differ between file systems.

Table 2.4: Different Ext timestamps found in inodes, listed in the order they are found in the structure

File System	Content Access	Metadata change	Content modification	Deleted	Creation
Ext2 or Ext3	a-time	c-time	m-time	d-time	N/A
Ext4	a-time	c-time	m-time	d-time	cr-time

As shown in Table 2.4 we can see that timestamps in Ext2 and Ext3 are almost equal to Ext4, except from the additional creation timestamp which is not co-located with the other timestamps.

In NTFS the timestamps are found in more than one attribute in each MFT record, in the Standard Information Attribute and in each File Name Attribute, and their order deviates from the order used in Ext2, 3 and 4 (Carrier 2005) as seen in Table 2.5.

In exFAT only three timestamps are in use, and these are also near co-located. The order of the timestamps are shown in Table 2.6.

Table 2.5: Timestamps found in two different MFT attributes, listed in the order they are found in each attribute.

File System	Creation	Modified	MFT Modified	Accessed
NTFS	cr-time	m-time	c-time	a-time

Table 2.6: Timestamps found in File Directory Entries in exFAT, listed in the order they are stored.

File System	Creation	Modified	Accessed
exFAT	cr-time	m-time	a-time

2.2.9 Atomic write and timestamps

Atomic write is a storage method that requires that a file, often consisting of multiple blocks/clusters, is stored completely, or not at all (Okun and Barak 2004). The method can be implemented as part of a file system, database, library, or application. Some file systems implement a similar approach called Copy-On-Write (COW) where the original data block is read, changed in memory, and written to another block (Chen et al. 2014). We do not consider COW as an atomic write of a file, since COW only manages each block as an atomic write operation, and as described by Chen et al. (2014) may result in recursive new COW writes of parent blocks. Applications may use their own atomic write method for files. One method of implementing atomic write for files is to create a temporary file, write the content to this temporary file. If previous write was successful, rename the file to the target file name (Richard Maw 2016). If the target file was an existing file, it must be unallocated before the renaming can take place.

Developers often use libraries for storing files, therefore, they may not even know how using atomic write functions may impact existing metadata. Using atomic write functions to change an existing file, may end up with a new file with a new set of timestamps from the new temporary file. In Figure 2.2 we can see a simplified example of how atomic write is implemented by an application using *glibc* libraries. It first reads the metadata of the file *Plans.txt* and loads the content from the data block pointers. The user adds some text and presses File Save. The App creates a new temporary file, which gets the current timestamps for the file creation (new timestamps are generated by the file system). Then the file content is written to a new location allocated to the new temporary file. If the write fails, the temporary file is unallocated and the process is exited. If the write succeeds, the original *Plans.txt* is unallocated. Then the temporary file is renamed using the same name as the original file (here *Plans.txt*) as target, which preserves the time-

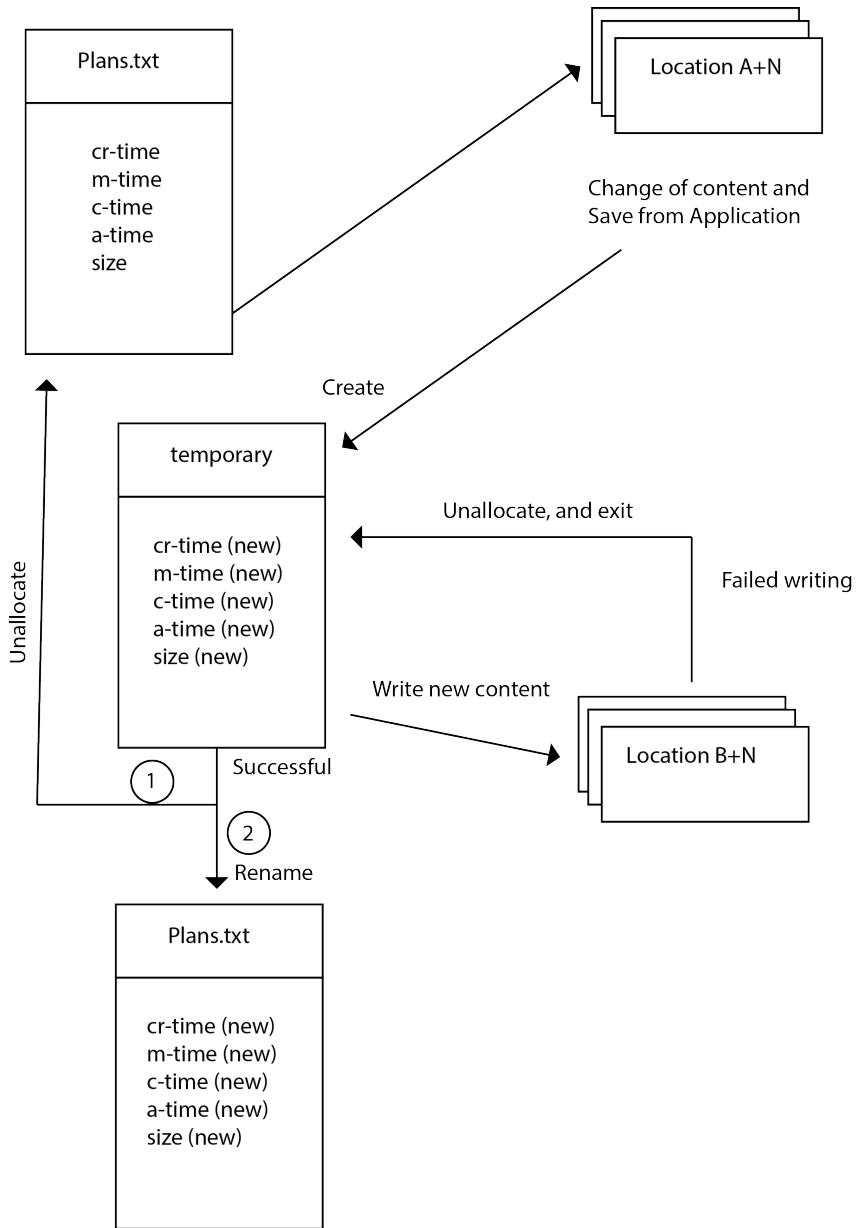


Figure 2.2: Atomic Write as implemented by an App using glibc in Linux.

stamps and other metadata from the temporary file, however, the original metadata is often overwritten and lost. This means if *Plans.txt* was originally created in 2015, changing it by using an application that uses atomic write in 2023 will impact the creation date and other timestamps, setting them to 2023.

The researcher tested this by creating a small application that implements a storage function² from the glibc libraries that uses atomic storage of files (Gnome Developer 2014). The researcher observed that changing an existing file resulted in a new file with the same name and a new set of timestamps in exFAT or Ext4, including the created timestamp. In Windows this behaviour is mitigated by File System tunneling, a feature that sets the same created timestamp of the previous file if the previous file is unallocated and another file is allocated with the same file name within 15 seconds as standard (Carvey 2018).

2.2.10 Timezone

Several file systems store their timestamps using UTC+0, however FAT uses local time, meaning the FS stores the time based on the local timezone of the computer or the phone it was located. This means there is no point in changing the timezone using a *DF* tool when analysing a FAT file system, since it does not know the original local time. Most *DF* tools will not allow adjusting the local time for a *FAT* file system even if we try to select another timezone.

ExFAT on the other hand should be storing the timestamp as local time, but also include UTC offset information for the timezone and/or daylight time used by the local computer (Microsoft 2021). Unfortunately, our experiments show that this is not always implemented correctly by the file system exFAT drivers. Knowing the UTC offset makes it easy to convert the timestamp back to any timezone wanted, however, it may not be supported by the digital forensic tools if they do not take the timezone offset byte into consideration, or if the stored UTC offset value is invalid. The 8 bits used for each UTC offset value use the most significant bit as the enable/disable bit, followed by a 7 bits two's complement number.

In Figure 2.4 we observed the UTC offset had the value UTC+1, the same as the local time we had on our Windows 10 computer. We also included the time within the filename as the time we had in CET at the moment we created the file. In Figure 2.3 we can see that the UTC-1 offset is used for storing the timestamp on the MacOS computer, even though the local timezone of the computer was UTC+1.

The standard describes that exFAT uses local time, however, we have already shown that this may not be the actual local time of the computer.

²We used the `g_file_set_contents` function.

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	UTF-8	
0000	0000160120	85	04	C4	92	20	00	00	00	85	56	39	54	B5	56	39	54	É V9T V9T
0010	0000160130	D0	56	39	54	B4	FC	FC	00	00	00	00	00	00	00	00	00	9T
0020	0000160140	C0	03	00	25	89	B4	00	00	3F	FE	06	00	00	00	00	00	% ?
0030	0000160150	00	00	00	00	68	00	00	00	3F	FE	06	00	00	00	00	00	k ?
0040	0000160160	C1	00	53	00	63	00	72	00	65	00	65	00	6E	00	73	00	S c r e e n s
0050	0000160170	68	00	6F	00	74	00	20	00	32	00	30	00	32	00	32	00	h o t 2 0 2 2
0060	0000160180	C1	00	2D	00	30	00	31	00	2D	00	32	00	35	00	20	00	- 0 1 - 2 5
0070	0000160190	61	00	74	00	20	00	31	00	32	00	2E	00	35	00	33	00	a t 1 2 . 5 3
0080	00001601A0	C1	00	2E	00	33	00	38	00	2E	00	70	00	6E	00	67	00	. 3 8 . p n g
0090	00001601B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	

Creation: 2022-01-25 10:53:42
 Creation: Additional 10 ms units, here 0xb4 (180 in dec). Meaning 1800 ms, or 1,8 seconds
 TimeZone offset: 0xFC, or 1111 1100 in binary. Enabled, -64+32+16+8+4 = -4. Units are 15 minutes, meaning -4*15= -60 minutes, or UTC-1 hour

Figure 2.3: ExFat timestamps on a file created in MacOS Big Sur v. 11.6., where the timezone is Central European Standard Time (UTC+1).

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	UTF-8	
0000	00001603A0	85	03	C0	A7	20	00	00	00	72	75	39	54	73	75	39	54	ru9Tsu9T
0010	00001603B0	73	75	39	54	65	00	84	84	84	00	00	00	00	00	00	00	su9Tn
0020	00001603C0	C0	01	00	1C	16	C4	00	00	00	00	00	00	00	00	00	00	
0030	00001603D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
0040	00001603E0	C1	00	43	00	72	00	65	00	61	00	74	00	65	00	64	00	C r e a t e d
0050	00001603F0	32	00	35	00	30	00	31	00	32	00	32	00	2D	00	31	00	2 5 0 1 2 2 - 1
0060	0000160400	C1	00	34	00	34	00	33	00	2D	00	77	00	69	00	6E	00	4 4 3 - w i n
0070	0000160410	31	00	30	00	2E	00	74	00	78	00	74	00	00	00	00	00	1 0 . t x t

Creation: 2022-01-25 14:43:36
 Created: 10 ms units, here 0x6E (110 in dec), meaning 1100 ms, or 1.1 additional seconds.
 TimeZone Offset: 0x84 or 1000 0100 in binary. Enabled 4 (15 minutes units). 4*15=60 minutes, or UTC+1

Figure 2.4: ExFat timestamps on a file created in Windows 10 pro v. 2004., where the timezone is Central European Standard Time (UTC+1), we selected (UTC+1) Amsterdam, Berlin, Bern, Rome, Stockholm, Vienna as the timezone.

2.2.11 Additional sources for File Systems

In this chapter only a brief introduction to file systems was given, but for additional reading we recommend the book “File System Forensic Analysis” by [Carrier \(2005\)](#), or the book “Forensic Examination of Windows Supported File Systems” by [Elrick \(2015\)](#). More information about Ext4 can be found in the paper “An analysis of Ext4 for digital forensics” by [Fairbanks \(2012\)](#).

2.3 Summary

In this chapter the basic background on file systems was introduced, focusing on similarities. All file systems use some sort of volumes, most FSs are managed by a partition system. Currently, most FSs use a volume boot record, and files store non-resident data in clusters or blocks. All file systems use endianness on multi-byte fields. Timestamps are used in metadata describing files, and timestamps use an epoch and they are often presented using a timezone. In addition, atomic write may effect the accuracy of timestamps connected to a file.

The next chapter expands on the works discussed in this chapter and provides an introduction to the related research focusing on legal guidance for scientific research, how to perform reliable experiments, an introduction to validation and verification, and previous research on file carving versus metadata carving.

Chapter 3

Related Research

3.1 Introduction

While the main focus for this research is about metadata in file systems, the findings impact research domains such as validation and verification of digital forensic tools or processes, carving, reverse engineering, investigation of criminal cases, and legal challenges. The selected domains and related work are not exhaustive. Instead, those selected and discussed below, give an overview of related work not previously discussed in sections of the publications included in this thesis.

3.2 Validation and verification

In order to understand tool or process testing, we must consider and understand the definitions of validation and verification.

"Validation is confirmation that the user requirements for a specific purpose (intended use) have been fulfilled, and the validation is carried out on processes (e.g., demonstrating the suitability of a selected algorithm for a specific process)" ([ISO/IEC 2015](#)).

The validation results are assessed by users of the tool (or process), and they decide if the tool is suitable for its intended purpose. This also means that one Law Enforcement Agency (*LEA*) may validate the process (or tool) for one purpose, but the same LEA may not validate the process (or tool) for another purpose. What is validated or not may even deviate between different LEAs depending on the requirements the LEA set.

"Verification is confirmation that a product conforms to specified requirements, and these requirements are related to the product specifications (e.g., showing that

the algorithm is implemented correctly)" ([ISO/IEC 2015](#)).

Verification is related to assessing if a product fulfills the specifications. The specifications for tools are normally in the form of the developer specifications. This type of verification is performed by the developer organisation during the development.

A product could also be the results from a tool. Verification of results are more related to how accurate the results are based on the available specifications for an artifact. This normally requires manual verification by the investigator who can access the raw data and interpret the structures based on available specifications. Very often accurate and reliable results are requirements LEAs use in order to validate a tool, because as [Bhat et al. \(2020\)](#) describe: "A digital forensic investigation may be rendered inconclusive if doubt creeps into the credibility of the forensic tool used".

3.3 Daubert Standard (criteria)

In the case *Daubert v. Merrell Dow Pharmaceuticals* ([United States Supreme Court 1999](#)) the Daubert standard was defined. It requires the forensic theory or technique to be: (1) tested, (2) peer-reviewed, (3) generally accepted in the scientific community, (4) account for error rates, and (5) within the examiner's expertise.

3.3.1 Testing

The criteria for how the testing should be performed is not described. Very often the technique or method is implemented by using tools. [NIST \(2023a\)](#) has a computer forensic tool testing program (*CFTT*) that focuses on the accuracy of disk imaging tools, mobile device tools, hardware write blocker tools, forensic string search tools, and forensic media preparation tools.

When it comes to testing of mobile acquisition tools, we can see that some of the results are not as expected ([NIST 2023a](#)), meaning they have failed to show an artefact that is present on the device and that should be supported by the tool. For instance the Magnet Axion v6.8.0.33717 showed results as not accepted on multiple devices ([NIST 2023b](#)):

- contacts and calendar (missing)
- notes/memos (missing)
- *MMS* attachments (missing)
- Call logs (missing).

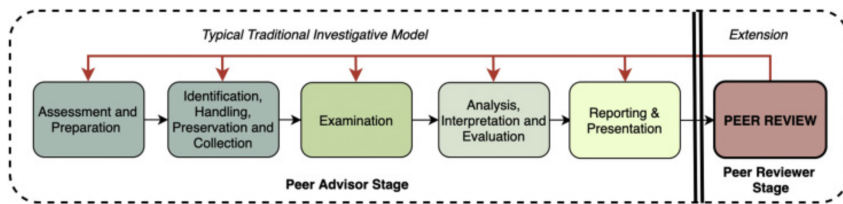


Figure 3.1: Investigation Stages (Sunde and Horsman 2021)

- Social media (missing or partial)

The tool in the example above is tested, and therefore complies with the first requirement of the Daubert standard. Meaning, everything the tool supports and finds as expected could be used as a potential evidence. However, it does not mean the tool will find everything. The investigator using only this tool may not find the information that may support the innocence or guilt of the suspect.

3.3.2 Peer-review

It is not described which competence the peer-reviewer should have, neither the type of peer review necessary, or the frequency of reviews. Sunde and Horsman (2021) propose the Phase-oriented Advice and Review Structure (PARS) for Digital Forensic investigations. During the early phases of the investigation an advisor is used to supervise and guide the investigation, as shown in Figure 3.1. Each of the phases of the investigation are treated as checkpoints where a review is performed by one advisor and advice is given after each checkpoint review to assure the quality of the investigation. Finally, the end report is reviewed by another independent reviewer at the end of the investigation. It is important that the final reviewer is not the same as the advisor to avoid bias.

3.3.3 General acceptance

While a procedure or technique may be accepted in the DF community, it does not mean that procedure or tool is reliable or currently valid¹. In digital forensics things change quickly, and a valid procedure may become invalid because of improved security of hardware or software, or limited by legal constraints. This acceptance requirement is also a barrier for new research and new techniques. The number of users of a tool is not a valid quality measurement of the procedures or techniques implemented by the tool (Carrier 2002).

¹A procedure may be valid when tested, but future changes in tools or in the file system used may impact the validity of the procedure.

3.3.4 Error rates

How to compute error rates are not described, and very often not performed or shared by the vendors. It was suggested by [Carrier \(2002\)](#) that there is a need for creating a standard for calculating error rates for tools or procedures. This is something, to date, that is not standardized in the DF community. [Lyle \(2010\)](#) discuss challenges with error rates but does not aim to give a solution. He states there are two types of errors; false positives (detecting something that is not there) and false negatives (not detecting something that is there). A true positive is therefore detecting something that is really there, and a true negative is not detecting something that is not really there. [Lyle \(2010\)](#) also discusses other variables that could be causing the errors other than the DF tool or procedure, for instance the operating system or the BIOS (number of cylinders and sectors are miscalculated), could be causing the error of an acquisition tool. In his examples he uses the equation:

$$k/n$$

where k is the number of incorrectly acquired bits, and n is the number of bits acquired. This is the same as

$$\frac{\text{False Positives}}{\text{True Positives} + \text{False Positives}}$$

[Erbacher \(2010\)](#) discusses the need of validation and error in order to assess the admissibility of evidence. He defines an error as the likelihood that the result is wrong. He further describes that users normally trust the output of a software tool to be correct, even though there could be implementation errors. Errors related to software implementation are not well defined and need to be researched. [Erbacher \(2010\)](#) suggests that legally trusted (admissible) algorithms or equations used in software tools must have a formal proof and verification that includes formal methods, testing (internal), peer and external evaluation. In addition, there could be human error interpreting the results in the analysis phase of the investigation. When it comes to computing error he suggests using mean error and standard deviations in order to assess if an error is significant or not. He gives an example of an error rate of 0.75%, which may not seem significant. However, if the mean error rate is 0.1% and the standard deviation is 0.05, then a measured error rate of 0.75% is significant.

In order to compute error rates the following formulae might be used to compute the error rates mentioned by [Erbacher \(2010\)](#). During an experiment i all observed false negatives and false positives should be interpreted as the observed errors e_i . In the formula below we compute the mean error \bar{e} of all experiments n .

$$\bar{e} = \frac{1}{n} \sum_{i=1}^n e_i$$

Each observed error rate may deviate from the mean error rate. Therefore, we must compute the empirical error variance.

$$\text{Variance} = S_e^2 = \frac{1}{n-1} \sum_{i=1}^n (e_i - \bar{e})^2$$

Then we can compute the standard error deviation.

$$\text{Standard deviation} = S_e = \sqrt{S_e^2}$$

[Nordvik et al. \(2020b\)](#), [Porter et al. \(2021\)](#) computed error rates using precision and recall where:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Precision is important in an investigative context to measure the percentage of real true positive hits. The recall is important in an investigation to measure the percentage that all true positives were found ([Porter et al. 2021](#)). The precision and recall correlate, high precision often means low recall and vice versa. Too many false negatives mean the investigator may not find all potential exculpatory or inculpatory evidence.

Precision and recall are simple to compute, but require that the relevant base truth is known for the dataset, or areas of it, in order to be computed accurately. For instance, the NTFS file system may have MFT records in the MFT table, in the MFT mirror, in the journal, in the page file, in swap file, in shadow volume copies, etc. We need to know exactly each of these in order to accurately classify a hit as true positive or true negative in order to test a tool that carves for MFT record metadata structures.

3.4 Method validation in Digital Forensics

The described definition of validation in section 3.2 is also supported by [The United Kingdom Forensic Science Regulator \(2020\)](#) (Method Validation in Digital Forensics) which describes validation as a fitness for purpose demonstration. Therefore, it is necessary to define the end-user requirements that must be met in order to validate a method or tool. A method includes a sequence of operations (procedures) for a defined task. The datasets used for validation should be representative of real life use. [The United Kingdom Forensic Science Regulator \(2020\)](#) also describe that a validation study creates all or part of the evidence meeting the requirements, but can collate evidence from scientific literature or other studies. “There are no standard methods in digital forensic science”, but methods are assumed adopted (fully or partly) and are usually not novel ([The United Kingdom Forensic Science Regulator 2020](#)). Figure 3.2 illustrates the validation process based on the validation framework described in [The United Kingdom Forensic Science Regulator \(2020\)](#). [The United Kingdom Forensic Science Regulator \(2020\)](#) describes that it is irrelevant to validate functions/features that will not be used.

The end-user is not only the forensic unit, public sector body, service providers, or independent consultants providing forensic science services to law enforcement agencies, but also legal stakeholders that depend on the results (prosecution, defence, and the court). For the court these requirements are important for evidence applicability ([The United Kingdom Forensic Science Regulator 2020](#)):

- Data quality on which the expert opinion is based, and the validity of the methods used to obtain the data.
- How safe/unsafe are inferences from findings.
- The degree of method precision or margin of uncertainty, accuracy, or reliability of methods that expert opinion relies upon (Daubert Testing and Error-rate criteria).
- To which extent the expert opinion is based on methods that are peer-reviewed by others with relevant expertise (Daubert Peer-Review criteria).
- To which extent the expert opinion is outside of the expert’s expertise.
- The completeness of information available to the expert, and if all relevant information was taken into consideration in the expert opinion.
- If there is a range of expert opinions, has the expert properly explained (justified) his/her opinion?

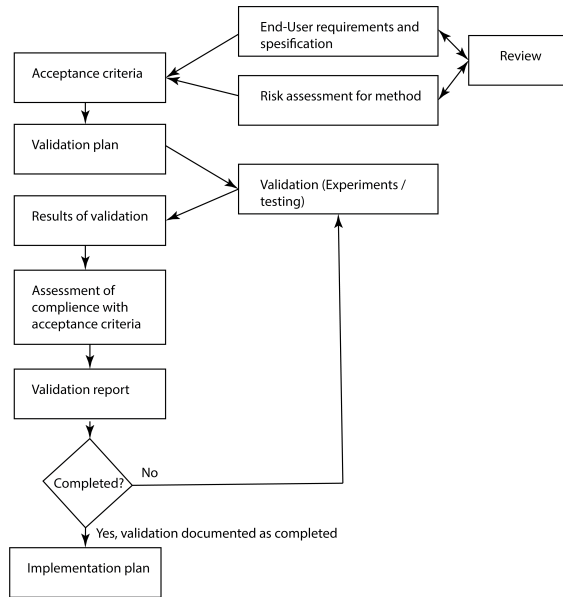


Figure 3.2: Validation Framework, based on [The United Kingdom Forensic Science Regulator \(2020\)](#)

- Justification for expert deviation from established practice (Daubert General acceptance criteria).

3.5 Dual tool verification

[Knight and Leveson \(1986\)](#) describe that it is erroneous to assume that programmers do not make the same errors if they were to implement the same specification. Therefore, it is not the best approach to use dual tools to verify the first tool ([Nordvik et al. 2021](#)). It is not unlikely that two similar functions can include similar errors, even when developed by different persons. Different tools may make use of the same library, and may therefore contain the same error ([Nordvik et al. 2021](#)).

For instance, assume that two similar tools give exactly the same results. How do we know if both tools are correct, or if they have shown the same erroneous result?

3.6 Reliability Validation Enabling Framework

[Nordvik et al. \(2021\)](#) proposed a minimum set of requirements necessary to assess the validation in digital forensics related to file system reverse engineering. Later [Stoykova and Franke \(2023\)](#) proposed the Reliability Validation Enabling Framework for Digital Forensics in general. If not all these requirements are avail-

able, it will be difficult to assess if a process or tool can be validated. It is how these requirements are implemented that must be assessed in order to classify if the process, method, or tool may be validated or not.

Table 3.1: Reliability Validation Enabling Framework

Level	5WH	Example requirements
Technology	What	Technical information about tool or process
Methodology	How	How the process is performed
Application	Who and why	Justify the examiner, and why a particular setting is used

As shown in Table 3.1, the Reliability Validation Enabling Framework consists of three levels. The framework was suggested by the second author Radina Stoykova, and we collaborated on adapting it to reverse engineering of file systems (Nordvik et al. 2021).

3.6.1 Technology

The technological level consists of documenting the tools (name, version, feature, function, algorithm, reference to previous validation/verification of tools, reference to previous error reports, identify if tool produce errors in output, etc.

3.6.2 Methodology

The methodology level describes how the tool / process has been developed. For instance, if the tool / process was peer reviewed during development then information about this peer review should be included. This might include how the peer review was performed and to what depth the peer review assessed the tool / process. This level includes a detailed description of how the tool was developed, including a description and justification for any assumptions made during the development process and also any limitations of said process. The methodology level also describes the means of testing / evaluation used in ensuring the correctness of the tool. This not only includes details of the tests performed but also of the datasets used during evaluation, again noting any assumptions of limitations of the employed evaluation methods. Process / tool implementation compliance with accepted digital forensics research should be documented and any deviations from such should be justified and published.

3.6.3 Application

The application level consists of who is using the tool, process, and why it is used in a particular way. Does the examiner have the right competence / experience to use the process correctly? Can the method, algorithm, parameters, and features selected be justified? Is precision and recall computed for the dataset it is used on?

Does the examiner have the right competence to interpret the tool results?

3.7 FRED - Framework for Reliable Experiment Design

Horsman (2018a) proposed a new Framework for Reliable Experimental Design (*FRED*) by using research methodology as methods and technologies in digital forensics enabling a more standardised methodology for designing and performing experiments.

FRED utilises six steps, which support the planning, implementation and analysis. These steps are:

1. Plan: Consists of defining goals, research questions, review of existing documentation or research, preliminary findings, valid assumptions, defining hypotheses, defining the test environment (accurate, repeatable, applicable)².
2. Implement: The experiment is conducted by performing defined user actions, and the set of actions are considered as "datasets", avoiding contamination (results should be a result of the actions in our experiment).
3. Evaluate: Identifies the effect of the experiment, which files, logs, metadata, databases are changed.
4. Repeat: The result is repeatable when the same actions are executed on the same data.
5. Analyse: Interprets the consistency of the results.
6. Confirm: Confirms that when investigating artifact X, user action Y results in outcome Z.

Horsman (2018a) claims using this framework will give factual accurate results, ensuring forensically sound testing.

If we compare FRED with the Reliable Validation Enabling Framework (*RVEF*), we can see that FRED is an element in the RVEF methodology phase (how are the experiments and datasets justified?).

3.7.1 Plan

The focus on the testing environment is crucial according to Horsman (2018a), and he suggests three options: flat analysis (forensic image analysis), reverse image

²If the environment is Windows 10, how to generate dataset (test results), it may not be applicable to generalise the results for Windows 11.

(restoring the image to another disk, which is used to start the device and perform experiments), and virtualisation of image. Which option to select must be based on the goal of the experiment.

3.7.2 Implement

This part is about running the experiment, defined as user actions in the plan. [Horsman \(2018a\)](#) emphasises that the results should be a result of the experiment, and not contaminated by other sources.

In the researcher's opinion, it could be challenging to control all variables when using a reverse image as we do not know all variables that may effect the result. Since the experiment is supposed to be valid for the data storage under investigation, the researcher of this work agrees that this is a good approach.

3.7.3 Evaluate

[Horsman \(2018a\)](#) suggests that assessing the modification timestamp of files can be used to identify which files are impacted by the experiment. In addition, process monitoring could help show which files, registry values, and processes are affected by the experiment by filtering to the process ID (*pid*) of the application used.

The results should aim to test the initial hypotheses. Is the result as expected and according to existing research? If not, does the researcher need to replan the design or rerun the tests? Are the use cases sufficient to generate the expected output within the dataset? Could the results have been encrypted? Are the changes stored in the cloud or in main memory only? Is the reason a user error?

If yes, can the results (dataset) be consistently reproduced?

3.7.4 Repeat

Experiments need to be repeatable, and get the same results. If this is not possible, then there is the need for re-designing within the planning stage. How many times the experiment has to be repeated is not easy to answer, but it should be repeated "beyond a negligible number of times" ([Horsman 2018a](#)), and it is when consistent results are first achieved that the analysis can start.

3.7.5 Analyse

The analysis is the interpretation of the results collected in the previous stages. The analysis should be able to explain how an artifact or application functions, and what the results of a user action are.

3.7.6 Confirm

This is a confirmation phase, that affirms that when focusing on one artefact, a specific user action results in a specific result. At this stage the testing procedures and methodology can be documented for peer review.

3.8 File Carving

This section describes previous research on file carving. File carving is a concept of recovering files from unallocated space, without the use of file system metadata (Garfinkel 2007). Typically, this is performed by searching for file signatures, or unique file patterns.

Karresand et al. (2019b) use file system allocation algorithms to create a probability map of where relevant user data should be located on the disk, and they use NTFS as a proof-of-concept. Instead of searching everywhere on the disk, it is better to search where the user data is most likely found. Karresand (2023) also describes that a high number of layers correspond to high file activity (Karresand 2023, p. 37), meaning these areas could be investigated to find previous remnants of files. Karresand (2023) extracts the initial *\$bitmap* file and extract the *\$bitmap* file after each file action in the experiment to create 128 areas where by, partitions are divided in 128 equal size areas (Karresand 2023, p. 60), mapping the allocated clusters.

Karresand et al. (2019b) observed a static location for the \$MFT table at 3 *GiB* into the NTFS volume at relative sector 786 432 on the disks that were a part of the experiments, yielding the starting sector for recovery of the metadata \$MFT system file. Similar findings were documented in 2020 by Karresand et al. (2020b) with the additional observation that the last part of the volume has a low allocation frequency. Even if a large file can fit contiguous in the last part of the volume, files will often be fragmented in the middle part.

The first 10-20 *GiB* of data has a low frequency of allocation, and typically relates to the files of the operating system. After this the allocation frequency increases rapidly, followed by a slow decrease towards the end of the NTFS volume (Karresand et al. 2019a).

Further research by Karresand et al. (2020a) also found that block storing decreases the fragmentation of files, while stream storing increases it from previously a low frequency. They describe that a file can be stored as a large known data block (block storing), or it can be stored as smaller blocks during downloading of a file with an unknown size (stream storing). This information can be used to increase the focus on searching for fragmented files in these areas when performing file

carving.

[Garfinkel \(2007\)](#) describes file carving as file reconstruction based on their content, not using the metadata that points to the content. He also claims that no file carvers can automatically reassemble fragmented files. Existing file carvers focus on contiguous files, but do not validate that the files are accurately recovered. [Garfinkel \(2007\)](#) used 300 active file systems from hard disks purchased on the secondary market from 1998 to 2006 (Garfinkel's Corpus)³. He also claims that interesting files in investigations are more likely to be fragmented, especially large log files and *PST* (email containers), *AVI*, *DOC*, and *JPEG* files. [Garfinkel \(2007\)](#) found that most bifragmented (two fragments) files had a gap of 8 blocks. Highly fragmented files were typically *DLL* and *CAB* files. They also found that the frequency of file fragmentation appears to decrease when the drive size increases. [Garfinkel \(2007\)](#) used JPEG decompression as object validation for JPEG fragments to build a JPEG carving tool with no false positives, which also used a bifragment gap carving algorithm to identify the splits. He also suggested a semantic validator, using the language to automatically validate data objects, but was not able to automate this approach. They did develop *MSOLE* and *ZIP* validators in addition to the JPEG validator. These validators were implemented in their file carver, which supports block carving or character based carving (supports objects embedded in containers) both for contiguous and fragmented files. [Garfinkel \(2007\)](#) describes that he was able to recover all Microsoft Word and Excel files in the *DFRWS* 2006 challenge that were split into two fragments. However, he also experienced false positives, such as recovered files that did not contain all the correct file fragments.

The research of [Garfinkel \(2007\)](#) is very interesting, and it shows it may be possible to recover fragmented unallocated files which depend on in-depth knowledge of each file format and implementation of suitable fragment validators. However, the precision, accuracy or reliability of these validators are not computed, nor is the error rate. Furthermore, the bifragment gap carving algorithm is not suited for finding files with more than two fragments ([Hilgert et al. 2019](#)).

[Garfinkel and McCarrin \(2015\)](#) suggest a hash-based carving method which use a hashset database (hashes of 4 KiB blocks) to assist with fragment assembly as a new approach for file carving of fragmented files. They found that many 4 KiB blocks happen to be present in many different files, which is a challenge for hash-based carving. Finding that a block hash exists in multiple files cannot be taken as evidence that all these files have been present on the hard drive. They propose

³This dataset meets the requirement of real life dataset ([The United Kingdom Forensic Science Regulator 2020](#)), but does not comply with the current GDPR regulation ([The European Parliament and of The Council 2018](#)).

a matching algorithm (HASH_RUNS) for hash-based carving that gives a specific range of contiguous sectors that can be mapped to a specific target file. The method is good for identifying known target files, even though there are false positives and the error rates were not computed.

[Gladyshev and James \(2017\)](#) suggest to use file carving for specific file types where relevant data most likely can be found, and this is suggested as a triage solution because of time and resource constraints. They call the approach for decision-theoretic (or best effort) file carving, where speed is more important than completeness. The decision-theoretic approach classifies the data as relevant (containing JPEG data) or non-relevant (does not contain JPEG data). The approach relies on being able to reliably detect JPEG blocks, not only the start or end blocks, but also those in between. As a proof of concept they developed a decision-theoretic JPEG file carver for large JPEG files. They used a trained *SVM* classifier that had 99 % true positive rate (recall), but had 33% false positive rate (hits that are incorrectly identified as JPEG blocks). The SVM classifier did not distinguish JPEG data from other data with high entropy. When a hit is found using sampling, the implemented carver jumps back and performs bounded sequential carving for the JPEG header and footer. If the header is not found within a specific limit, the carver switches to sampling mode again. If JPEG header and footer is found, it continues in bounded sequential mode assuming there could be more JPEG files. The approach does not solve the fragmentation problem, since their proof of concept requires non-fragmented JPEG files. The approach finds normally less files than traditional file carvers, and the speed is dependent on speed of disk, the number of target JPEG files, and the distance skip size for the sampling (larger skip size, less accuracy). However, for triage purposes it is more practical to see if there could exist unallocated interesting JPEG files on the disk as fast as possible.

[Hilgert et al. \(2019\)](#) suggest to use syntactical file carving (by taking advantage of the file format). The internal file format structure can be used to identify the start and end of *PNG* files, and the chunks that the *PNG* consists of have their own length fields. They use the chunk's length field to identify the complete chunk, and the assumed start of the next chunk, then they compute the chunks CRC value and compares it to the chunk CRC field. This is repeated until a chunk error is found or the end chunk is found for contiguous files. In the latter case the complete file is recovered. For fragmented files they utilise that most files are bifragmented, similar to [Garfinkel \(2007\)](#). For files fragmented in more than 2 fragments, they use a sliding window approach from the start fragment to the end fragments (or backwards). Then they can use the area between to identify the order of fragments and compare CRC values. They also suggest tracking of already validated chunks, and not assessing blocks that already are found to belong to

another file. This latter approach will increase file carving efficiency in many file systems, but will impact file carving on systems like APFS negatively since this file system may reuse blocks (deduplication) between multiple files (Hansen and Toolan 2017). Hilgert et al. (2019) compared their methods with other file carvers including Scalpel (Richard III and Roussev 2005), Foremost (United States Air Force Office of Special Investigations and The Center for Information Systems Security Studies and Research 2007), and PhotoREC (Grenier 2019). These latter tools were not able to recover fragmented files, while their syntactical approach was able to carve all files from 20 of 26 fragmentation scenarios. The syntactical approach can only be utilised for file formats that have enough internal structures that include fields that can validate their file fragments.

Ali et al. (2022) describe recovering *OOXML* documents from *RAM* using machine learning techniques (K mean, Hierarchical clustering, Mean shift clustering). They used the components present in *OOXML* word files as feature selection, where only the first five were static. Decompressing the components and using XML tags in order to identify and extract textual content. In order to use machine learning techniques they needed to pre-process the raw text. They tokenize all words, meaning every word is a single entity (removing stop words such as "are" or "is"). They use the frequency of words as features, and every word is converted to a vector. Their technique depends on the amount of data present in *RAM*, and they found that hierarchical clustering performed better than K mean clustering for both recall and precision. They used five datasets, where dataset S1 only included eight objects, which resulted in 90.625 percent recovery. Dataset S5 had 534 objects, which resulted in 92.134 percent recovery. The recovery rate increased if the files were open and edited compared to when the files were closed.

An et al. (2023) describe recovery of files from NTFS on Windows servers (from version 2016 or above) that use deduplication of files. In such systems, deleting a file does not necessarily mean the deduplication chunks will be unallocated, since they need to stay allocated if another file uses the same blocks. NTFS uses variable sized block units called chunks for deduplication. In Windows Server 2012 the deduplication used resident attributes, while Windows Server 2016 or above use non-resident attributes (deduplication block units external to the MFT table). When the deduplication process starts, the file gets unallocated (DATA attribute dataruns are zeroed and the bitmap representing the file clusters are zeroed). This mean the original file content will be part of the unallocated area, and traditional file carving techniques can be used as long as the clusters are not overwritten. However, the MFT record will be updated with a REPARSE_POINT attribute which will have the dataruns to the structures (FeRp, RbRp, DdRp) containing the information necessary to reassemble the file. All MFT records with a REPARSE_POINT will

also be recorded in the \$Reparse file in the \$R index (found in the INDEX ROOT and INDEX ALLOCATION attributes) which contains a MFT record number for each entry, allowing a more efficient way of identifying deduplicated MFT records. When deleting a file, the Delete.log which is part of the allocated area under System Volume Information, Dedup, Chunkstore, Chunkstor *UUID* directories, can be used to identify the deleted file, and the file can be recovered as long as the MFT record is not overwritten by using its REPARSE_POINT attribute to find the runlist to the necessary structures. This is not file carving, but ordinary parsing of deduplicated files. If the file is not found in the Delete.log file, then there is a need for file carving for the original file or for run lists structures (FeRp, RbRp, DdRp) pointed to from previous overwritten MFT records. This is similar to metadata carving, discussed in the next section.

3.9 Metadata carving

As described in the previous section, the file carving method tries to recover files based on signatures (header and/or footer) and not metadata. The process is rather easy for contiguous files that have signatures, but it is challenging to recover fragmented files (Garfinkel 2007). A better approach is to carve for metadata first, and use these to recover files (contiguous or fragmented). Ali et al. (2022) describe that the information needed to recover file content is located in the file metadata.

The approach is not new, using the "FILE" *ASCII* signature, which all NTFS MFT records start with to find MFT records is a well known method (Carrier 2005, p. 327). However, not all file systems have a universal static signature that can be used as signature.

Dewald and Seufert (2017a) were the first to propose a heuristic search pattern to identify inodes in the Ext4 file system (carving for inodes), without using the information from the superblock or group descriptor table. This technique is able to deal with the fragmentation problem, which is a challenge for traditional file carving (except for a few approaches for a few file types). Since there is no static signature for an inode, they use more complex patterns to identify the inodes. The identified inodes are used for file recovery. They developed a plugin module in Sleuthkit to implement their approach. The authors suggest two modes for their approach; content mode (only the block size must be provided) and metadata mode. For their inode carving they use the type field (4 most significant bits of the first 2 bytes) focusing on regular files and directories, evaluated timestamps, and the extent signature *0xf30a*. In order to connect file names with inodes in metadata mode, they use additional information about the group descriptor table, size of flex groups, number of inodes per group, etc in order to find the start and end offsets of the inode table. Dewald and Seufert (2017a) claim that the content

mode cannot map physical addresses to inode numbers, and they use the inode address as file names instead.

3.10 Timestamps

Willassen (2009) describes how timestamps rely on the clock, which could be more or less inaccurate (drifting, fail, adjusted, etc). Timestamps should not be used as evidence without justification. Different file actions (creation, modification, access) may impact and set timestamps on files (within metadata structures) in any file system, and a file can have a number of different timestamps. The author describes that the file action created will update created, modified, and accessed timestamps in a file system that uses three timestamps for a file. The file action Write will update modified and accessed. Reading a file will only update accessed. Willassen (2009) describes the possible action sequences for a file system with three timestamps per file as shown in Table 3.2.

Table 3.2: Action sequences for the simple file system. All other combinations are impossible without manipulation. The list is filtered down from Willassen (2009) original list.

Number	Timestamp order	Action Sequence
1	$(t_c < t_m < t_a)$	(Create, Write, Read)
7	$(t_c = t_m < t_a)$	(Create, Read)
10	$(t_c < t_m = t_a)$	(Create, Write)
13	$(t_c = t_m = t_a)$	(Create)

This led the researcher to the idea of using equality as a search mechanism to find timestamps, since many files are only created or written to, not read. File systems like NTFS (which uses four co-located timestamps) will not always update access, meaning only the actions create and write will impact created, modified and accessed timestamps as shown in Table 3.3.

Table 3.3: Action sequences for a simple file system with the update on access disabled, meaning the action Read has no impact. All other combinations are impossible without manipulation

Number	Timestamp order	Action Sequence
10	$(t_c < t_m = t_a)$	(Create, Write)
13	$(t_c = t_m = t_a)$	(Create)

The only set of actions that can result in only unequal timestamps are if the user at different times creates, then writes, and later reads the file, as shown in number 1

in Figure 3.2. If updating the last access is disabled⁴, then there will typically be equality between some of the timestamps. Which actions update timestamps are important for digital forensic investigators, and are part of file metadata analysis.

3.11 Summary

This section has provided an overview of some relevant research to the legal standard for digital forensics (Daubert), methodology and experiments (FRED), validation and verification, peer-review, file carving, metadata carving, and the issues with fragmented files.

Legal standards, methodology and experiments, validation, verification, and peer-review are relevant for file system metadata research because the court needs to assess the applicability of the applied scientific research. File carving is relevant because this technique is often used without taking advantage of metadata carving, resulting in less accuracy for recovering fragmented files.

The next chapter describes the results of the works performed in this study, and also consists of a summary of the motivation, contributions and limitations related to each publication included in the thesis.

⁴Note that last access will always be stored on Create.

Chapter 4

Contribution and publication summaries

4.1 Publication A - Using the object ID index as an investigative approach for NTFS file systems

Rune Nordvik, Fergus Toolan and Stefan Axelsson. “Using the object ID index as an investigative approach for NTFS file systems” In: *Digital Investigation Volume 28, Supplement*. April 2019, Pages S30-S39. DFRWS 2019 Europe — Proceedings of the 19th Annual DFRWS Conference. DOI: <https://doi.org/10.1016/j.diin.2019.01.013> (Nordvik et al. 2019b)

4.1.1 Motivation

The link files in Windows may say a lot about user activity (Parsonage 2008), and these link files also have object ids within them. The attribute OBJECT_ID in NTFS master file table is used as an alternative way of identified files using the object id instead of the file name (Carrier 2005, p. 335). Carrier (2005, p. 335) described that there were no tools that searched for files based on the object id, and this motivated the main author to see if we can use the *\$ObjID* index as a starting point to find all MFT records that have an OBJECT_ID attribute, and this way identify user activity even if the link files were deleted or not available¹.

¹There are typically no link files on an external storage device.

UUID, rfc 4122

time_low 32 bits	time_mid 16 bit	time_high_and _version 16 bits	clk_seq_hi _res 8 bits	clk_seq_low 8 bits	node 48 bits
---------------------	--------------------	-----------------------------------	---------------------------	-----------------------	-----------------

UUID version 1, rfc 4122

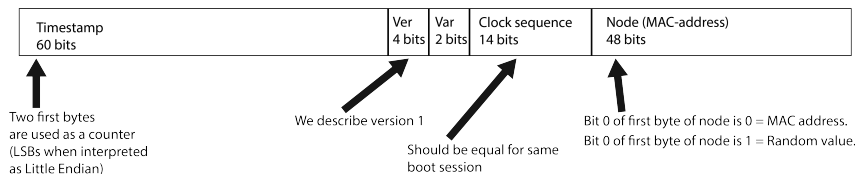


Figure 4.1: Structure of an Object ID UUID version 1.

4.1.2 Research contributions

This research shows the meaning of the system file *\$ObjID* in NTFS, and how it can be used to connect the device to other computers, but also identify files that are typically created or opened by users, and in which boot session the Object IDs (OIDs) were created. On system volumes in cases where the suspect has deleted link files to hide traces, we can use the *\$ObjID* index to identify files that are related to user activity. For instance if the user double clicks to open files from File Explorer, all these files will have OIDs in the *\$ObjID* index.

The first 60 bits in Figure 4.1 is a timestamp field using 15th of October 1582² as the epoch using a granularity of 100 nanosecond intervals (Leachi et al. 2005). The timestamp describes the time of the last boot. The two least significant bytes of this timestamp when read as little endian is a counter used to define the object id order. The easiest way to convert this timestamp to a human readable format is to first subtract 0x146BF33E42C000, the number of 100 nanosecond intervals between 15th of October 1582 and the 01st of January 1601, from the timestamp under investigation and then use a FILETIME converter. The next 4 bits describes the version, and the Object ID is using version 1 described in RFC4122 (Leachi et al. 2005). The next two bits describe the variant (the type used in RFC4122 is 10_b), followed by the 14 bits describing the clock sequence. The clock sequence will be equal for all OIDs created during the same boot session, and created OIDs in the same boot session can be ordered by using the object id order.

The last 6 bytes of an *OID* contain a MAC-address for a selected device on the computer, see Figure 4.1. This *MAC*-address could be from a network enabled device, or a random value if no network device is available. In the latter case bit

²The date of the Gregorian reform.

0 of the first byte of the node is set. If a valid unicast MAC address is used, this bit will have the value 0 (Leachi et al. 2005). In this case it is possible to identify the Organisation Unique Identifier (*OUI*). Using a database containing vendor OUIs, the investigator will find the vendor name of the network card used. This may help identifying the computer, and in some cases an organisation may have an inventory database that includes this information. In any case the investigator may use simple commands, such as *ipconfig /all* to list the network devices including their MAC address.

Our approach helps in identifying what is relevant to acquire and analyse from a computer. This is because files with OIDs typically have either been created or opened by the user. Since the OID also contains the timestamp of the latest boot session before the OID was created, and the OIDs order, this will help reconstruct events. The approach may also detect timestamp manipulation, for instance if all Standard Information Attribute (SIA) timestamps are before the boot time in the boot session the OID was created. This should not be possible because the record modified timestamp must be after or equal to the creation of the OBJECT_ID Attribute.

We also suggest a new method for matching the \$ObjID index with their respective MFT records as shown in Figure 4.2. In the first hex dump at the top the MFT Record reference field is highlighted in green background. The two most significant bytes (here 0x0001 LE) is the sequence number of the MFT record when this object id entry was created, and it must match the sequence number of the MFT record in order to map the object id with the MFT record. The next six bytes of the reference is the MFT record number (here 0x27 LE or 39 in decimal). Since every MFT record is 1024 (0x400) bytes as standard, we can easily find the offset to the start of any MFT record from the extracted MFT table. In the middle hex dump in Figure 4.2 the MFT record is highlighted with a blue background, and the sequence number can be found in the green highlighted area at relative offset 0x10 and is 2 bytes in size. The value of the sequence number is 0x0001, meaning it can be mapped since it matches the sequence number from the object id entry. The last hex dump in Figure 4.2 show the OBJECT_ID attribute for this MFT record, and highlighted in blue we can see the Object ID key. This must match the Object ID key in the \$ObjID index in order to be considered a valid mapping.

4.1.3 Technical contributions

We created a prototype OID parser, which we call the NTFSObjIDParser. It is written in C++ using the *QT* graphical libraries. It requires two files from a NTFS volume, the \$ObjID (both the Index Root Attribute, and the Index Allocation Attribute) and the \$MFT, which the user must extract from the volume. This can be

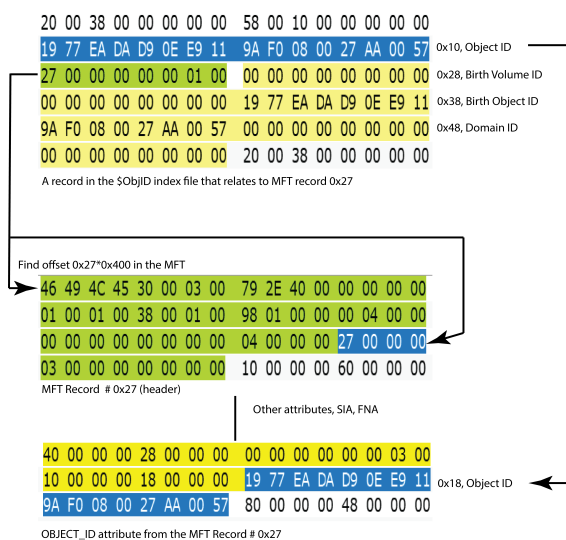


Figure 4.2: Connecting ObjID index record with NTFS MTF record.

done by using the `icat` command from Sleuthkit. The prototype reads each entry in the ObjectID index, and parses the corresponding entry in the MFT table. This way we can use the index to find the MFT entry and locate additional metadata, such as the timestamps assigned to the Standard Information Attribute (SIA) and any File Name Attribute (FNA). We also show the `OBJECT_ID` attribute that is part of the MFT record, and it should if correctly identified be equal to the Object ID key from the `$ObjID` index entry.

4.1.4 Limitations

One limitation of this research was that the researchers did not perform experiments on Servers or Workstations that were part of a domain, and therefore observed only zeros for the Domain Object ID.

4.1.5 Summary

In this publication we identified a new investigation method for identifying user activity, using the `$ObjID` index to identify the files the user has accessed, in which order, and in which boot session. We also showed how an external storage device can be linked to a particular computer by utilising the MAC address found in the node field of the Object ID.

Our observations show that:

- OIDs were always created when using File Explorer or LibreOffice to open

files that did not already have OIDs assigned, but not all tested tools (command prompt, Veracrypt, Notepad) generated OIDs on user activity.

- MAC addresses were stored in the last 6 bytes of the Object ID index, which can be used to connect the external device to the computer used. However, on computers without network cards a random value may be used (Parsonage 2008).
- Deleting files also deletes any connected object id entry in the \$ObjId index, but may leave the Object ID attribute in the unallocated MFT record.
- Tested commercial tools showed information about OIDs, but not all the information that is relevant for an investigation. The CrossVolumeMoveFlag, and the Object ID order was missing from both EnCase and X-ways. Autopsy did not support OIDs.

4.2 Publication B - Generic Metadata Time Carving

Rune Nordvik, Kyle Porter, Fergus Toolan, Stefan Axelsson and Katrin Franke. “Generic Metadata Time Carving” In: *Forensic Science International: Digital Investigation Volume 33, Supplement*. July 2020, Pages 301005. DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA. DOI: <https://doi.org/10.1016/j.fsidi.2020.301005> (Nordvik et al. 2020b)

4.2.1 Motivation

The main author and the second author discussed how research on search algorithms can help law enforcement. The main author described that it could be very useful if the search algorithms identified timestamps, since all metadata entries related to files contain a set of co-located timestamps. Identifying the file metadata would help recover files using the metadata found, and will solve the file carving challenges with fragmented files (assuming the fragments are not overwritten), or where a file system volume is reformatted with another file system. The researchers also hypothesised that in NTFS the set of timestamps belonging to a file often has multiple equal timestamps, which is stated by the simplified general file system with three timestamps in the research of Willassen (2009).

4.2.2 Research contributions

The Generic Metadata Time Carving (GMTC) approach enables the recovery of inodes or attributes related to time, either in the inode table, the MFT table or the journal. This is important in order to identify and recover previous file systems, individual deleted files, including fragmented deleted files. This approach should

be tried before attempting file carving if an Ext4 or NTFS file system has been reformatted with another file system.

For an Ext4 or NTFS file system that is not reformatted, the GMTC approach may be used to show metadata for all inodes or MFT records (allocated or de-allocated) and it is possible to filter the results based on inode/MFT record numbers to identify multiple versions of the same inodes/MFT records. Using this approach it should be possible to detect inodes/MFT records that are unallocated and find the allocated version of the inode/MFT record in the journal. This is possible because the GMTC approach searches everywhere on the storage media, not only in the inode table.

4.2.3 Technical contributions

The researchers developed a potential time carver using the C programming language, and additional files system validation tools using Python scripting. The cPTS tool uses a search algorithm to compare data based on a pre decided granularity (4 or 8 bytes), which compares an element with the next set of elements of the same size. The cPTS tool gives a list of the position of where equal elements, potential timestamps, may be found. Then we used file system validators developed in Python to verify if each element identified is a valid metadata structure for a file or directory. This increased the precision of the end results.

4.2.4 Limitations

The suggested method depends on equality between the timestamps, using the granularity of the timestamp used. This means that if all timestamps in an inode or MFT record differ, then this inode/record will not be found. Fortunately, the File Name Attribute in Windows very often has more than one equal timestamp, and therefore the record will be found. This is not true for Ext4, which means a decreased recall.

We had some issues in connecting the directory entries to the Ext4 inodes, because we did not want to consider the group descriptor where all the bitmaps and the inode pointers are stored. We assumed that when we found a directory inode and opened its content, we could look at the current directory entry (.) and its inode number. Then we can say that the next inode following this directory inode would increase its inode number with 1. Then we could match the directory entry list with each filename and inode number found within the directory entries, as illustrated in Figure 4.3. This is valid if the inode tables are contiguous following each other, but if the inode tables are fragmented, then this may fail. Although, when parsing the next directory inode the inode number counter will be updated again.

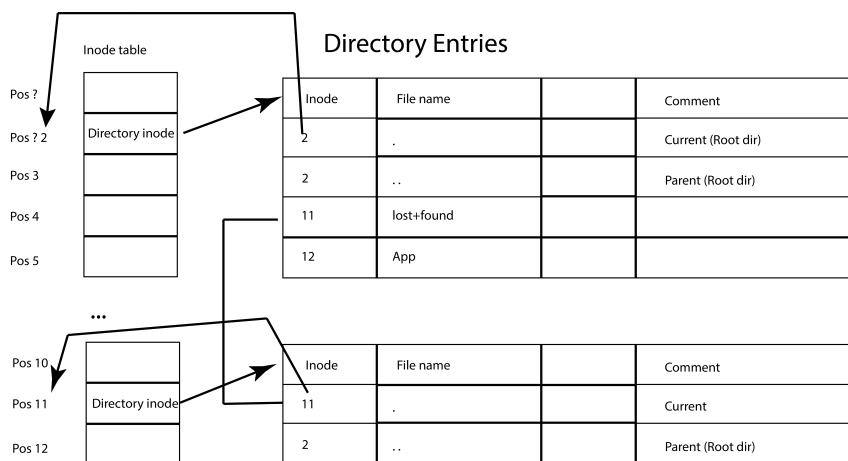


Figure 4.3: Connecting inodes with directory entries.

The GMTC approach is currently not good if the previous file system was re-formatted with the same file system, especially since inode or MFT tables are overwritten by the same new file system.

4.2.5 Summary

In this publication the researchers have used equality as a search mechanism for timestamps, and using the features of timestamps as a signature to identify metadata for files. Since equality is first used to identify possible metadata for files, we experienced also many hits that were not metadata related to files. Therefore, we filter out all these false positives using semantic validators tailored for the file system under investigation (currently supported validators are NTFS and Ext4). Using the metadata, we can easily recover the data content if it is not overwritten, even if the file is fragmented.

4.3 Publication C - Timestamp prefix carving for filesystem metadata extraction

Kyle Porter, Rune Nordvik, Fergus Toolan and Stefan Axelsson. “Timestamp prefix carving for filesystem metadata extraction” In: *Forensic Science International: Digital Investigation Volume 38*. September 2021, Pages 301266. DOI: <https://doi.org/10.1016/j.fsidi.2021.301266> (Porter et al. 2021)

4.3.1 Motivation

The authors found a weakness in publication C, the GMTC approach, which requires timestamps to have equal values in order to get a hit for a potential timestamp. Therefore, we focus in this paper on approximate equality. A timestamp may just deviate partly from another timestamp, and timestamps are usually presented as a number since a specific epoch. The most significant bytes (*MSB*) in a set of co-located timestamps (the prefix) is usually equal if the timestamps do not deviate too much.

4.3.2 Research contributions

The new approach of comparing the equality of a number of the MSB will increase the recall of potential timestamps, and decrease the number of false negatives. This allows for more accurate and reliable results than the previous GMTC method by increasing the recall.

4.3.3 Technical contributions

Improvements were made to the *cPTS* to allow for prefix comparison. If the granularity of the timestamps searched for is 4, for instance timestamps in Ext4, the tool is able to compare the selected X MSB for each of these, and when X=4, the method is the same as the GMTC approach. X may not be higher than the granularity of the timestamp and it must be higher than 0. If X is lower than the granularity, the recall increases, and precision decreases. However, the Python file system validators test each hit from the *cPTS* and increase the precision by excluding false positives.

4.3.4 Limitations

We still have issues with the accuracy of connecting the directory entries with the inodes if the inode tables are fragmented.

4.3.5 Summary

This paper improves the GMTC approach by introducing approximate equality. The X most significant bytes of the potential timestamps are compared for equality which increase the recall but keep the precision due to very strict file system validators. We still use tailored file system validators to filter out false positives, which increases the precision. High recall makes sure we include more true positives, potentially including metadata for additional files with evidential value.

4.4 Publication D - Its about time—Do exFAT implementations handle timestamps correctly?

Rune Nordvik and Stefan Axelsson. “Its about time—Do exFAT implementations handle timestamps correctly?” In: *Forensic Science International: Digital Investigation Volumes 42-43*. October-December 2022, Pages 301476. DOI: <https://doi.org/10.1016/j.fsidi.2022.301476> (Nordvik and Axelsson 2022)

4.4.1 Motivation

When investigating exFAT file systems the main author has observed a difference in storing UTC offsets between Windows and MacOS. In Windows the main author observed the UTC offset was stored as local time including any daylight savings, however, in MacOS it stored the UTC offset and the timestamps differently. In addition we were not sure how the exFAT implementations in Linux stored timestamps.

4.4.2 Research contributions

This research shows that while exFAT specifications are available (Microsoft 2021), there is no guarantee for accurate and reliable file system implementations. In addition, we found that commonly used digital forensic tools contain bugs that impact the timestamp interpretations when interpreting the exFAT file system. Especially, digital forensic tools often fail if invalid UTC offsets are stored, like the Linux exFAT fuse driver does. Each timestamp has a UTC offset field. When the UTC offset field value’s most significant bit is set to zero, then the field is not in use (not valid). The specification describes this bit as the OffsetValid field, and the other 7 bits as the OffsetFromUtc field (Microsoft 2021). This is the first publication about exFAT where file system implementations are observed on different operating systems. We found differences in how the file system drivers implemented the exFAT specification resulting in inaccurate metadata when users use multiple operating systems for the same external storage device, and we found differences in how digital forensic tools interpreted the exFAT file system. In Table 4.1 we have summarised the experiments for all file system drivers included in the experiments, only using one time zone as an example. The table field Stored TZ is the hex value found in the Create UTC offset field on the raw disk and in parenthesis the value is converted to the UTC offset if valid. The table field Real TZ show the real timezone the computer used during the experiment.

The results in Table 4.1 show that if we know the stored timezone hex value we can identify the timezone used only if MacOS or Windows was used, and only if we know which of these OSes were used. If a user uses the Linux fuse driver to

Driver	Base TZ	Action	Stored TZ	Real TZ	Observations
MacOS	Europe/Oslo	Created	0xFC (UTC-1)	UTC+1	100
Linux Native	Europe/Oslo	Created	0x80 (UTC+0)	UTC+1	100
Linux Fuse	Europe/Oslo	Created	0x00 (Not valid)	UTC+1	100
Windows	Europe/Oslo	Created	0x84 (UTC+1)	UTC+1	100

Table 4.1: Summary for experiments creating files - All tested file system drivers limiting to one particular timezone.

create a file, this resulted in a not valid UTC offset. If a user then uses MacOS and TextEdit to change the same file, the invalid UTC offset value was assumed by the file system driver to be equal to the timezone of MacOS and was converted to a “valid” value based on this possible wrong assumption. However, using Linux to change the original file resulted in a new file because of the atomic storage feature, effectively losing the original UTC offset fields as soon as the original unallocated file is overwritten. The UTC offset field for the new file was set to either 0x80 for the Linux native exFAT driver or 0x00 for the Linux fuse exFAT driver.

We found that Autopsy v. 4.19.3 used the initial timezone used when attaching the storage device as if this timezone was the stored raw value, not interpreting the UTC offset fields at all. Changing the timezone in Autopsy later resulted in converting the timestamps assuming the initial value was the true timezone and not using the UTC offset fields.

FTK Imager v. 4.5.0.3 and 4.7.1.2 were able to interpret all timestamps with a valid UTC offset, and did not show timestamps with an invalid UTC offset. Timestamps with valid UTC offsets were converted correctly to UTC+0. However, these FTK versions did not show all directory entries.

We found that EnCase Forensics v. 22.1 was able to show timestamps for all valid UTC offset, but misinterpreted the invalid UTC offsets by assuming they mean UTC+0.

X-Ways v. 20.04 SR-4 was able to show timestamps with an invalid UTC offset using LT (Local Time) as the timezone indicator, and showed timestamps with a valid UTC offset correctly. However, it misinterpreted if there were a mix between valid and invalid UTC offsets for the same file. In this use case it tried to convert the timestamp with an invalid UTC offset value to the selected timezone by making assumptions about the invalid UTC offset.

4.4.3 Technical contributions

The results of the works in this study were assessed using three commercial digital forensic solutions, and one open source digital forensic solution. However, no additional tools were created.

4.4.4 Limitations

The atomic storage was not initially considered, where the application or a library store files temporarily on change, and on success de-allocates the original file, and then renames the temporary file using the original filename as target.

The authors assumed that simple text editors like Notepad, TextEdit, and Gedit would not implement atomic storage since these editors are not typically used for writing large documents. When Maxim Suhanov contacted us and claimed the Linux results could be wrong due to atomic storage, we performed more experiments and found that he was correct about Gedit and atomic storage.

Having used a mix of MacOS Monterey and MacOS Mojave in the experiments related to MacOS could impact the results. However, they use the same exFAT driver extension v 1.4 and do not impact the validity of the experiments.

The authors recommended using *FTK* Imager or XWays to parse exFAT file systems if the UTC offset fields were valid. However, since FTK imager does not show all directory entries, we cannot recommend this tool for exFAT parsing. We were informed by Maxim Suhanov about this bug in the latest FTK Imager v. 4.7.1.2. In this thesis the new updated and unpublished version of the paper is included. A corrigendum to the original version of the paper was published (Nordvik and Axelsson 2023), which describes that Gedit uses atomic storage, and it is therefore not the file system driver's fault that files that were changed got new created dates. The metadata from the original file was unallocated and the temporary file used during the atomic storage was renamed to the original file name. We also observed that 98 percent of the changed files lost their unallocated old entry when changing the files using the Gedit tool. However, Notepad and TextEdit did not use atomic storage.

4.4.5 Summary

In this paper, the authors evaluated the implementation of exFAT file system on Linux, MacOS and Windows. We have observed that the different implementations may deviate from the exFAT specifications, which could impact the accuracy of timestamps on a device if used on different operating systems or impact the accuracy of the interpretation performed by digital forensic tools.

4.5 Summary

This chapter has focused on providing a brief summary of the research motivation, results (contributions), and limitations of each publication. In chapter 5 a discussion of the results related to the overall research questions are given.

Chapter 5

Discussion

The focus of this discussion is how the presented research is related to the following research questions.

- R1: To what degree can user activity be documented from non-OS volumes (external storage devices) using only FS metadata from the file system?
- R2: To what degree can an external storage device be used to identify the computers it has been attached to by only assessing metadata from the file system?
- R3: To what extent can file system metadata reliably identify deleted files?
- R4: To what extent can the reliability and accuracy of file system parsing performed by tools be assessed?

5.1 General weaknesses with methodology

The published scientific papers all have methodology weaknesses. Summarised these weaknesses are:

- Only a few operating systems have been included for each experiment. Windows has many versions as does MacOS and Linux. We selected OS versions based on popularity and availability.
- Only a few applications or digital forensic tool versions were tested or used for experiments. Commercial tools are expensive, and we had only a limited number of licenses available for this research. We selected the latest version available during the research.

- The data sets are usually small. This is due to the unavailability of large relevant data sets with documented base truth, and the resource limitations to create larger data sets.
- Synthetic data sets are used, instead of real data sets, to comply with *GDPR* and national legislation. There is no need to use private data to perform experiments on file system metadata, since the creation of synthetic data behaves in an identical manner to real data.
- There are page limits for some of the conferences or journals where the papers were submitted, impacting the prioritisation of results on behalf of a detailed methodology section.
- Prototypes developed should be considered as proof of concept, and should not be used in criminal investigations before the LEA has validated the prototype.

In addition to these listed weaknesses, included are subsections describing other methodology weaknesses when appropriate.

5.2 General strengths with methodology

This research has included:

- The *IMRAD* method for organising the scientific papers ([Wu 2011](#)).
- Previous relevant research.
- A methodology section in each paper describing the methodology used.
- Data sets when appropriate
- The scientific papers as open access publications in a peer-review journal.

5.3 R1: Identifying user activity using FS metadata

Two papers directly address the question of identifying user activity using FS metadata. These are:

- Publication A: Using the object ID index as an investigative approach for NTFS file systems([Nordvik et al. 2019b](#)).
- Publication D: Its about time—Do exFAT implementations handle time-stamps correctly? [Nordvik and Axelsson \(2022\)](#)

5.3.1 Publication A: Using the object ID index as an investigative approach for NTFS file systems.

The strength of the approach documented in [Nordvik et al. \(2019b\)](#) is that an Object ID entry is created if the file does not have an Object ID already assigned when using File Explorer, and tools like LibreOffice, and most likely other graphical user interface programs that use the same *API* to open a file. This is also the case in Windows 10 when creating a file using File Explorer and Libreoffice.

Very often users use File Explorer to open files, or they may use File Open, File Save, or File Save As functionality of their application. These actions will result in new Object ID entries if they do not exist for the particular file. Therefore, the Object ID index is a good source for documenting user activity related to files. Developers creating Windows programs often use the Windows API when creating these dialogs instead of programming all the functionality themselves. These developers may not be aware of the Object ID feature included in their own programs.

File activities that do not create Object IDs

When extracting files from a zip archive using File Explorer, no Object ID entry is created. Copying files using File Explorer (CTRL drag), or copying using the command line does not leave traces in the Object ID index. Using Notepad in Windows 10 and the Save As feature does not store a new Object ID. However, Notepad does create an Object ID in Windows 7 when using Save As.

Moving files and Object IDs

As long as the Birth Volume Object ID (*BVOID*) is not equal 0, then the Object ID and the Birth Object ID are preserved, while the least significant bit of the BVOID is set if moved to another NTFS volume (the *CrossVolumeMoveFlag*). If the BVOID is 0, then no Object IDs are preserved and new Object IDs are created.

This means we cannot detect if a previous file that had an Object ID is moved to another volume as long as the BVOID is zero. We identified why an Object ID entry can contain a BVOID equal to zero, which is due to a missing Object ID for the \$Volume file.

Special weaknesses of implemented methods

In order for a NTFS volume to get assigned an Object ID for the \$Volume system file, it is necessary to reboot after formatting the volume, and the volume needs to be attached to the computer or virtual machine during this first reboot. When using virtual machines in Virtual Box, we experienced that the external volume

was released to the host, and not attached to the guest virtual machine during the first experiments. This resulted in a zero value for the Object ID for the *\$Volume* file, and for all the Birth Volume Object IDs in the Object ID index. Investigators should not assume manipulation attempts if they detect zeroed out Birth Volume Object IDs for all entries in the Object ID index, since the reason could just as well be that the *\$Volume* file was not assigned an Object ID.

5.3.2 Publication D: Its about time—Do exFAT implementations handle time-stamps correctly?

[Nordvik and Axelsson \(2022\)](#) make several interesting observations. The use of graphical user interface apps on MacOS leave fork files on the exFAT volume for each file changed in the same directory as the changed file. Resource fork streams give more information about files, and are used by macOS in addition to data forks ([Wani et al. 2020](#), [Mahajan et al. 2014](#)). However, the exFAT file system does not support resource forks, therefore, the forks are stored as additional files. These fork files start with `._` and the original file name. The fork files are created based on normal user activity, and the forks also contain information about which application was used to change or create the file ([Mahajan et al. 2014](#)). Also mounting and unmounting an exFAT volume will leave traces on Windows, where the System Volume Information directory is created, and on MacOS where the *.fsevents* and *.SpotLight-V100* directories are created. Attaching and mounting removable USB storage are typical user activities. Further, all the creation, opening, and changing of files are typical user activities, especially when related to removable external storage.

Special weaknesses of implemented methods

It is difficult to reliably interpret timestamps based on user activity when the external storage has been used on multiple OS. This is because a change using Gedit in Linux will replace the original file with a new temporary file with all new timestamps due to the use of atomic storage. In addition the exFAT fuse driver sets the `UtcOffset` fields using invalid values, which the other OSes and DF tools may interpret incorrectly.

5.4 R2: Connecting storage devices to computers using FS metadata

Connecting an external device to a specific computer is important in order to seize relevant computers during an investigation of a criminal case. The scientific papers listed in this section give answers to this research question.

- Publication A: Using the object ID index as an investigative approach for NTFS file systems (Nordvik et al. 2019b).
- Publication D: Its about time—Do exFAT implementations handle time-stamps correctly? (Nordvik and Axelsson 2022)

5.4.1 Publication A: Using the object ID index as an investigative approach for NTFS file systems.

The \$ObjId system file is located on every NTFS volume, and the Object ID is using the MAC address of one of the network cards installed in the computer. It should be feasible to find the computer that first created the object ID for a file, as long as this computer is available. It could be one of several computers in a suspect's company, or one of several computers from the suspect's apartment. Each MAC address also has an Organizationally Unique Identifier (*OUI*), which is the first 24 bits of the MAC address (IEEE 802 2014). The OUI can be used to identify the company that created the network device. This can also help in identifying the computer. The least significant bit of the first byte (octet) in the OUI is if set used for a group (multicast) or if not set used for an individual address (IEEE 802 2014). However, for the Object ID the bit if set indicates that a random value is used, and if not set a valid MAC address is used (Leachi et al. 2005). It is only when a valid MAC address is used that we can connect the external device to the computer used to create the Object ID.

Special weaknesses of implemented methods

The node part of the Object ID may be a random value, for instance if no network card was installed. However, this is not usual on current Windows computers. We may not have access to the computer used when a particular Object ID was created, but the information can be used for justification for seizing these devices from the suspect. Another weakness is that MAC addresses may be changed by the user or the user may have changed the network card.

5.4.2 Publication D: Its about time—Do exFAT implementations handle time-stamps correctly?

There are patterns that show which OS has been used to store files on an exFAT volume. When mounting the exFAT volume on Windows we can find the System Volume Information directory within the root directory. When mounting the storage on MacOS we find the *.fsevents* and the *.SpotLight-V100* directories within the root directory. In addition, fork files will be found if graphical user interface apps are used to change files on MacOS. When using the exFAT fuse driver files created use the UTC offset 0x00 (field not valid), and if the native driver is used we

will identify that UTC offset value 0x80 (*UTC+0*) is used. If the *System Volume Information*, the *.fseventd* and *.Spotlight* directories are not present, then the UTC offset value 0x80 will indicate Linux usage.

Special weaknesses of implemented methods

When a removable device with exFAT has been used on multiple OSes, it requires manual verification in order to describe which file has been created on which OS. We do not find any indication of which computer has been used within the FS metadata, only which OS. The information from the fork files from MacOS will describe which tools have been used, and this may help identifying which MacOS computer has been used.

5.5 R3: Using FS metadata to identify deleted files

As long as the block pointers or data runs are available in the file system metadata it should be simple to recover unallocated deleted files. Therefore, finding previous metadata structure like the inode table or MFT table would increase the reliability when recovering files, especially fragmented files, and we would be able to connect other relevant metadata such as timestamps and file names. The scientific papers listed below answer this research question.

- Publication B: Generic Metadata Time Carving ([Nordvik et al. 2020b](#)).
- Publication C: Timestamp prefix carving for filesystem metadata extraction ([Porter et al. 2021](#)).
- Publication D: Its about time—Do exFAT implementations handle timestamps correctly? ([Nordvik and Axelsson 2022](#))

5.5.1 Publication B: Generic Metadata Time Carving and Publication, C: Timestamp prefix carving for filesystem metadata extraction

The GMTC approach is based on the likelihood of several of the co-located timestamps being equal. This may not always be true, meaning we can miss some potential timestamps related to files.

Another argument is that we just as easily could use the FILE signature in NTFS to find MFT records. This is true for working records in the MFT, however, some records may use the signature BAAD indicating corrupt MFT records or an anti forensic technique to hide data ([Lin 2018](#)). Further, the GMTC approach is generic, meaning it can be used on different file systems that have co-located timestamps. We have shown this using Ext4, which does not have a special signature

for all inodes ([Dewald and Seufert 2017b](#)). The GMTC approach is improved in the publication D, since we allow for approximate equality by comparing a number of the most significant bytes in each co-located timestamp.

The approach is most suitable for recovering files from a damaged or overwritten (reformatted) file system.

Weakness of identifying deleted NTFS files

Deleted files that do not have any equal, or approximately equal, timestamps in the SIA and the FNA, will not be detected. However, currently the FNAs are not frequently updated, and normally contain at least two equal timestamps. Our approach will, by finding the FNA, be able to identify the SIA based on the static distance between attributes and then find the DATA attribute for an ordinary file based on the length of each previous attribute.

Weakness of identifying deleted Ext4 files

There may be an issue connecting/mapping the inode and the directory entry (file name) in Ext4, especially in cases where previous directory entries are overwritten/damaged. Our mapping method depends on finding directories in order to have a known location for the estimation of inode numbers and to create a dictionary of filenames and inodes.

5.5.2 Publication D: Its about time—Do exFAT implementations handle time-stamps correctly?

When we assessed related work, we identified that [Vandermeer et al. \(2018\)](#) had already found a method of detecting if an unallocated directory entry in exFAT was deleted, or just moved or renamed. Renaming a file will set the previous set of directory entries for this file to unallocated, then a new set of directory entries are created including the new name, however the cluster(s) belonging to the file are not changed in the allocation bitmap. A similar thing will happen if a file is moved; the previous set of directory entries are set as unallocated in the source directory; a new set of allocated directory entries are created within the target directory; and the cluster bitmaps are not changed in the allocation bitmap.

On file deletion the file's set of directory entries are set to unallocated by setting the most significant bit in the directory entry type to zero. Then the allocation status of the clusters belonging to the deleted file is set to zero in the allocation bitmap.

There may be use cases where all of a deleted file's clusters are allocated by a new file. In this case one may interpret that a move or rename of the file has happened.

Field Name	Offset	Size	Value example
CreateTimestamp	0x08	0x04	0x54717040
LastModifiedTimestamp	0x0C	0x04	0x54717040
LastAccessedTimestamp	0x10	0x04	0x54717040
Create10msIncrement	0x14	0x01	8B
LastModified10msIncrement	0x15	0x01	8E
CreateUtcOffset	0x16	0x01	0x80
LastModifiedUtcOffset	0x17	0x01	0x80
LastAccessedUtcOffset	0x18	0x01	0x80

Table 5.1: File directory entry (The fields are mandatory)

However, it should be possible to identify this by assessing the timestamps. [Vandermeer et al. \(2018\)](#) states that the creation time should be the same, while the modification timestamp of the new directory entry set is more recent. Unfortunately, this statement is not supported by experiments or references.

The thesis author has after the release of the paper made initial observations on how the timestamps are updated in Windows if the user renames a file located on an exFAT file system using File Explorer in Windows 10. The base exFAT *USB* storage was from creating files in Linux Ubuntu 20.04 with its native exFAT driver. The first file name in [Figure 5.1](#) shows the unallocated directory entry set, initially created on Linux using the native Ubuntu 20.04 exFAT driver, after renaming the file using Windows 10. [Figure 5.2](#) shows the new allocated directory entry set. Renaming a file in the same directory, will not result in differences of any timestamps when comparing the unallocated with the new allocated directory set, as can be seen in [Figure 5.1](#) and [Figure 5.2](#), where all timestamps have the value 0x54717040 (LE) both the unallocated directory entry set and the allocated directory entry set. The highlighted area contains the fields shown in [Table 5.1](#). However, these equal timestamps are from when the renaming was performed, not the original timestamps of the file.

Weaknesses of identifying deleted ExFAT files

As observed in the initial testing of renaming files it will be difficult to separate a deleted file with a renamed file by interpreting the differences in timestamps between unallocated and allocated directory entry sets. However, one hypothesis is that if a file has been deleted and another file has reused the clusters, we should be able to identify that the previous file has been deleted because there are differences

05 04 74 6F	20 00 00 00	40 70 71 54	40 70 71 54	to	@pqT@pqT
40 70 71 54	8B 8E 80 80	80 00 00 00	00 00 00 00	@pqT<	žēēē
40 03 00 23	28 4B 00 00	2A 00 00 00	00 00 00 00	@	# (K *
00 00 00 00	9F 01 00 00	2A 00 00 00	00 00 00 00	Ÿ	*
41 00 44 00	32 00 30 00	32 00 32 00	2D 00 30 00	A D	2 0 2 2 - 0
33 00 2D 00	30 00 32 00	54 00 31 00	36 00 2D 00	3 -	0 2 T 1 6 -
41 00 31 00	31 00 2D 00	35 00 32 00	2D 00 74 00	A 1	1 - 5 2 - t
7A 00 2D 00	30 00 2D 00	66 00 69 00	6C 00 65 00	z -	0 - f i l e
41 00 31 00	2E 00 74 00	78 00 74 00	00 00 00 00	A 1	. t x t
00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00		

Figure 5.1: Unallocated directory set after rename.

85 05 29 F4	20 00 00 00	40 70 71 54	40 70 71 54	...)ô	@pqT@pqT
40 70 71 54	8B 8E 80 80	80 00 00 00	00 00 00 00	@pqT<	žēēē	
C0 03 00 3C	B9 B5 00 00	2A 00 00 00	00 00 00 00	À	<¹µ	*
00 00 00 00	9F 01 00 00	2A 00 00 00	00 00 00 00	Ÿ	*	
C1 00 44 00	32 00 30 00	32 00 32 00	2D 00 30 00	Á	D	2 0 2 2 - 0
33 00 2D 00	30 00 32 00	54 00 31 00	36 00 2D 00	3 -	0 2 T 1 6 -	
C1 00 31 00	31 00 2D 00	35 00 32 00	2D 00 74 00	Á	1	1 - 5 2 - t
7A 00 2D 00	30 00 2D 00	66 00 69 00	6C 00 65 00	z -	0 - f i l e	
C1 00 31 00	2D 00 72 00	65 00 6E 00	61 00 6D 00	Á	1 -	r e n a m
65 00 64 00	2D 00 32 00	30 00 32 00	32 00 2D 00	e	d -	2 0 2 2 -
C1 00 30 00	34 00 2D 00	31 00 33 00	54 00 31 00	Á	0	4 - 1 3 T 1
37 00 2D 00	33 00 38 00	2E 00 74 00	78 00 74 00	7 -	3 8 .	t x t

Figure 5.2: Allocated directory set after rename.

in all timestamps. However, renaming a file will result in unallocated and allocated directory entry sets with the same timestamps. Testing this hypothesis will require more experiments, since our initial testing was only performed by renaming using Windows 10.

5.6 R4: FS tool reliability and validation

In most of this thesis research, testing how the digital forensic tools are interpreting the file system metadata has been included. This is necessary in order to see if the tools are reliable and if the tool can be validated by law enforcement.

- Publication A; Using the object ID index as an investigative approach for NTFS file systems (Nordvik et al. 2019b).
- Publication B; Generic Metadata Time Carving (Nordvik et al. 2020b).
- Publication C; Timestamp prefix carving for filesystem metadata extraction (Porter et al. 2021).
- Publication D: Its about time—Do exFAT implementations handle time-stamps correctly? (Nordvik and Axelsson 2022)

5.6.1 Publication A: Using the object ID index as an investigative approach for NTFS file systems

Even though NTFS is an old file system, we can still find artefacts that are not understood by all tools or investigators. The \$ObjID index is one of these artefacts. [Carrier \(2005, p. 335\)](#) described that the OS could assign a unique object identifier that will identify files even when a file has been changed or moved to another volume. [Carrier \(2005, p. 335\)](#) also describes that it is possible to refer to the \$ObjId index to find files that have an object id. When we wrote this paper in 2019, there were no tools to our knowledge that used this index to identify files. We suggested a new method using the \$ObjId index to detect files based on user activity and to connect the external storage media to the OS devices to which it has been attached.

Strengths of the implemented methods

Based on our experiments we observed how Object Identifiers (OIDs) were created or updated. We tested file creation, opening a file, copying or moving files to the same and other NTFS volumes, and file deletion. We exported the MFT table using the sleuthkit *icat* command, the \$ObjId Index Allocation attribute (named as \$O) and the Index Root attribute. Even though Sleuthkit did not show the Index Allocation attribute when using the *fts* command, we could easily export the index allocation attribute using the *icat* command with the file record number and the index allocation attribute number (160).

Based on our interpretation of the results, we created a prototype tool in C++ using the QT libraries that parses the \$ObjId index by using the output of Index Root Attribute or Index Allocation Attribute, and then we use the MFT record reference to identify the correct file record in the MFT table. The meaning of OIDs are based on previous research of link files and the meaning of OIDs ([Parsonage 2008](#)).

Our method considers indexes found in the \$ObjId Index Root Attribute if the number of entries are less than 7, or using the Index Allocation Attribute if it is more. Even though the tested tools are limited, we have detected that the File Time Extractor (*fte*) tool ([Yamazaki 2015](#)) that is using the \$ObjId does not support the Index Root Attribute and only supports parsing a live NTFS volume.

We also check if the Birth Volume Object ID has the volume moved bit set by assessing the least significant bit in its first byte when read as it is stored (not read as little endian). This can be shown in the hex dump of [Figure 5.3](#) where we have highlighted this byte with blue background. Its value is 0x3E, in binary 0011 1110, and here we can see that the move bit is not set, the highlighted bit is 0.

20 00 38 00 00 00 00 00	58 00 10 00 00 00 00 00	8	X	
86 13 5D 2E A9 DA EC 11	8B D3 18 CC 18 D6 E0 C4	† J. @Ui	< Ó I CãA	Object ID (the Key)
58 CF 12 00 00 00 01 00	BE 9B 78 06 C7 71 38 46	XI	x Cq8F	Birth Volume Object ID
8A 2C D4 A2 11 06 51 20	86 13 5D 2E A9 DA EC 11	Š, Öt Q † J. @Ui		Birth Object ID
8B D3 18 CC 18 D6 E0 C4	00 00 00 00 00 00 00 00	< Ó I CãA		Domain Object ID
00 00 00 00 00 00 00 00	20 00 38 00 00 00 00 00	8		
58 00 10 00 00 00 00 00	87 13 5D 2E A9 DA EC 11	X	† J. @Ui	
8B D3 18 CC 18 D6 E0 C4	14 D5 12 00 00 00 01 00	< Ó i CãA Ó		

Figure 5.3: ObjID index entry.

Our method can be used for:

- Reconstruction of user activity
- Connect which computer device an external hard drive has been attached by using the MAC address included in the Object ID.
- Creating timelines, based on Object ID boot session timestamp and its counter
- Detecting manipulation of timestamps by comparing SIA timestamps with Object ID boot session timestamp.

We compared the interpretation of OIDs in several popular DF tools, including X-Ways, EnCase, and Autopsy with our manual interpretation. X-Ways only showed the ObjectID key, Autopsy (Sleuthkit) failed to show information about OIDs. En-Case showed OIDs, parsed the Object ID timestamp, sequence number, and MAC address. Both EnCase and X-ways did not show the CrossVolumeMoveFlag and the Object ID order, which are important for event reconstruction. These findings show how important it is to validate tools based on the need of Law Enforcement.

Special weaknesses of the research contributions

The MAC address could be a random value if no network enabled device is present, and the MAC address is not always from the main *NIC*, it could be from other cards with a MAC address. We did not experiment with computers connected to a domain, and therefore the Domain Object ID only contained zeros (not used). Deleted files will get their entry removed from the \$ObjId index, and these files are not identified by our approach.

Our experiments only included Windows 7 and Windows 10, and we only tested LibreOffice, File Explorer, Notepad, Veracrypt and the command line (*CMD* prompt).

5.6.2 Publication B: Generic Metadata Time Carving

It is important for Law Enforcement techniques to have both high recall (getting metadata for all files) and high precision (only getting metadata for files) when

recovering file systems. Investigators need to avoid using resources investigating false positives, by only focusing on true positives. They also depend on that there are no false negatives.

Tool testing for validation

We tested commercial tools such as X-ways v 19.8 and EnCase v 8.08 to identify previous FILE records. The tools did not find any MFT records from the previous NTFS volume that had been reformatted with exFAT.

We also tested the tool EaseUS Data Recovery Wizard v 11.15, and this tool was able to recover all the 50 files and their content from this image.

We tested the same tools with the storage device that had a previous Ext4 volume that had been reformatted with NTFS. EnCase 8.08 was not able to recover the previous Ext4 partition, and only carving was an option. Carving for every supported file type was stopped after 6 hours of tool searching. We also tested the Bulk_Extractor tool, and it found all MFT entries, but no inodes from Ext4. We had 25000 *txt* files without signatures, so file carving was not feasible on this storage.

Based on our testing we can see that many popular commercial and open source tools fail in recovery of files from previous partitions that have been reformatted. Our GMTC method outperforms many of these tools when a volume with NTFS or Ext4 has been reformatted with another file system.

Strengths of the research methods

We have computed precision and recall of our results, and the approach identifies timestamps based on equality, which detects possible file metadata entries (inodes or MFT records) or other repeated data of a particular granularity. Then we use a file system semantic parser that increases the precision by evaluating expected metadata structures, which effectively removes any false positives.

Special weaknesses of the research contributions

This approach depends on timestamp entries for files being co-located, and have multiple timestamps where at least two are equal to the first. If all timestamps are unequal, the GMTC approach will not detect it, and we have a false negative.

5.6.3 Publication C: Timestamp prefix carving for filesystem metadata extraction

Tool testing for validation

We refer to the previous discussion of the GMTC approach when it comes to testing other tools.

Strengths of the research methods

This method fixes two of the weaknesses of the GMTC method. First it considers the p most significant bytes of each timestamp when checking equality. If p equals the granularity of the timestamp m , then this method is equal to the GMTC approach. Secondly, it allows the user to set the threshold h of how many matches must exist in order to consider it as a potential timestamps. The method is illustrated in Figure 5.4.

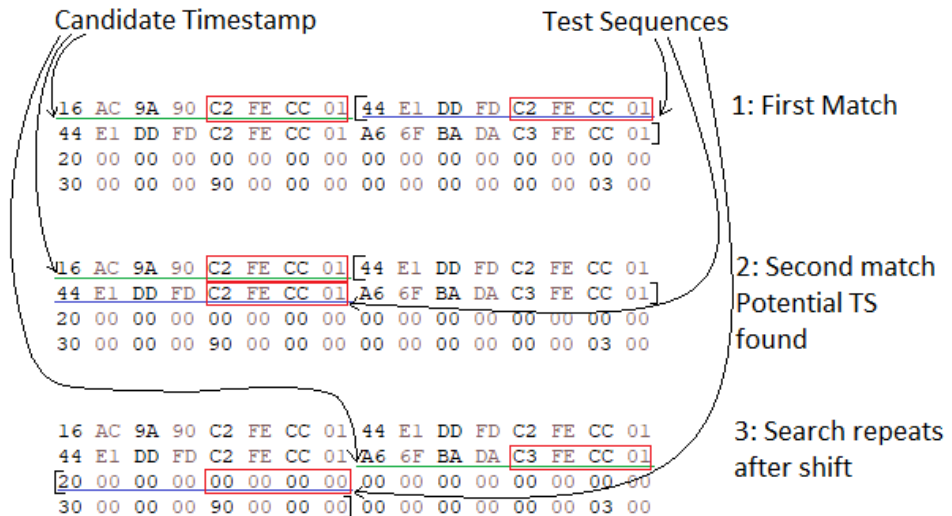


Figure 5.4: Hex dump with highlights to illustrate the timestamp prefix matching search procedure. The byte sequence underlined in green represents the current candidate timestamp, and those underlined with blue are test sequences. The brackets represent the candidate timestamp's search window ($k = 24$). The red boxes represent the little-endian prefixes ($p = 4$) that are being compared for equivalency. The first two examples show matches, despite the fact the candidate timestamp does not equal the subsequent ones. If three matching timestamps are required ($h = 3$), the third example shows the advancement of the search by k bytes, and begins to repeat the entire procedure.

Special weaknesses of the research contributions

The suggested approach of carving for metadata based on timestamp equality requires that the timestamps are co-located, and have multiple timestamps where at least $h \in 1 < h < (k/m + 1)$ (threshold) are equal. The timestamps are of m bytes. The prefix $p \in 0 < p \leq m$ is the number of most significant bytes that are compared for equality. The lower p is, the more true and false positives. The high number of potential timestamps, the more this increases the time the selected semantic parser will need to use in order to classify if the potential timestamps is part of a file metadata structure or not.

We still may experience that the mapping of directory entries and the inode numbers in the inode tables may fail due to fragmented inode tables. Even when we have flex groups, where each group is described in each group descriptor, we may experience that there are gaps between inode tables. They are not always contiguous following each other. We also depend on finding inodes that are directories, and to parse them and evaluate the dot entry to find out which inode number that directory inode has. The assumption that the data following an inode is the next inode will fail when the inode table is fragmented.

We could of course try to identify the group descriptors, but we want our approach to be generic and valid for multiple file systems. When reformatting the Ext4 file system with another file system group descriptors may be overwritten, and part of the inode tables may also have been overwritten. Our approach will work for Ext4 since it only depends on inodes.

5.6.4 Publication D: Its about time—Do exFAT implementations handle timestamps correctly?

When investigating criminal cases, accurate and reliable timestamps are critical. Misinterpretation of time may impact the decision of guilt or innocence by the court.

Reliability of FS drivers

Our research shows that using a removable storage with the exFAT file system on multiple operating systems may impact the reliability of timestamps. This is because the drivers used have been implemented differently, even when the exFAT specifications have been made publicly available. We have seen different behaviour between MacOS (uses local UTC offset, but switches sign), Windows (uses local UTC), Linux Ubuntu exFAT native driver (uses UTC+0), and the Linux Ubuntu exFAT fuse driver (uses an invalid 0x00 value).

Tool testing for validation

Digital forensic tools need to be tested and validated, also for interpreting file systems, since misinterpretation of metadata or not supported file system features may impact the accuracy of the file system metadata presented to the investigator. In our experiment we concluded that the tested version of Autopsy cannot be validated for interpreting the exFAT file system. Autopsy does not take the *UtcOffset* fields into consideration, but uses whatever local time selected when adding the forensic image to the case. Several of the commercial tools tested are having issues when interpreting invalid *UtcOffset* values. Whenever an invalid value is found, the tool should not try to convert it to another timezone. FTK Imager showed invalid timestamps as NA (Not Applicable), and converted the valid timestamps to UTC+0. X-Ways converted valid timestamps to the selected timezone, and showed LT (Local Time) if only invalid *UtcOffset* values for a File Directory Entry were found. However, it had issues if there were a mix of valid and invalid values in the same File Directory Entry. EnCase tried to convert valid and invalid *UtcOffset* values to the selected timezone, interpreting the invalid value 0x00 as UTC+0, which will fail in most cases. EnCase can only be validated for interpreting timestamps with valid *UtcOffsets*.

Strengths of our method

We have made the scripts available making it possible to repeat the experiments. We performed experiments on creation of files, opening files, and changing files. In addition we performed experiments with mounting and unmounting only. The datasets are also made available, allowing other researchers to verify our findings.

Special weaknesses of our contributions

We did not experiment on multiple versions of the same operating system, and results may deviate based on the version of the OS used. The graphical user interfaces used to open or change file content is just a few selected standard tools, and our research would have benefited by performing experiments using more tools.

5.7 Summary

This chapter has focused on how the results of the included papers relate to and answer the overall research questions. For each paper we discussed strengths and weaknesses of the selected method or results. We found that Object IDs can be used to identify user activity, create a timelines, connect files to the computers used to create them, and detect timestamp manipulation. Further, we found that the generic time carving approach (including the prefix carving) outperforms commercial tools, especially when testing using a reformatted Ext4 file system. The

exFAT findings revealed that different file system drivers implement timestamp metadata differently, and using a portable device on multiple OSes would make timestamps unreliable. Further, we observed that commercial tools had difficulties in interpreting UTC offsets that contained a mix of valid and invalid UTC values. In chapter 6 the thesis concludes and presents further research opportunities.

Chapter 6

Conclusion and further work

In order to conclude it is natural to include the research questions for this thesis. Therefore, the following sections try to answer each of the questions.

6.1 To what degree can user activity be documented from non-OS volumes (external storage devices) using only FS meta-data from the file system?

This research found that the *\$ObjId* system file in NTFS uses a special type of object identifier that can be used to not only identify which MAC address was used, but also identify under which boot session it was created, and when the boot session started. We observed that these Object IDs were created on typical user activity such as creation of files, opening files, or saving files. Our research also shows the order of the files created in a specific boot session, which result in the order the user created the Object IDs. Not all applications create a set of Object IDs on every user activity, but tools such as File Explorer always do on user activity open, and typical office applications generate Object IDs based on user activity. However, Veracrypt and the command line do not create new Object IDs.

Additionally, this research found that fork files stored on exFAT removable devices using MacOS may include the name of the graphical user interface tool used to change the files. These fork files indicate user activity.

6.2 To what degree can an external storage device be used to identify the computers it has been attached to by only assessing metadata from the file system?

Since the NTFS Object ID yields a MAC address from one of the computer network interfaces for each of the ids created by the user, this can be used to connect the device to a specific computer, at a specific boot session. However, it would not be possible to do this connection if the computer does not have a network interface card, or if the MAC address has been manually changed.

Even though exFAT metadata does not store metadata that uniquely identifies a computer, the OS automatically stores directories that identify either Windows or MacOS usage. In addition, it should be possible to connect the local timezone of the computer if the files are created on either Windows or MacOS, which is not possible if Linux were used.

6.3 To what extent can FS metadata reliably identify deleted files?

File system metadata will include detailed information of the blocks used by a file. Both for NTFS and Ext4 we found records or inodes that enable recovery of files. To ensure reliability the investigator needs to scrutinise the allocation bitmap, and visually assess the results. Our research shows that the metadata recovery method is more reliable than traditional file carving, especially the prefix-based GMTC method which has a higher recall and a high precision. We have also shown that we can use co-located timestamps to identify file system metadata for recovery purposes. The approach is general and should work on all file systems that use co-located timestamps, as long as file system validators are made available. The main use case for the approach is when a file system is damaged, since the current approach does not deviate between allocated or unallocated file records.

Identifying deleted files using exFAT is more difficult because renaming or moving a file will also leave unallocated entries behind, although no deletion has been performed. It is possible to test if the bitmaps used are zeroed, which will identify a deleted file. However, an overwritten file could be more difficult to separate from a renamed or moved file.

6.4 To what extent can the reliability and accuracy of file system parsing performed by current DF tools be assessed?

The DF tools tested partly show important relevant information about connected OIDs to files, where X-Ways v.19.8 only showed the Object ID key, missing the Birth Volume ID (including the CrossVolumeMoveFlag), Birth Object ID and the Domain ID. EnCase v.8.02 showed all OIDs except from the Domain ID, however the CrossVolumeMoveFlag in the Birth Volume ID was not interpreted. The CrossVolumeMoveFlag is important since it yields if the file has been moved from one NTFS volume to another or if it has been created on the volume under investigation. Both tools did not interpret the object id order, which is important for showing the order of when the OIDs were created within a boot session.

When it comes to recovery of files from a damaged file system (previous file system formatted with another file system), none of the tested tools were able to recover Ext4 text files. Recovering of previous files are important in digital forensics, and tool developers should take into consideration that users may reformat their file system with a different file system.

This research shows that file systems such as exFAT have different implementations on Windows, MacOS, and Linux, where one OS may wrongfully manipulate timestamps created on another implementation. The exFAT specifications are known, but when file system developers do not comply with specifications this may impact the DF tools parsing of file systems. Especially, digital forensic tools have difficulties interpreting timestamps where a mixture of valid and invalid UTC offsets are used.

This research shows that digital forensic tools fail to parse some file system metadata structures, not supporting a particular file system, or failing to recover data from supported file systems. We have also shown that dual tool verification is a method often relied on, but is based on an incorrect assumption that different developers do not make the same errors.

6.5 Further work

More file systems could be analysed to see if there are metadata that can connect an external storage device to a specific computer. Further research on other file systems and metadata recovery is needed.

Users may use a specific file system on multiple operating systems. Therefore, it will be necessary to perform file system research to compare and document deviations between different file system drivers, not assuming that they are equally implemented in each operating system. File systems are often ported to other

platforms for interoperability. These interoperability ports may or may not follow specifications, therefore, future experiments on multiple file system drivers for the same file system are needed.

It would be beneficial to include experiments using more popular applications to see if and when they update Object IDs. These experiments should include using Windows 11. In addition research should be performed using computers that are part of a domain, to see how and when the Domain Object ID is updated. Further, it would be interesting to correlate other system files to see if it is possible to identify exactly which operation was responsible for updating the Object IDs. Reverse engineering of APIs handling file operations could further yield the functions that update Object IDs, and these should be documented.

The thesis author did not include the paper about Reverse Engineering of ReFS in this thesis because the ReFS file system have been further developed. In Windows Server 2022 ReFS v. 3.7 is used, while in the latest developer release of Windows 11 ReFS v. 3.11 is used. ReFS v. 3.11 allows installing Windows 11 on a ReFS file system (Shee 2023). The most current version of ReFS should be included in further work, and if the Object ID feature is implemented it should be compared to the implementation in NTFS, especially since this feature indicates user activity and may be used to connect storage devices to computers.

We included semantic file system validators for NTFS and Ext4 for the generic metadata time carver. Adding support for additional file systems by developing more semantic validators would contribute to law enforcement. In addition, there is a need to improve the Ext4 connection between the inodes in the inode table and file names found in the directory entries without using the group block descriptor. Accurately identifying files that have been unallocated, would be beneficial for active file systems.

There is a need to reliably identify if an exFAT file has been deleted, deleted and overwritten, renamed or moved. These experiments should be performed on multiple operating systems, because we have already identified that different file system drivers have implemented the exFAT specification (Microsoft 2021) differently.

Performing research on mobile phone file systems is needed, assuming decryption of file systems will be available in the future. When only files are encrypted, the file system metadata will be available and the generic time carver may be used if the file system is Ext4. Still content recovery will depend on decryption techniques.

All prototype tools related to this thesis need to be further improved, especially for efficiency.

Even though existing tools have trouble interpreting the file system metadata from our research, we hope tool developers use the contribution from this research to increase the quality of their tools. After all, investigators depend on using tools.

Bibliography

- Ali, N. U. A., Iqbal, W., and Afzal, H. (2022). Carving of the ooxml document from volatile memory using unsupervised learning techniques. *Journal of Information Security and Applications*, 65:103096.
- An, S. H., Lee, S., and Han, J. (2023). Data reconstruction and recovery of de-duplicated files having non-resident attributes in ntfs volume. *Forensic Science International: Digital Investigation*, 46:301571.
- Apple (2018). Apple file system reference. <https://developer.apple.com/support/apple-file-system/Apple-File-System-Reference.pdf>, visited 2018-10-02.
- Arora, M. (2012). *Handling Endianness*, pages 155–168. Springer New York, New York, NY.
- Baloja, E. (2017). How to backup files form non-bootable windows pc. https://answers.microsoft.com/en-us/windows/forum/windows_other-performance/how-to-backup-files-from-non-bootable-windows-pc/c3d2b8bd-09ac-494f-a49a-c5d28b11deed.
- Bhat, W. A., AlZahrani, A., and Wani, M. A. (2020). Can computer forensic tools be trusted in digital investigations? *Science & Justice*.
- Carrier, B. (2002). *Open Source Digital Forensics Tools: The Legal Argument*.
- Carrier, B. (2005). *File System Forensic Analysis*. Addison-Wesley Professional.
- Carvey, H. (2018). Chapter 5 - setting up a testing environment. In Carvey, H., editor, *Investigating Windows Systems*, pages 97–115. Academic Press.

- Caviglione, L., Wendzel, S., and Mazurczyk, W. (2017). The future of digital forensics: Challenges and the road ahead. *IEEE Security & Privacy*, 15(6):12–17.
- Chen, J., Wang, J., hu Tan, Z., and Xie, C. (2014). Recursive updates in copy-on-write file systems - modeling and analysis. *J. Comput.*, 9:2342–2351.
- Dewald, A. and Seufert, S. (2017a). Afeic: Advanced forensic ext4 inode carving. *Digital Investigation*, 20:S83 – S91. DFRWS 2017 Europe.
- Dewald, A. and Seufert, S. (2017b). Afeic: Advanced forensic Ext4 inode carving. *Digital Investigation*, 20:S83 – S91. DFRWS 2017 Europe.
- DFRWS (2020). Generic metadata time carving - winner of the best paper award for usa 2020. https://dfrws.org/wp-content/uploads/2020/07/2020_USA_pres-generic-metadata-time-carving.pdf, visited 2020-07-24.
- Elrick, D. (2015). *Forensic Examination of Windows-Supported File Systems*. Lulu. com.
- Erbacher, R. F. (2010). Validation for digital forensics. In *2010 Seventh International Conference on Information Technology: New Generations*, pages 756–761.
- Ext4 (and Ext2/Ext3) Wiki (2019). Ext4 disk layout.
- Fairbanks, K. D. (2012). An analysis of ext4 for digital forensics. *Digital Investigation*, 9:S118 – S130. The Proceedings of the Twelfth Annual DFRWS Conference.
- Fandom (1993). Windows NT 3.1. https://microsoft.fandom.com/wiki/Windows_NT_3.1.
- Garfinkel, S. L. (2007). Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2 – 12.
- Garfinkel, S. L. (2010). Digital forensics research: The next 10 years. *Digital Investigation*, 7:S64 – S73. The Proceedings of the Tenth Annual DFRWS Conference.
- Garfinkel, S. L. and McCarrin, M. (2015). Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation*, 14:S95–S105. The Proceedings of the Fifteenth Annual DFRWS Conference.

- Gladyshev, P. and James, J. I. (2017). Decision-theoretic file carving. *Digital Investigation*, 22:46 – 61.
- Gnome Developer (2014). Gnome developer file utilities. <https://developer-old.gnome.org/glib/stable/glib-File-Utilities.html#g-file-set-contents>.
- Grenier, C. (2019). Photorec. Last visited: 2024-02-19.
- Hansen, K. H. and Toolan, F. (2017). Decoding the apfs file system. *Digital Investigation*, 22:107 – 132.
- Hilgert, J.-N., Lambertz, M., Rybalka, M., and Schell, R. (2019). Syntactical carving of pngs and automated generation of reproducible datasets. *Digital Investigation*, 29:S22 – S30.
- Horsman, G. (2018a). Framework for reliable experimental design (fred): A research framework to ensure the dependable interpretation of digital data for digital forensics. *Computers & Security*, 73:294 – 306.
- Horsman, G. (2018b). “i couldn’t find it your honour, it mustn’t be there!” – tool errors, tool limitations and user error in digital forensics. *Science & Justice*, 58(6):433–440.
- Horsman, G. (2019). Tool testing and reliability issues in the field of digital forensics. *Digital Investigation*, 28:163–175.
- Horsman, G. (2020). Acpo principles for digital evidence: Time for an update? *Forensic Science International: Reports*, 2:100076.
- IEEE 802 (2014). Ieee standard for local and metropolitan area networks: Overview and architecture. *IEEE Std 802-2014 (Revision to IEEE Std 802-2001)*, pages 1–74.
- ISO/IEC (2015). ISO/IEC 27041:2015 guidance on assuring suitability and adequacy of incident investigative method. <https://www.iso.org/standard/44405.html>.
- Karresand, M., Axelsson, S., and Dyrkolbotn, G. O. (2019a). Using ntfs cluster allocation behavior to find the location of user data. *Digital Investigation*, 29:S51–S60.
- Karresand, M., Axelsson, S., and Dyrkolbotn, G. O. (2020a). Disk cluster allocation behavior in windows and ntfs. *Mobile Networks and Applications*, 25(1):248–258.

- Karresand, M., Dyrkolbotn, G. O., and Axelsson, S. (2020b). An empirical study of the ntfs cluster allocation behavior over time. *Forensic Science International: Digital Investigation*, 33:301008.
- Karresand, M., Warnqvist, A., Lindahl, D., Axelsson, S., and Dyrkolbotn, G. O. (2019b). Creating a map of user data in ntfs to improve file carving. In Peterson, G. and Sheno, S., editors, *Advances in Digital Forensics XV*, pages 133–158, Cham. Springer International Publishing.
- Karresand, N. M. M. (2023). *Digital Forensic Usage of the Inherent Structures in NTFS*. PhD thesis, Norway, Gjøvik.
- Knight, J. C. and Leveson, N. G. (1986). An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109.
- Leachi, P., Mealing, M., and Salz, R. (2005). A Universally Unique Identifier (UUID) URN Namespace. Available at <https://www.ietf.org/rfc/rfc4122.txt>. Last accessed on 2018-08-30.
- Lillis, D. and Scanlon, M. (2016). On the benefits of information retrieval and information extraction techniques applied to digital forensics. In Park, J. J. H., Jin, H., Jeong, Y.-S., and Khan, M. K., editors, *Advanced Multimedia and Ubiquitous Engineering*, pages 641–647, Singapore. Springer Singapore.
- Lin, X. (2018). *Data Hiding and Detection*, pages 271–301. Springer International Publishing, Cham.
- Lyle, J. R. (2010). If error rate is such a simple concept, why don't i have one for my forensic tool yet? *Digital Investigation*, 7:S135–S139. The Proceedings of the Tenth Annual DFRWS Conference.
- Mahajan, R., Singh, M., and Miglani, S. (2014). Ads: Protecting ntfs from hacking. In *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, pages 1–4.
- Marshall, A. M. and Paige, R. (2018). Requirements in digital forensics method definition: Observations from a uk study. *Digital Investigation*, 27:23 – 29.
- Microsoft (2021). exFAT file system specification. <https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification>.
- Neale, C., Kennedy, I., Price, B., Yu, Y., and Nuseibeh, B. (2022). The case for zero trust digital forensics. *Forensic Science International: Digital Investigation*, 40:301352.

- NIST (2023a). Cftt federated testing project. <https://www.nist.gov/itl/ssd/software-quality-group/computer-forensics-tool-testing-program-cftt/cftt-federated-testing>, visited 2023-06-01.
- NIST (2023b). Test Results for Mobile Device Acquisition Tool - Magnet Axiom v6.8.0.33717. https://www.dhs.gov/sites/default/files/2023-04/23_0411_st_test_results_for_mobile_device_acquisition_tool-magnet_axiom_v6.8.0.33717.pdf, visited 2023-06-01.
- Nordvik, R. and Axelsson, S. (2022). It is about time—do exfat implementations handle timestamps correctly? *Forensic Science International: Digital Investigation*, 42-43:301476.
- Nordvik, R. and Axelsson, S. (2023). Corrigendum to “it is about time—do exfat implementations handle timestamps correctly?” [forensic science international: Digital investigation 42–43 (2022) 301476]. *Forensic Science International: Digital Investigation*, 45:301542.
- Nordvik, R., Georges, H., Toolan, F., and Axelsson, S. (2019a). Reverse engineering of ReFS. *Digital Investigation*, 30:127–147.
- Nordvik, R., Porter, K., Toolan, F., Axelsson, S., and Franke, K. (2020a). Generic metadata time carving. *Digital Investigation*, 33:301005. The Proceedings of the Twentieth Annual DFRWS USA.
- Nordvik, R., Porter, K., Toolan, F., Axelsson, S., and Franke, K. (2020b). Generic metadata time carving. *Forensic Science International: Digital Investigation*, 33:301005.
- Nordvik, R., Stoykova, R., Franke, K., Axelsson, S., and Toolan, F. (2021). Reliability validation for file system interpretation. *Forensic Science International: Digital Investigation*, 37:301174.
- Nordvik, R., Toolan, F., and Axelsson, S. (2019b). Using the object id index as an investigative approach for ntfs file systems. *Digital Investigation*, 28:S30–S39.
- Okun, M. and Barak, A. (2004). Atomic writes for data integrity and consistency in shared storage devices for clusters. *Future Generation Computer Systems*, 20(4):539–547. Advanced services for Clusters and Internet computing.
- Parsonage, H. (2008). The meaning of linkfiles in forensic examinations. Last accessed on 2017-06-01.

- Porter, K., Nordvik, R., Toolan, F., and Axelsson, S. (2021). Timestamp prefix carving for filesystem metadata extraction. *Forensic Science International: Digital Investigation*, 38:301266.
- Richard III, G. G. and Roussev, V. (2005). Scalpel: A frugal, high performance file carver. In *Refereed Proceedings of the 5th Annual Digital Forensic Research Workshop, DFRWS 2005, Astor Crowne Plaza, New Orleans, Louisiana, USA, August 17-19, 2005*.
- Richard Maw (2016). Atomic file creation with temporary files. <https://yakking.branchable.com/posts/atomic-file-creation-tmpfile/>.
- Scanlon, M. (2016). Battling the digital forensic backlog through data deduplication. In *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*, pages 10–14.
- Shee, J. (2023). How to install windows 11 on refs partition, try it now. Available at <https://iboysoft.com/howto/install-windows-11-on-refs-partition.html>. Last accessed on 2023-08-02.
- Stoykova, R. and Franke, K. (2023). Reliability validation enabling framework (rvef) for digital forensics in criminal investigations. *Forensic Science International: Digital Investigation*, 45:301554.
- Sunde, N. and Horsman, G. (2021). Part 2: The phase-oriented advice and review structure (pars) for digital forensic investigations. *Forensic Science International: Digital Investigation*, 36:301074.
- The European Parliament and of The Council (2018). General data protection regulation. <https://gdpr-info.eu/>.
- The United Kingdom Forensic Science Regulator (2020). Method Validation in Digital Forensics, FSR-G-218, issue 2. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/921392/218_Method_Validation_in_Digital_Forensics_Issue_2_New_Base_Final.pdf.
- United States Air Force Office of Special Investigations and The Center for Information Systems Security Studies and Research (2007). Photorec. Last visited: 2024-02-19.
- United States Supreme Court (1993-1999). *Daubert v. merrell dow pharmaceuticals, inc.*, 509 u.s. 579, 1993. the daubert criteria was further elaborated in general

- electric co. v. joiner 522 u.s. 136 (1997), and kumho tire co. v. carmichael 526 u.s. 137 (1999).
- Vandermeer, Y., Le-Khac, N., Carthy, J., and Kechadi, M. T. (2018). Forensic analysis of the exfat artefacts. *CoRR*, abs/1804.08653.
- Wani, M. A., AlZahrani, A., and Bhat, W. A. (2020). File system anti-forensics – types, techniques and tools. *Computer Fraud & Security*, 2020(3):14–19.
- Willassen, S. Y. (2009). A model based approach to timestamp evidence interpretation. *International Journal of Digital Crime and Forensics (IJDCF)*, 1:1–12. Issue 2.
- Williams, J. (2012). Acpo good practice guide for digital evidence. *Metropolitan Police Service, Association of chief police officers, GB*, pages 1556–6013.
- Wu, J. (2011). Improving the writing of research papers: Imrad and beyond. *Landscape Ecology*, 26(10):1345–1349.
- Yamazaki, T. (2015). Filetime extractor. Available at <http://www.kazamiya.net/en/fte>. Last accessed on 2019-01-08.

Appendix A

Publication A: Using the object ID index as an investigative approach for NTFS file systems

Using the object ID index as an investigative approach for NTFS file systems, Rune Nordvik, Fergus Toolan and Stefan Axelsson. In: *Digital Investigation Volume 28, Supplement*. April 2019, Pages S30-S39. DFRWS 2019 Europe — Proceedings of the 19th Annual DFRWS Conference. DOI: <https://doi.org/10.1016/j.diin.2019.01.013>

Abstract

When investigating an incident it is important to document user activity, and to document which storage device was connected to which computer. We present a new approach to documenting user activity in computer systems using the NTFS file system by using the \$ObjId Index to document user activity, and to correlate this index with the corresponding records in the MFT table. This may be the only possible approach when investigating external NTFS storage devices, and is hence a valuable addition to the storage forensics toolbox.

A.1 Introduction

Users interact with the file system by navigating, creating, moving, renaming, copying or deleting files, or directories. Digital forensic investigators normally use digital forensic tools to investigate criminal cases (Garfinkel 2010, Gül and Kugu 2017). When digital forensic tools parse the NTFS file system they often show only selected parts of each MFT record. In order to validate the results of the tools it would be necessary for digital forensic investigators to use hex viewers, or tools such as mftcrd (Schicht 2018) to manually interpret the MFT records. In NTFS, metadata about files is mainly found in the system file \$MFT (master file table) (Carrier 2005, p. 353), but metadata might exist in other system files including \$ObjId, \$LogFile, \$UsnJrnl, \$Secure, etc. Typically, file metadata could include timestamps, file names, block allocations (data runs or extents), Object IDs, different indexes, etc (Carrier 2005, chap. 13).

This paper will focus on Object Identifiers (OIDs). The Object ID index found in the \$ObjId system file can help the investigator to find all allocated files that have an Object ID, which will assist in event reconstruction of user activity. OIDs are created based on typical user activity and are used by Windows in order to track an object (file, directory or link) even if the object changes location or name (Microsoft 2016). OIDs will be created when a file is opened by the user in Windows File Explorer, or when the file is opened or saved by some applications (Parsonage 2008). A user can also use the command line tool `fsutil objectid` to create, delete or set OIDs. If a user moves a file to another volume the Object ID might change, however, the Birth Object ID and the Birth Volume Object ID should be preserved (Microsoft 2016). A volume is a collection of addressable sectors that can be used for storage, and a volume can also be a partition (Carrier 2005, p. 70). In the context of this paper, the volume is a partition using the NTFS file system. According to Microsoft the Windows OS uses OIDs in order to track files (Microsoft 2016).

A digital forensic tool might show OIDs connected to a file, but different digital forensic tools deviate in how OIDs connected to a file are presented. We performed

an experiment to determine if forensic tools display OIDs. Thus we tried X-Ways Forensics and Autopsy on a file which was known to have connected OIDs. The results were that X-Ways Forensics showed only the Object ID key, and Autopsy (Sleuthkit) failed to show any information relating to OIDs. EnCase shows Object IDs and parses the Object ID timestamp, sequence number and the MAC address ([Habben 2018](#)). If a file, directory or link is assigned OIDs, the following will be assigned:

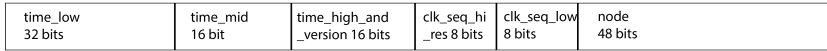
- Object ID (used as a key in the index)
- Birth Volume Object ID (special identifier equal to the Object ID of the \$Volume system file from the volume the OIDs were created)
- Birth Object ID (equal to the first Object ID assigned and should not change)
- Domain Object ID (always zeros, reserved)

It is not enough to just display an artifact, the investigators need to understand what it means. The authors consider OIDs to be important for digital forensics for the following reasons:

- OIDs will show which boot session a file with OIDs belongs too ([Leachi et al. 2005](#)), which can assist in timeline creation.
- OIDs can show the node (MAC-address) used by the computer that created the OIDs ([Leachi et al. 2005](#)). This means we will be able to determine to which computers the external storage medium has been attached, as long as the user has accessed files and created new OIDs.
- OIDs can show in which sequence files have been assigned OIDs within a boot session. This might assist in detecting manipulation of timestamps and in building timelines.
- The \$ObjId index can be used as a triage tool in order to identify files or directories the user has accessed.
- The Birth Volume Object ID might be used to identify the file system volume used when the file was first assigned OIDs.

The Object ID is a unique 16 byte identifier used to identify files on a NTFS volume ([Microsoft 2016](#)). Any file that obtains an Object ID, will also have a

UUID, rfc 4122



UUID version 1, rfc 4122

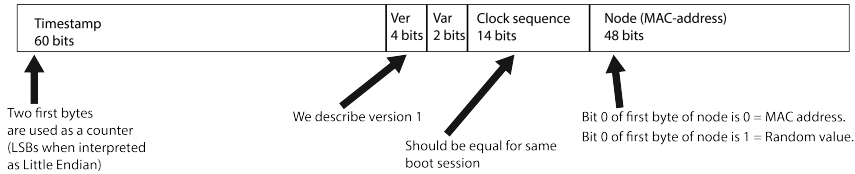


Figure A.1: Structure of an Object ID UUID version 1.

Birth Volume Object ID, a Birth Object ID and a Domain ID (Microsoft 2016). The Birth Volume Object ID (16 bytes) is used for identifying the volume the file was located on when it first obtained an OID (Microsoft 2016). The Birth Object ID (16 bytes) is the first Object ID assigned to the file. The Object ID may change if the file is moved, but the Birth Object ID should remain constant (Microsoft 2016). The Domain ID is a 16 byte structure reserved for identifying a domain, and must be 16 bytes of zeros (Microsoft 2016). Our experiments attempt to observe and assess if the description of **fsutil** by Microsoft is still true in Windows 10.

OIDs are 16 bytes in size and contain a 60 bit timestamp, which is the number of 100 nanosecond intervals since 15.10.1582 (Leachi et al. 2005, Parsonage 2008). This timestamp is found in the first 60 bits of the OID and is related to the start of the boot session in which the OID was created (Leachi et al. 2005). The two least significant bytes of this timestamp, when interpreted as Little Endian, are also used as a counter showing the order of OID creation within the specific boot session (Parsonage 2008). The counter is the only two bytes that separates Object IDs assigned in the same boot session. The timestamp can be converted to FILETIME by subtracting the hex value 0x146BF33E42C000, allowing tools that interpret FILETIME to convert it. The OIDs have a clock sequence which will be identical for all OIDs created in a particular boot session. Finally the last 6 bytes of the OID will normally include the MAC address of the default Network adapter. If no NIC is available this will contain a random number (Leachi et al. 2005). A graphical illustration is shown in figure A.1.

The Object ID is used as an index key in the \$ObjId\$ file and this Object ID is also located in the Object ID Attribute (type 0x40) in the corresponding MFT record. We can also find the MFT record number in the \$ObjId\$ index entry (Car-

rier 2005, p. 335). This way it is easy to find the correct record in the index, knowing the Object ID key from the MFT record, but also to find all Master File Table (MFT) records that have an Object ID by examining the \$ObjId\$O index entries. It is the latter approach that is presented in this paper. A prototype tool has been developed which implements this approach and was used during the course of these experiments.

The remainder of this paper is organized as follows. Section A.2 describes related work. Section A.3 illustrates the research goals. Section A.4 describes the methodology and details about our experimental setup. Section C.4 presents the results of our experiment. Section A.6 presents the evaluation methodology and results of assessing the feasibility and reliability of the approach. Section A.7 discusses and interprets the results. Finally, Section A.8 concludes and provides recommendations for future work.

A.2 Related work and contributions

Previous work on Object IDs has focused on interpreting the meaning of OIDs found in link files (shortcut files), or OIDs from link files found in the NTFS journal. In this section we describe this previous related work and finally we describe our contributions.

A.2.1 Related work

Carrier provided a description of the \$OBJECT_ID structure (Carrier 2005, p. 367) and the index \$ObjId structure (Carrier 2005, pp. 386-387). Carrier describes OIDs as an alternate method of addressing files, which allows for locating the file even if the name and location have changed (Carrier 2005, p. 335). Carrier does not describe the format the OIDs are using or their exact meaning.

In Windows, users can create shortcut files that point to other files. The Windows OS often creates these shortcut files automatically based on user activity. These shortcut files normally have the extension **lnk** and are called link files (Parsonage 2008). Parsonage describes which OIDs can be found within link files and compares them to the output of the **fsutil** command. Within link files the following OIDs might be stored:

- New VolumeID (corresponds to the Volume Object ID of the \$Volume system file, but not found in the \$ObjId index if this is from another NTFS volume)
- New File ObjectID (should be identical with the Object ID found in the \$ObjId index)

- Birth VolumeID (should be identical with the Birth Volume Object ID found in the \$ObjId index entry, but the move bit is not set)
- Birth File ObjectID (should be identical with the Birth Object ID found in the \$ObjId index entry)

[Parsonage \(2008\)](#) describes the importance of Link Files, and mentions that there exists an index of Object IDs, however, little use is made of this. This research is based on the description of OIDs from this article, but does not focus on the binary content of link files. Parsonage claims that the OIDs are not preserved on removable media. This paper attempts to determine the veracity of this claim. We will use the \$ObjId index as an approach to find all allocated files on a volume which would indicate user activity. The OID structure, described in [Leachi et al. \(2005\)](#), [Parsonage \(2008\)](#), can be used to connect the device to one or multiple computer system(s) using the MAC address included in the OID.

In Windows, jump lists are used for saving recently used items for an application or for the OS itself. For instance, the list of recently opened documents is made possible using a jump list. [Singh and Singh \(2016\)](#) describe jump lists, and show how to interpret these for Windows 10, which is different from Windows 7 and 8. Their work shows that OIDs are used in DestList and LNK streams, which includes embedded shortcut files. Within these shortcut files / streams both the new Volume Object ID and the Birth Volume Object ID might be shown, which is helpful for tracking purposes. However, an investigator may have no access to the system volume, meaning they would have no access to the jump lists. In these cases the investigator only has the \$ObjId index, the MFT table or other system files to investigate. In this index we find the Object ID, the Birth Volume Object ID, the Birth Object ID, the Domain ID (unused) and the reference to the MFT record ([Carrier 2005](#), pp. 386-387).

[McGrath and Gladyshev \(2013\)](#) describe how to use the NTFS \$logfile to find cleartext files after encryption. The authors use **fsutil** to determine the Birth Volume Object ID from the known ciphertext file. They state that the ciphertext file and the fact that the Birth Volume Object ID was found inside the \$logfile and conclude that the encryption took place on that volume. They also do the same for the cleartext file found in the \$logfile. Even if the file was deleted, the previous dataruns, used by the cleartext file, might still be present within \$logfile. In our experience not all encryption software creates Object IDs or link files, and not all cleartext files have OIDs. Some encryption software will create OIDs both for the cleartext file selected and the ciphertext file created.

[Cowen \(2018a\)](#) has performed a few experiments regarding Object IDs. The results

of his experiments are not published in a peer-reviewed paper. He is using a python script parsing every MFT record for the OBJECT_ID attribute. He suggests that the last 6 bytes of the Object ID is the MAC address, even for the \$Volume. His testing also shows that he does not find valid MAC addresses for the Object ID connected to the \$Volume. Further, his script source code shows he has based his parsing on the knowledge from Parsonage (Cowen 2018b). Cowen's testing shows that there are Object IDs even for some of the files installed on the system, and that their MAC addresses have been preserved. He concludes that there is less Object IDs for pre-installed files on Windows 10 compared to Windows 7. Cowen does not use the system file \$ObjId in his experiments, and therefore he only finds the main Object ID key.

Yamazaki (2015) has published a closed source tool, **fte**, that should be able to parse the \$MFT, \$ObjId system file and other NTFS indexes. When we tested this tool on Windows 10, it was only able to parse the \$ObjId system file when selecting a live volume, and only if the Index Allocation Attribute existed. The tool shows correctly the date from the Object ID, but the column describes ctime which easily could be interpreted incorrectly as change time. The tool detected correctly if a file has been moved from another NTFS volume. The fte tool does not parse the Index Root Attribute, when there are just a few files with OIDs on a volume.

A.2.2 Contributions

None of the above related work address the meaning of the Object IDs saved in the \$ObjId index. As can be seen, no-one has previously identified what kind of operations update the \$ObjId index. Hence, our contributions include novel investigation methods for:

- Event reconstruction of user activity using the Object ID index correlated with the \$MFT table.
- Documentation of computer devices to which an external hard drive has been attached.
- Finding the boot times of a computer by investigating the Object ID index of attached NTFS volumes, which could be correlated with external NTFS storage devices that have been attached.
- Creating timelines.
- Detection of manipulation of timestamps by analyzing Object IDs.

A.3 Research goals

This research focuses on the feasibility and reliability of using \$ObjId\$O index to document user activity.

- **Feasibility:** The selected approach should be feasible for use with new versions of Windows, and therefore we have selected Windows 7 and 10 as our test systems.
- **Reliability:** The selected approach should reliably detect user activity.

A.3.1 Research questions

This paper aims to determine if user activity can be documented from non-OS NTFS volumes using FS metadata from the NTFS file system. It also aims to determine if it is possible to discover what machine(s) a device was connected to by using artifacts present on the device.

When the NTFS system volume is unavailable, investigators can no longer rely upon jump lists, recent link files, registry, event logs and prefetch files in order to determine user activity. Only the artifacts found in the NTFS file system can be relied upon, hence, it is the opinion of the authors that this approach may be the only means of recreating user activity for external NTFS media.

A.3.2 Automation

Forensic tools have basic support for parsing the MFT record attributes, but to the authors' knowledge only two tools, mftcrd ([Schicht 2018](#)) and fte ([Yamazaki 2015](#)), show all the timestamps from all the File Name attributes (FNAs) within the MFT record. Furthermore, only the fte tool is able to parse the \$ObjId index to list all OIDs in the system under investigation. As part of this work an open source tool has been developed that automates the parsing of \$ObjId and correlation with the pertinent attributes in the MFT record. The prototype tool, **NTFSObjIDParser** ([Nordvik 2019](#)), was developed in C++ using the graphical QT Libraries. The target users are computer forensic investigators. Users need to export the \$MFT table and the \$ObjId\$O Index Allocation Attribute (type 0xA0) or the Index Root Attribute (type 0x90), as shown in [Figure A.2](#) and [Figure A.4](#). Using these inputs, the prototype tool will correlate each index entry with the corresponding MFT record.

In NTFS indexes are used for storing \$MFT attributes in a sorted order, and a B-tree is used ([Carrier 2005](#), p. 290). The root node is always located in the resident Index Root Attribute ([Carrier 2005](#), p. 294). If all the nodes can not fit resident

(7 or more entries) in the Index Root attribute, a non-resident Index Allocation Attribute is used (Carrier 2005, p. 294). The \$MFT record of the \$ObjId system file contains these two attributes (Index Root and/or Index Allocation), and the indexed attribute is in this case the \$MFT \$OBJECT_ID Attribute(type 0x40).

A.4 Methodology

The `NTFSObjIDParser` prototype tool was used in our experiments. The output was verified using the `xxd` hex viewer and the forensic suite `Sleuthkit` (Carrier 2017).

A.4.1 Object ID creation

The purpose of this experiment is to determine when an Object ID is created. Multiple tools were evaluated. These included: the command prompt; File Explorer; Notepad; VeraCrypt; and LibreOffice. The scenarios tested on the NTFS filesystem were:

- **File creation:** Using a tool to create a new file.
- **Opening a file:** Using a tool to open an existing file, with or without an Object ID, and test if rebooting impacts the result.
- **Copying a file (same volume):** Using a tool to copy a file, with or without an Object ID, to the same volume, and test if rebooting impacts the result.
- **Copying a file (other volume):** Using a tool to copy a file, with or without an Object ID, to another NTFS volume, and test if rebooting impacts the result.
- **Moving a file (same volume):** Using a tool to move a file, with or without an Object ID, to another directory on the same volume, and test if rebooting impacts the result.
- **Moving a file (other volume):** Using a tool to move a file, with or without an Object ID, to a directory on another volume, and test if rebooting impacts the result.
- **Deleting a file:** Using a tool to delete a file.

The reboot means that after the test, the machine is rebooted, and the test repeated. The reboot was performed to see if the 60 bit timestamps within the OIDs were updated to the last timestamp for the most recent boot time. This was tested for all scenarios where files have an existing OID.

After each test, and after the reboots, the MFT table and the clusters found in the Index Allocation Attribute data runs were exported. Simple Sleuthkit commands (Carrier 2017) were used for extraction of the MFT, but during the experiments we thought it was necessary to use **dd** to gather the clusters from the Index Allocation Attribute. Sleuthkit v. 4.4.1 to v. 4.6.2 did not show the Index Allocation Attribute, only showing the Index Root Attribute. However, it is possible to extract an existing Index Allocation Attribute using Sleuthkit by combining the MFT record number and the attribute type. The USB device was unmounted from Windows, and mounted in MacOS where Sleuthkit was installed.

```
# List partition table
sudo mmls /dev/rdisk2
# Export the record 0 (MFT Table)
# from volume starting on sector 32
sudo icat -o 32 /dev/rdisk2 0 > MFT.bin
# Showing the 25th MFT record
dd if=MFT.bin bs=512 skip=$(( 25*2 )) count=2 | xxd
```

Figure A.2: Exporting the MFT table, and MFT record number 25.

The **mmls** command in Figure A.2 was used to show the partition tables, and to find the correct volume. Using this information the MFT table was exported. From the extracted MFT table the \$ObjId MFT record (25) was shown in the hex viewer. It should be noted that the 25th MFT record is not always used for the \$ObjId file.

```
00000150: 0000 0000 0000 0000 a000 0000 5000 0000 ..... P...
00000160: 0102 4000 0000 0300 0000 0000 0000 0000 ..@.....
00000170: 0000 0000 0000 0000 4800 0000 0000 0000 .....H.....
00000180: 0010 0000 0000 0000 0010 0000 0000 0000 .....
00000190: 0010 0000 0000 0000 2400 4f00 0000 0000 .....$.O....
000001a0: 1101 23 00 00c0 ffff b000 0000 2800 0000 ..#.....(...
```

Figure A.3: Hex dump of the Index Allocation Attribute.

Figure A.3 shows the Index Allocation Attribute which commences at offset 0x158 (type 0xA0). Skipping 0x48 bytes, and examining the value at offset 0x1A0, the bytes 0x110123 are seen. This provides the data run for the attribute in question. Interpreting this shows that the contents start at cluster 0x23 and occupy a single cluster. The test disk has 8 sectors per cluster, therefore the data content of the \$ObjId\$O index is located at sector 280 relative to the start of the volume. Allowing for the 32 sectors before the volume, sectors 312 - 319 are extracted as shown in the first command in Figure A.4.

The last command skips the index file header (64 bytes) within this file and shows an object ID index entry. The result of this is shown in Figure A.5. We observed

```

# Export the Index Allocation non resident content
sudo dd if=/dev/rdisk2 bs=512 skip=312 count=8 of=ObjectID.bin
# Alternative method: It is possible to extract the
# Index Allocation Attribute using Sleuthkit, however
# it's corresponding MFT-fileid-attribute is not shown by
# fls when using Sleuthkit. If $ObjId is inode 25:
sudo icat -o 32 /dev/rdisk2 25-160 > ObjectID.bin
# If no Index Allocation Attribute exist, extract the
# Index Root Attribute
sudo icat -o 32 /dev/rdisk2 25-144 > ObjectID-IR.bin

# Show one of the index entries
sudo dd if=ObjectID.bin bs=1 skip=64 count=88 | xxd

```

Figure A.4: Exporting the Object ID Index Allocation non resident data, and show one Object ID Index Entry.

that when it was less than 7 entries, there was no Index Allocation Attribute, and all the indexes were, in this case, stored resident in the Index Root Attribute. In this case it is necessary to extract the Index Allocation Attribute, and to skip the file header (32 bytes) in order to find the first Object ID index entry.

```

00000000: 2000 3800 0000 0000 5800 1000 0000 0000  .8.....X.....
00000010: 2535 8c37 c7f3 e611 9c55 0800 2737 afb0  %5.7.....U..'7..
00000020: 2500 0000 0000 0100 0000 0000 0000 0000  %.....
00000030: 0000 0000 0000 0000 2535 8c37 c7f3 e611  .....%5.7....
00000040: 9c55 0800 2737 afb0 0000 0000 0000 0000  .U..'7.....
00000050: 0000 0000 0000 0000  .....

```

Figure A.5: Hex dump of an Object Index Entry.

The structure of an Object ID index entry is shown in Figure A.6. The basic parsing of this index structure is defined by Carrier (2005, p. 387). The 8 bytes at offset 0x20 provide a reference to the MFT table. The 16 most significant bits (in this case all multibyte data fields are stored in Little Endian format) are for the sequence number and the remainder is for the MFT record number (0x25). There are a total of 4 universally unique identifiers (UUIDs), but the Domain UUID always has a zero value. The Object ID UUID will also be found in the MFT record Object ID Attribute, but none of the other UUIDs will be present. Both the Object ID and the Birth Object ID will have a 60 bit timestamp, as described in Section A.1. The two least significant bytes represent the Object ID order, in other words the order in which OIDs were created. In bytes 8 and 9 of the UUID the clock sequence number is found. Remember to set the two variant bits to 0. Then we read the two bytes as an array of bytes. This sequence number is equal for all UUIDs that were created / updated within the same boot session. The last 6 bytes,

```

typedef struct _INDEX_ENTRY
{
    quint16 OffsetData; // 0x00
    quint16 SizeData; // 0x02
    quint8 Padding1[4]; // 0x04 - Unused
    quint16 SizeIndexEntry; // 0x08
    quint16 SizeIndexKey; // 0x0A Size of the Object ID
    quint32 Flags; // 0x0C - DOS flags
    quint8 ObjectID[16]; // 0x10 - Used as an index key
    quint64 MFTRecord; // 0x20
    quint8 BirthVolumeID[16]; // 0x28 Does not follow the standard for
    ↳ OIDs, as described. Can be correlated to the $Volume Object
    ↳ Attribute. Windows 10 set it to zero for external storage
    ↳ devices!
    quint8 BirthObjectID[16]; // 0x38, Should remain the same
    quint8 DomainID[16]; // 0x48 Not used, set to zero values
} INDEX_ENTRY; // Total of 88 bytes or 0x58 bytes

```

Figure A.6: C structure of an Object ID index entry.

when read as an array of bytes rather than a multibyte field, will show the MAC address of the standard NIC used. If no NIC was used a random number appears at this location (Parsonage 2008). More details on how to parse an Object ID entry are shown in Table A.1 and in Figure A.6.

Offset	Length	Meaning
0x00	0x02	Offset to data
0x02	0x02	Size of data
0x04	0x04	Padding (Unused)
0x08	0x02	Size of Index Entry
0x0A	0x02	Size of Index Key (Object ID)
0x0C	0x04	Flags
0x10	0x10	Object ID UUID (the key)
0x20	0x08	Reference to MFT record
0x28	0x10	Birth Volume Object ID UUID
0x38	0x10	Birth Object ID UUID
0x48	0x10	Domain ID UUID

Table A.1: Offset table index entry, based on Carrier (2005, pp. 386-387).

Manually parsing each Object ID entry is too time consuming when every index entry must be parsed, and therefore the prototype tool, *NTFSObjIDParser*, is used for automation. The prototype correlates Object ID entries with the MFT record found in the entry reference by parsing the MFT record's Standard Information Attribute (SIA), all File Name Attributes (FNAs) and the Object Identifier Attribute (OIA). The first column in Figure A.7(a) shows the MFT record reference for each row. This shows which rows represent the same item. Then the byte offset to the entry or to the MFT record is shown, and the relative offset from each entry where the MFT attribute or Object ID type can be found. Knowing the offsets will allow

verification of results by computer forensic investigators.

Next the MFT Header flags are shown if the entry is an MFT record or the Object ID entry flags are shown if the entry is an OID. The MFT header flags will show if the file or directory is allocated or unallocated. It is unlikely that unallocated files or directories will be found, this is due to the fact that deleted files will be removed from the index, however it will be present in the MFT table as long as the record has not been reused. An attempt was made to find patterns describing what actions created the OID: creation; opening; copying; moving; deleting. This is not fully implemented in the prototype. In the Name column we show the OIDs or File Names. The SIA does not have a File Name or an OID, so it is left empty.

For Object ID, Birth Object ID and the MFT OIA attribute the Created timestamps are shown. It should be noted that the time is the system boot time before creating the OIDs. For SIA or FNA the Created, Modified, Record Modified and Accessed timestamps are shown, as can be seen from Figure A.7(b). Note that these timestamps are approximately real time, but that the Accessed timestamp does not get updated all the time. Then the MAC address computed from the last 6 bytes of the OIDs is shown. In the field Object ID Order the decimal value of the two least significant bytes of the 60 bit timestamp in the OIDs is shown. This is not shown for the Birth Volume Object ID, since this OID does not have a timestamp. The last column shows the clock sequence, which shows which OID entries were created within a boot session.

We used Virtual Box v. 5.1 to virtualize Windows 7 Home Premium SP1 (32bit) and Windows 10 Pro (64bit). The attached SATA USB3 disks were of the type Lacie Porsche Mobile (1 TiB), each using one volume and formatted as NTFS. For Windows 7 we needed to install USB3 drivers. Since we were using virtual machines, the USB disks were automatically released to the host (MacOS High Sierra v 10.13) OS when rebooted. This was also why we observed that the \$Volume Object ID was not always set. In the final stages of preparing this paper, we found that we could add the USB device to the USB device filters in the Settings, Ports, USB in Virtual Box. This way we could restart the virtual machine while the USB disk was attached during the reboot. The internal NTFS volumes were created by adding a vmdk disk device using Virtual Box, and then formatting it in Windows. We also tested using different USB thumb drives, however rebooting with the USB thumb drives attached did not assign a \$Volume Object ID.

A.5 Results

We observed that for the Index Allocation Attribute (type 0xA0) to be used as an attribute in the \$ObjId MFT record, it is necessary to have more than 6 entries

	MFT Record	Byte Offset	Attribute or Type	MFT Header Flags	Volume Action	Name
8	41	Object\DSO offset = 152 + 16	ObjectID	0		4b87f0a4fef4e6119c5608002737afb0
9	41	Object\DSO offset = 152 + 40	BirthVolumeID	0		00000000000000000000000000000000
10	41	Object\DSO offset = 152 + 56	BirthObjectID	0		4b87f0a4fef4e6119c5608002737afb0
11	41	Object\DSO offset = 152 + 72	DomainObjectID	0		00000000000000000000000000000000
12	41	MFT offset = 41984+56	SIA	Allocated File		
13	41	MFT offset = 41984+152	FNA	Allocated File		\\testin2.odt
14	41	MFT offset = 41984+264	OIA	Allocated File		4b87f0a4fef4e6119c5608002737afb0

(a)

	Created	Modified	Record Modified	Accessed	MAC Address	ObjectID Orde
8	Fri Feb 17 10:48:40 2017				8-0-27-37-af-b0	34635
9						
10	Fri Feb 17 10:48:40 2017				8-0-27-37-af-b0	34635
11						
12	Sun Feb 19 21:44:47 2017	Sun Feb 19 21:44:47 2017	Sun Feb 19 21:44:47 2017	Sun Feb 19 21:44:47 2017		
13	Sun Feb 19 21:44:47 2017	Sun Feb 19 21:44:47 2017	Sun Feb 19 21:44:47 2017	Sun Feb 19 21:44:47 2017		
14	Fri Feb 17 10:48:40 2017				8-0-27-37-af-b0	34635

(b)

Figure A.7: NTFSObjIDParser output. Results are split between (a) and (b).

on a newly NTFS formatted volume. This might also depend on the number and size of the attributes within the \$ObjId MFT record. If we have less entries, the indexes will be found in the Index Root Attribute (type 0x90). In the experiment a NTFS formatted USB disk was used. Since MFT record 3 (\$Volume) did not get an Object ID attribute, the value used for Birth Volume Object ID UUID was zero. This result was unexpected, as all documentation consulted described that the Birth Volume Object ID should be assigned a unique value identifying the volume (Microsoft 2016, McGrath and Gladyshev 2013, Singh and Singh 2016, Parsonage 2008). The missing Birth Volume Object ID UUID was also observed when using Windows 7. In these cases the \$Volume Object ID attribute was also not present in the MFT table. This was observed on recently created volumes on internal disks, and on removable disks. Whenever the \$Volume Object ID attribute was available in the MFT table, then a non-zero Birth Volume Object ID UUID was present in the \$ObjId index. We were only successful in creating an Object ID for the \$Volume system file if we performed formatting of an internal or external disk using Windows 7 or 10. We also observed that a reboot might be necessary after the formatting in order for the \$Volume Object ID to be assigned, and that the disk must be attached during the boot process. When this internal or external disk was quick reformatted again, the Object ID for the \$Volume system file was normally preserved.

In the following tables (A.2 - A.8) the tests that were performed are summarized. The following abbreviations have been used: W7 (Windows 7); W10 (Windows 10); OID (Object ID); BOID (Birth Object ID); and BVOID (Birth Volume Object ID). The **OS** column contains the operating system used. **Impact** contains the different OIDs that the action might impact. **Existing OID** has the value Yes if the file had existing OIDs before the operation was performed, **Preserved OID** contains Yes if previous OIDs from the source file were preserved after the operation. **New OID** has the value Yes if the action created a new Object ID. **Tool** describes the tool used for the operation.

A.5.1 File creation

Table A.2 shows that creating a file makes an entry in the \$ObjId\$O index if File Explorer is used on Windows 10, but not when using Windows 7. If LibreOffice is used to create a new file a new entry is created in the \$ObjId\$O index on both versions of Windows. If the command prompt is used and the output is redirected to a file, no entry is made in the Object ID index. When using Notepad to create a file, no entry is made in the Object ID index. If File Explorer is used to extract a zip container (including a directory and a file), this creates the directory with an Object ID entry in Windows 10, but not in Windows 7. However, the file that was extracted did not get any entry in the Object ID index. A 100 MiB container, created using VeraCrypt, did not result in any OIDs.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10	OID, BOID, BVOID	No	No	-	Yes	LibreOffice
W7	OID, BOID, BVOID	No	No	-	No	File Explorer (File or Directory)
W10	OID, BOID, BVOID	No	No	-	Yes	File Explorer (File or Directory)
W7	OID, BOID, BVOID	No	No	-	No	Extract directory from zip (File Explorer)
W10	OID, BOID, BVOID	No	No	-	Yes	Extract directory from zip (File Explorer)
W7, W10	OID, BOID, BVOID	No	No	-	No	Extract file from zip (File Explorer)
W7, W10	OID, BOID, BVOID	No	No	-	No	CMD prompt, Notepad
W7, W10	OID, BOID, BVOID	No	No	-	No	VeraCrypt

Table A.2: Experiment 1 - Test 1: File Creation.

A.5.2 Opening a file

Table A.3 shows that if a file with no Object ID entry was opened by double clicking on it in File Explorer, then it received an Object ID entry. Identical UUIDs for Object ID and Birth Object ID were created. When double clicking a file with existing OIDs in File Explorer, the OIDs were preserved. This was also the case if we rebooted the system first. If LibreOffice was used to open a file without OIDs, a new entry was added to the Object ID Index. Using Libreoffice to open a file with existing OIDs preserved the OIDs. The same behavior was observed when rebooting the system first. No deviations were observed between Windows 7 and

10 when opening files. Both Notepad and the Command Prompt failed to create an OID after opening a file. They did, however, preserve existing OIDs.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10	OID, BOID, BVOID	No	No	-	Yes	File Explorer (double click)
W7, W10	OID, BOID, BVOID	No	Yes	Yes	No	File Explorer (double click)
W7, W10	OID, BOID, BVOID	Yes	Yes	Yes	No	File Explorer (double click)
W7, W10	OID, BOID, BVOID	No	No	-	Yes	LibreOffice (File Open)
W7, W10	OID, BOID, BVOID	No	Yes	Yes	No	LibreOffice (File Open)
W7, W10	OID, BOID, BVOID	Yes	Yes	Yes	No	LibreOffice (File Open)
W7, W10	OID, BOID, BVOID	No	No	-	No	CMD prompt, Notepad (File Open)
W7, W10	OID, BOID, BVOID	No	Yes	Yes	No	CMD prompt, Notepad (File Open)
W7, W10	OID, BOID, BVOID	Yes	Yes	Yes	No	CMD prompt, Notepad (File Open)

Table A.3: Experiment 1 - Test 2: Opening a file.

A.5.3 Copying a file (same volume)

In Table A.4 File Explorer is used to drag and drop a file while holding CTRL (this ensures the file is copied) (Microsoft 2001). The original file did not have any Object ID before the operation. Both the original and the copy did not get any entry in the Object ID index after this operation. If the source file had OIDs before, these are preserved for the source file, but no OIDs were found for the new copy. Using LibreOffice Save As created new OIDs for the copy. If the **copy** terminal command was used to copy a file to the same volume, it did not create new OIDs for the copy. Notepad was used to create a copy using Save As. In Windows 7 OIDs were created for the copy, but not in Windows 10. Copying a file also is creation of a file based on an existing file. A new entry will be created in the MFT table, and therefore this is also a part of the copy operation.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10	OID, BOID, BVOID	No	No	-	No	File Explorer (CTRL drag)
W7, W10	OID, BOID, BVOID	No	Yes	No	No	File Explorer (CTRL drag)
W7, W10	OID, BOID, BVOID	Yes	Yes	No	No	File Explorer (CTRL drag)
W7, W10	OID, BOID, BVOID	No	Yes	No	Yes	LibreOffice (Save As)
W7, W10	OID, BOID, BVOID	Yes	Yes	No	Yes	LibreOffice (Save As)
W7, W10	OID, BOID, BVOID	No	Yes	No	No	CMD prompt (copy)
W7, W10	OID, BOID, BVOID	Yes	Yes	No	No	CMD prompt (copy)
W7	OID, BOID, BVOID	No	Yes	No	Yes	Notepad (Save As)
W7	OID, BOID, BVOID	Yes	Yes	No	Yes	Notepad (Save As)
W10	OID, BOID, BVOID	No	Yes	No	No	Notepad (Save As)
W10	OID, BOID, BVOID	Yes	Yes	No	No	Notepad (Save As)

Table A.4: Experiment 1 - Test 3: Copying file to the same volume.

A.5.4 Copying a file (other volume)

Table A.5 shows the results of copying a file to another volume using File Explorer's drag and drop functionality while holding the CTRL key. The results, regarding OIDs, were the same as when the file is copied to the same volume. We also show the result when using LibreOffice's Save As feature, which created new OIDs for the target file. When using Notepad's Save As feature only Windows 7 created new OIDs for the target file. Using the command prompt `copy` command did not create OIDs for the target file.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10	OID, BOID, BVOID	No	No	-	No	File Explorer (CTRL drag)
W7, W10	OID, BOID, BVOID	No	Yes	No	No	File Explorer (CTRL drag)
W7, W10	OID, BOID, BVOID	Yes	Yes	No	No	File Explorer (CTRL drag)
W7, W10	OID, BOID, BVOID	No	Yes	No	Yes	LibreOffice (Save As)
W7, W10	OID, BOID, BVOID	Yes	Yes	No	Yes	LibreOffice (Save As)
W7, W10	OID, BOID, BVOID	No	Yes	No	No	CMD prompt (copy)
W7, W10	OID, BOID, BVOID	Yes	Yes	No	No	CMD prompt (copy)
W7	OID, BOID, BVOID	No	Yes	No	Yes	Notepad (Save As)
W7	OID, BOID, BVOID	Yes	Yes	No	Yes	Notepad (Save As)
W10	OID, BOID, BVOID	No	Yes	No	No	Notepad (Save As)
W10	OID, BOID, BVOID	Yes	Yes	No	No	Notepad (Save As)

Table A.5: Experiment 1 - Test 4: Copying file to another volume.

A.5.5 Moving a file (same volume)

In Table A.6 File Explorer's drag and drop functionality is used while holding the SHIFT key (to ensure the file was moved) (Microsoft 2001). The file did not get an entry in the Object ID index after this operation. If the file had existing OIDs, then these were preserved. The same was observed when using the `move` command from the CMD prompt.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10	OID, BOID, BVOID	No	No	-	No	File Explorer (SHIFT drag)
W7, W10	OID, BOID, BVOID	No	Yes	Yes	No	File Explorer (SHIFT drag)
W7, W10	OID, BOID, BVOID	Yes	Yes	Yes	No	File Explorer (SHIFT drag)
W7, W10	OID, BOID, BVOID	No	No	-	No	CMD prompt (move)
W7, W10	OID, BOID, BVOID	No	Yes	Yes	No	CMD prompt (move)
W7, W10	OID, BOID, BVOID	Yes	Yes	Yes	No	CMD prompt (move)

Table A.6: Experiment 1 - Test 5: Moving file to the same volume.

A.5.6 Moving a file (other volume)

The behaviour when moving a file from one NTFS volume to another depends on the OS used, and if the volume is an internal volume or an external volume. All

our observations show that external disks have an Object ID equal to zero for the \$Volume system file when these external disks have not been connected during reboot. This was observed for both Windows 7 and 10, regardless of the format method. However, internal disks when formatted normally (not quick) will have an Object ID for the \$Volume file. If the same internal or external drive is reformatted, then the Object ID for the \$Volume system file is preserved.

Table A.7 shows File Explorer's drag and drop, while holding the SHIFT key, being used to move a file to a different volume. The file did get an entry in the Object ID index after this operation. The least significant bit, when reading the timestamp location as Little Endian, was set in the Birth Volume Object ID (the move bit). The Object ID and the Birth Object ID were preserved in this new index entry. We also observed an exception if the volume was an NTFS volume without an Object ID Attribute in the \$Volume system file (external disk). In this case we observed that the moved file got a new Object ID and Birth Object ID in Windows 10, but no Object IDs were created in Windows 7. For Windows 10 the Birth Volume Object ID was also set to 0.

For internal disks with an Object ID in the \$Volume system file using the command prompt and the **move** command will preserve the Object ID and the Birth Object ID. The Birth Volume Object ID is also preserved, but the least significant bit is set to 1. If this bit is already set, then the Birth Volume Object ID is preserved. If the Object ID of the \$Volume of the target volume was zero (external disk), then the OIDs were not preserved and no new OIDs were created even if the **move** command was used.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10	OID, BOID, BVOID	No	No	-	No	File Explorer (SHIFT drag)
W7, W10	OID, BOID, BVOID	No	Yes	OID, BOID	BVOID LSB=1	File Explorer (SHIFT drag)
W7	OID, BOID, BVOID	No	Yes	No	No	File Explorer (SHIFT drag). Target BVOID = 0
W10	OID, BOID, BVOID	No	Yes	No	Yes + (BVOID=0)	File Explorer (SHIFT drag). Target BVOID = 0
W7, W10	OID, BOID, BVOID	Yes	Yes	OID, BOID	BVOID LSB=1	File Explorer (SHIFT drag)
W7, W10	OID, BOID, BVOID	No	No	-	No	CMD prompt (move)
W7, W10	OID, BOID, BVOID	No	Yes	OID, BOID	BVOID LSB=1	CMD prompt (move)
W7, W10	OID, BOID, BVOID	Yes	Yes	OID, BOID	BVOID LSB=1	CMD prompt (move)
W7, 10	OID, BOID, BVOID	No	Yes	No	No	CMD Prompt (move). Target BVOID = 0
W7, 10	OID, BOID, BVOID	Yes	Yes	No	No	CMD Prompt (move). Target BVOID = 0

Table A.7: Experiment 1 - Test 6: Moving file to another volume.

A.5.7 Deleting a file

If a file is deleted that has an entry in the \$ObjId index, then the B-tree index will re-organize, and the result is often that the previous entry will be overwritten. The same was observed when using the **del** command in the CMD prompt. This is also shown in Table A.8.

OS	Impact	Reboot	Existing OID	Preserved OID	New OID	Tool
W7, W10		No	No	-	No	File Explorer (SHIFT delete)
W7, W10		No	Yes	No	No	File Explorer (SHIFT delete)
W7, W10		Yes	Yes	No	No	File Explorer (SHIFT delete)
W7, W10		No	No	-	No	CMD prompt (del)
W7, W10		No	Yes	No	No	CMD prompt (del)

Table A.8: Experiment 1 - Test 7: Deleting a file.

A.6 Evaluation

To evaluate the results, we focus on the two research goals described in section [A.3](#), Feasibility and Reliability.

A.6.1 Feasibility

Is it feasible to use the file \$ObjId to document User Activity? Only the operations that actually create OIDs will be detected. Creating a new file (W10) or opening a file from File Explorer (W7 and W10), LibreOffice or other applications using the same API will be detected. We will not detect all user activity on the NTFS File System by only scrutinizing the Object ID index and the MFT records. However, it is feasible to assume that files with an entry in the Object ID index are there because of user activity. In many real cases, when the investigator only has access to a removable disk, this approach might be the only method of documenting user activity. It can also be used to map possible hosts to which the removable device has been attached.

A.6.2 Reliability

To answer the question of the reliability of this approach it is necessary to focus on what it does not detect. When a file is deleted, the B-tree \$ObjId index is re-organized, and the previous content in the object index is normally overwritten. However, the Object ID in the MFT record can still be found, as long as the MFT record is not reused. This is an indication that the file has been opened, created or saved by the user or a software tool. Using the command line shell will normally go undetected, except when moving a file to another NTFS volume that has an Object ID assigned to the \$Volume system file. However, if the \$Volume Object ID is zero, moving a file to this volume will go undetected. When copying a file from one NTFS volume to another, the target file will not get OIDs. Creating an encrypted container will not generate OIDs when VeraCrypt is used.

It seems that all applications that use the Windows API FileOpen or FileSave dialogs will create OIDs. We tested this by creating a very simple tool that used the

IFileOpenDialog interface, and OIDs were created when we used it to open a file that did not have OIDs. Normal users tend to use graphical user interfaces when using Windows, and therefore it is possible to detect a significant portion of user activity by utilizing the OIDs.

A.7 Discussion

Since OIDs are created based on typical user activities on NTFS volumes, using the \$ObjId index will be a very efficient way to detect which files were accessed by the user. Not all Object IDs will have a LNK (shortcut) file in its Recent folder or as a LNK stream in a Jumplist. The user can even create their own LNK files, which could be stored in a selected directory. We do not claim that the Object ID index will find all user activity, but users using the File Explorer or other Graphical User Interface (GUI) tools have little control over index entry creation. Windows tool developers often use the Windows API instead of creating their own FileOpen or FileSave dialogs, meaning that Object ID creation will be enabled regardless of the programmer's awareness.

As observations show, normal user activity will create entries in the \$ObjId index file. The \$ObjId file is not directly accessible by the normal user, as it is a system file. This makes it more difficult to hide the traces. It is easy to hide traces by deleting LNK files or eventlog entries or by using a tool to clear UserAssist and RecentDocs in the Registry. It is possible to delete entries in the \$ObjId index by using the **fsutil** tool, or by deleting files. The latter will still preserve the Object ID attribute in the MFT record, as long as the MFT record is not reused. This is because only a flag in the corresponding MFT record header is changed when deleting a file (Carrier 2005). It is currently possible to change the \$ObjId index file from user space by using fsutil in Windows 10 (not in Windows 7) or by utilizing the correct API when developing new anti-forensic tools. It is not possible to set new OIDs using fsutil if there exists a set of OIDs for the particular file. In order to set new OIDs it is necessary to delete the existing OIDs first, then create new ones. In NTFS there are other system files that will be updated when using fsutil to change the OIDs, for instance the \$UsnJrnl have entries that describe the type of change (Carrier 2005, p. 394). Manipulation of OIDs can easily be detected if they do not follow the same format as Windows. If the MFT record SIA created timestamp is manipulated to a future date within another Object ID session, analyzing the previously assigned Object ID will normally detect this manipulation. This because the Object ID identify the boot session it belongs to, and therefore the MFT SIA created date should not be in the time range of a later Object ID boot session.

An interesting question is if all OIDs are only created based on User Activity?

The answer depends on how we define user activity. In this study any process that behaves on behalf of a user, as a user agent or a chain of user agents, is user activity (Buchholz and Spafford 2004). For instance a process is normally executed by a user or the OS. Even though the user started the OS, we do not count automatic OS activity not initiated by the user as user activity. A malicious program is started somehow by a user, not necessarily the local user, and we consider this user activity.

The Object ID index can be used to find all allocated files that have an Object ID. The Object ID keys found in the \$ObjId file can also be compared with the unallocated entries in the MFT table which contain an OID. This will indicate that the user did more than just delete the file, and the file should therefore be recovered for further investigation.

Even if users wipe their system drive, the computer used can be discovered by analyzing a previously attached removable NTFS volume. This is because the MAC address is usually contained within the OIDs. If OIDs are created during multiple sessions on different computers, the removable NTFS volume can also yield different boot times for the computers to which it has been attached.

We can not depend on the move flag (least significant bit in the timestamp when read as LE (Parsonage 2008)) of the Birth Volume Object ID when the target Birth Volume Object ID is 0. In this case other Object ID and Birth Object ID are created, which makes it look as if the file was not moved. In these cases the file can only be connected to a computer using the MAC address found in the Object ID and the Birth Object ID. When a user moves a file from one volume to another, the move flag will only be set if the target Birth Volume Object ID is not zero.

A.8 Conclusions and future work

Users will use File Explorer or other software tools to create, open, copy, move and delete files. In the cases in which OIDs are created, it will yield user activity. Even if the system volume is not available, we know that the OIDs are artifacts from some form of user activity. On external drives the \$ObjId is one of the very few artifacts found that can yield user activity.

Our experiments using Windows 7 and 10 show that a Birth Volume Object ID is not always created, even if Birth Object ID and Object ID are created. Previous research has documented that Birth Volume Object IDs are created or updated (Parsonage 2008), but our results show Birth Volume Object IDs with only zeros. This means that we are even more dependent on the MAC address found within the Birth Object ID to connect the computer used to create the OIDs. If an external disk with a NTFS volume is attached while rebooting, our experiments show that

\$Volume system file is assigned a new Object ID if the existing one is not set. However, we have observed exceptions to this when using USB thumb drives.

Analyzing the \$ObjId index is important in order to:

- create timelines
- connect NTFS volumes to one or more computers by using the MAC address found within the Object ID
- select which files to analyze (data reduction or triage)
- detect boot sessions and the order of OIDs creation
- detect MFT created date manipulation

For further work we suggest to determine if correlation with other system files can be used to validate the interpretation of the \$ObjId system file. In this context, \$UsnJrnl system file (Carrier 2005, p. 343) and the \$logfile (Carrier 2005, p. 340) is known to be useful for event reconstruction. However, the \$logfile is normally very small (64 MiB) and the transactions will start overwriting the oldest transactions when necessary (Zareen and Aslam 2014). This means the NTFS \$logfile journal transactions are very volatile and will only document file activity for a particular time range, with that range dependent on the degree of volume activity. It would also be interesting to expand this study by correlating the \$ObjId index with other system files in order to see if it is possible to reliably detect what kind of operation created the Object IDs.

More research could be performed on which APIs implement the use of \$ObjId system file. We have documented that the IFileOpenDialog API will create OIDs. Even if our work shows similarities between Windows 7 and 10, it also shows differences. This was expected, since programmers change their software tools regularly, and they decide which APIs they want to use in each release. The APIs themselves could also change in the future.

More experiments should be performed to determine what a change of the \$Volume Object ID can have on existing OIDs in the \$ObjId index. Further experiments should be performed in order to see if adding a partitioning scheme on USB thumb drives will impact the creation of \$Volume Object ID.

Bibliography

- Buchholz, F. and Spafford, E. (2004). On the role of file system metadata in digital forensics. *Digital Investigation*, 1(4):298 – 309.
- Carrier, B. (2005). *File System Forensic Analysis*. Addison-Wesley Professional.
- Carrier, B. (2017). The Sleuth Kit® (TSK) is a library and collection of command line tools that allow you to investigate disk images. Available at <https://www.sleuthkit.org/sleuthkit/>. Last accessed on 2018-08-30.
- Cowen, D. (2018a). Forensic lunch test kitchen 9/13/18. Available at <https://www.hecfblog.com/2018/09/daily-blog-491-test-kitchen-92718.html>. Last accessed on 2019-01-01.
- Cowen, D. (2018b). Objectidscannerv2. Available at <https://github.com/dlcowen/TestKitchen/blob/master/OBJECTIDScannerV2.py>. Last accessed on 2019-01-01.
- Garfinkel, S. L. (2010). Digital forensics research: The next 10 years. *Digital Investigation*, 7:S64 – S73. The Proceedings of the Tenth Annual DFRWS Conference.
- Gül, M. and Kugu, E. (2017). A survey on anti-forensics techniques. In *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*, pages 1–6.
- Habben, J. (2018). Ntfs object ids in encase. Available at <https://4n6ir.com/2018/09/20/ntfs-object-ids-in-encase/>. Last accessed on 2019-01-18.

- Leachi, P., Mealing, M., and Salz, R. (2005). A Universally Unique Identifier (UUID) URN Namespace. Available at <https://www.ietf.org/rfc/rfc4122.txt>. Last accessed on 2018-08-30.
- McGrath, N. and Gladyshev, P. (2013). *Investigating File Encrypted Material Using NTFS \$logfile*, pages 183–203. Springer Berlin Heidelberg, Berlin, Heidelberg.
- Microsoft (2001). Will dragging a file result in a move or a copy? Available at <https://blogs.msdn.microsoft.com/oldnewthing/20041112-00/?p=37323>. Last accessed on 2018-08-30.
- Microsoft (2016). Fsutil objectid. Available at [https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/cc788098\(v=ws.11\)](https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/cc788098(v=ws.11)). Last accessed on 2018-08-30.
- Nordvik, R. (2019). Ntfs object id parser. Available at <https://github.com/RuneN007/NTFSObjectIDParser>. Last accessed on 2019-01-07.
- Parsonage, H. (2008). The meaning of linkfiles in forensic examinations. Last accessed on 2017-06-01.
- Schicht, J. (2018). Command line \$mft record decoder. <https://github.com/jschicht/MftRcrd>. Last accessed on 2018-09-27.
- Singh, B. and Singh, U. (2016). A forensic insight into Windows 10 Jump Lists. *Elsevier - Digital Investigation*, 17:1–13.
- Yamazaki, T. (2015). Filetime extractor. Available at <http://www.kazamiya.net/en/fte>. Last accessed on 2019-01-08.
- Zareen, M. S. and Aslam, B. (2014). \$logfile of ntfs: A blueprint of activities. In *17th IEEE International Multi Topic Conference 2014*, pages 305–310.

Appendix B

Publication B: Generic Metadata Time Carving

Generic Metadata Time Carving, Rune Nordvik, Kyle Porter, Fergus Toolan and Stefan Axelsson and Katrin Franke. In: *Forensic Science International: Digital Investigation Volume 33, Supplement*. July 2020, Pages 301005. DFRWS 2020 USA — Proceedings of the Twentieth Annual DFRWS USA. DOI: <https://doi.org/10.1016/j.fsidi.2020.301005>

Abstract

Recovery of files can be a challenging task in file system investigation, and most carving techniques are based on file signatures or semantics within the file. However, these carving techniques often only recover the files, but not the metadata associated with the file. In this paper, we propose a novel, generic approach for carving metadata by searching for equal and co-located timestamps. The rationale is that there are some common metadata for files and directories within each file system. Our generic time carver provides potential timestamp locations for repeated timestamps in each metadata structure, identifying potential metadata for files. A semantic parser then filters the results with respect to the specific file system type. In our experiments, extraction of MFT entries in NTFS and inodes in Ext4 had near perfect precision for metadata entries with multiple equivalent timestamps, and for such metadata structures we obtained perfect recall for NTFS. For known file systems, we use the information found within identified metadata to recover files, and by recovering files and their associated metadata we increase the evidential value of recovered files.

B.1 Introduction

File carving is a technique which identifies and extracts files from unallocated areas based on signatures found within the file content, and not by using file system metadata (Garfinkel 2007). While extremely useful, file carving has a few challenges. First, investigators need to decide which file type to carve. To decrease the file carving time, investigators often select the file types they assume could be pertinent for the criminal case. For instance, by carving for typical image files in cases related to sexual abuse of children, the investigator limits the ability to identify other file types. Furthermore, not all files have a signature, and will not be found by using file carving. Some carving techniques will carve based on the assumption that files have contiguous blocks, which will fail when trying to carve a fragmented file (Garfinkel 2007).

Our novel approach does not use file carving, but rather metadata carving. We search for repeated co-located timestamps, based on equality, in a small window to obtain locations of potential timestamps. In this way, timestamps are used as a kind of dynamic signature. Once verifying the timestamp as likely to be legitimate for a particular file system, we use the metadata surrounding it to fully or partially recover the file. Our approach handles both contiguous and fragmented files.

We only recover file or directory metadata from NTFS and Ext4 to demonstrate the usability of the novel approach, but the approach can be extended to recover metadata from other file systems. In order to achieve a realistic scenario, we dam-

age the original file system by reformatting the volume with another file system. The tools developed in this paper are prototypes, and the main target group are file system experts with the competence to manually assess file system structures. The tools and the disk images can be downloaded for review at [Nordvik et al. \(2020\)](#).

To our knowledge, no one has used equality between closely co-located timestamps to identify metadata before as a carving technique. Previous attempts suffer from many shortcomings including the inability to find static signatures for all pertinent metadata structures.

Our approach focuses on metadata structures found in MFT entries or inodes carved from unallocated space. There will always be a risk that the blocks (clusters) pointed to by the discovered metadata structures may be overwritten by new or existing allocated files, but this may be identified by examining the allocated file system bitmap for allocated blocks, and by comparing metadata information with the content of the recovered file. Most file systems have some sort of bitmap system which has bits representing each block (cluster), and allocated blocks have their corresponding bit set ([Carrier 2005](#), p. 311).

Our new approach is suited for recovery of metadata and file content from storage devices that have been reformatted with another file system, with the same kind of file system when not assessing the allocated inode/file table, or from generally damaged file systems. The approach is also useful for finding historical metadata structures located on disk that are not contained in MFT or inode tables.

Detailed file system structures are described by [Carrier \(2005\)](#). Even though his book does not include details about Ext4, it contains most of the basic information from Ext2 and Ext3. [Dewald and Seufert \(2017\)](#) include more details about Ext4.

B.1.1 Assumptions

Currently, most file systems will include at least 3 contiguous timestamps. Linux file systems normally use the MAC (Modified, Accessed and Changed) timestamps ([Carrier 2005](#), p. 297), for instance Ext2 and Ext3 use the contiguous `atime` (accessed), `ctime` (inode changed), `mtime` (data modified) and `dtime` (deleted) ([Carrier 2005](#), p. 298). Ext4 also contains the same contiguous timestamps, but adds the `ctime` (creation) in the end of the inode ([Ext4 development team 2019](#)). NTFS and ReFS use 4 contiguous timestamps (Creation, Modified, MFT modified, and Accessed) in multiple attributes ([Carrier 2005](#), [Nordvik et al. 2019](#)). Listing B.1 shows a few file systems with closely co-located timestamps.

File System	Co-located timestamps	Granularity
NTFS (Carrier 2005)	4	64 bit - ns intervals since 1.1.1601
ReFS (Nordvik et al. 2019)	4	64 bit - ns intervals since 1.1.1601
APFS (Hansen and Toolan 2017)	4 (5)	64 bits - ns since 1970
HFS+ (Apple 2004)	4	32 bits - s since 1904
BTRFS (Bhat and Wani 2018)	3 (4)	64 bits - s since 1970 + 32 bits (ns)
ExFAT (Hamm 2009)	3	32 bits + UTC offset
FAT (Carrier 2005)	3	16 bits for time (except accessed), 16 bits for day
UFS1 (Carrier 2005)	3	32 bits - s since 1970 + 32 bits (ns)
UFS2 (Carrier 2005)	4	64 bits (ns) since 1970
Ext2/3 (Carrier 2005)	4	32 bits - s since 1970
Ext4 (Dewald and Seufert 2017)	4	34 bits - s since 1970 + 30 bits (ns) (Göbel and Baier 2018)

Table B.1: File Systems with timestamps co-located within metadata structures

B.1.2 Objectives

- Can we reliably use time as a generic identifier to carve for file and directory metadata structures in different file systems?
- What is the reliability of recovery of files using the discovered metadata in Ext 4 and NTFS?

We aim to identify file or directory metadata structures from different file systems based on a common identifier. In our case, equal and closely co-located timestamps, which will allow for a generic approach for metadata carving. We identify the potential timestamps by using a simple string matching algorithm, and then we interpret the semantics¹ of the expected file system metadata in order to significantly reduce the number of false positives.

Identifying the metadata is not enough in order to connect file content and the metadata, because the content may be overwritten by allocated files. We discuss why it is important to perform a manual assessment of both metadata, content, and context in order to decide if the metadata and the file content can be connected.

B.1.3 Novelty of the new approach

Existing techniques for metadata carving do not use timestamps as a common identifier (dynamic signature) for different file systems. Even Dewald and Seufert (2017) describe that there is no magic signature for inodes in the Ext 4 file system, and they depend on semantics from Ext 4 in order to identify the locations of the inode metadata structures. However, metadata structures can easily be found by using string pattern matching based on equality, but unfortunately with a large number of false positives which have similar properties. Thus, obtaining high re-

¹Previous authors used "semantic filtering" to describe this, we have chosen to adopt that terminology

call and low precision for finding file system metadata entries. We also do not depend on a start date or an end date to identify the timestamps. This datetime agnostic nature of our approach allows the support of any file system that has closely co-located timestamps. While we do not depend on other semantics in order to identify the locations of these potential timestamps, we do utilize semantic parsers to validate and reduce the number of false positive hits of file system metadata structures significantly.

B.1.4 Importance for Digital Forensics

By using the novel generic metadata time carving approach, we do not need to specify which specific file types to carve for. The approach does not consider file types or file signatures; it only carves for metadata structures that potentially can be used for recovery of files. By accurately connecting the metadata to the corresponding file content, we also increase the evidential value of the files recovered, which most existing carving techniques do not accomplish.

B.1.5 Organization of this paper

We have introduced the objectives and the novel generic metadata time carving approach in the [Introduction](#) section. In the [Related work and contributions](#) section we discuss the current state of the art related to file and metadata carving. In the [Method](#) section we describe the carving algorithms and the methods we used for the experiments, and in the [Results](#) section we describe our results using precision and recall. Then we discuss our results in the [Discussion](#) section, and we conclude in the [Conclusion and further work](#) section.

B.2 Related work and contributions

There has been a significant amount of literature published on file carving, both using signature based contiguous carving, specific file type semantic carving or other statistical approaches in order to identify and carve for fragmented files. We address literature more specifically related to metadata carving, therefore, a complete list of all file carving literature will be out of scope.

[Mueller \(2008\)](#) introduced the idea of searching for NTFS timestamps as a string in unallocated space, since each timestamp is 64 bits and represents the number of nanosecond intervals since 1.1.1601. He also describes that the timestamps are in groups of 4 contiguous timestamps for each group. He created an `EnScript` (plugin for `EnCase`) that searched for the NTFS timestamps and bookmarked them. The `EnScript` uses a `grep` search for a particular date range, and it has an option for checking the next 8 bytes in order to only include hits that are followed by another valid timestamp. Use of the consecutive timestamps search

ideally reduces the number of false positives, but based on the comments on this blog post it appears as though the consecutive timestamps search does not work correctly.

In order to decrease the number of false positives, our idea is to search for a set of identical timestamps within a small window in order to detect metadata structures that describes files. Only metadata structures with a specific number of equal timestamps will be found. Our approach is more generic since we do not need to know how the timestamp is formatted (other than that they are closely co-located).

[Mccash \(2010\)](#) based his work on the EnScript from [Mueller \(2008\)](#), and adds the idea of using this information to detect MFT records and their attributes to extract the data content. He also describes that the script can be used to identify directory indexes and Registry key nodes.

B.2.1 Metadata carving

[Dewald and Seufert \(2017\)](#) consider the case in which the `Ext4` superblock or group descriptor table is corrupted or overwritten, and they use either metadata mode or content mode for parsing file systems or metadata carving respectively. In content mode their solution is to carve for inodes, which potentially provides the metadata necessary to extract the file content. However, the filename and inode number is not recovered in content mode. Since inodes have no magic bytes (except for extent headers in `Ext4`), they describe that they carve for them using pattern matching and analysis of the metadata. They conclude that their approach can reconstruct files from `Ext4` despite not knowing about the specific structure of the file system. They do, however, describe that they need multiple `Ext4` parameters in metadata mode for file system parsing. They describe that these can either be given by the user, or estimated based on the file system size.

Their work shows that carving for metadata structures is already suggested for file recovery. Their metadata mode approach explicitly depends on semantics specific for `Ext4` in order to include both metadata and file content, which enables parsing of the file system (not carving). Their carving approach, content mode, is not able to recover filename or inode numbers.

[Plum and Dewald \(2018\)](#) describe carving for APFS container superblocks, volume superblocks, or inode carving. APFS uses multiple container superblocks, and each of them may contain a reference to the previous container superblock. Within each container superblock they find volume superblocks, which describe specific volumes. These can be used to parse a specific volume and recover files from previous states of the file system. They further describe that inodes do not have a specific signature, but they can be carved using a combinations of the object type

and subtype inode fields. These inodes can be used to potentially recover files with the connected metadata.

Their approach is similar to the work of [Dewald and Seufert \(2017\)](#), but it differs by depending on specific semantics from APFS. Our generic approach will also work for the APFS inodes, since each inode has a set of contiguous timestamps. However, we have not implemented a semantic parser for APFS.

Work by [Garfinkel \(2013\)](#) describes the Bulk_Extractor tool which parses a large stream of data, using multiple threads, for feature extractions (URLs, e-mail addresses, Google search terms, Exif data, etc), which utilizes optimistic decompression before extracting the features. The features are detected based on rules which consider local context, which improve precision and recall. The features extracted do not need to be found within file entities. As part of the result, histograms of extracted features are created.

B.2.2 Evaluating recovered files

[Casey et al. \(2019\)](#) describe forensic processes such as authentication, classification and evaluation of recovered files. The problem is that different recovery tools do not use the same names for the same thing. They suggest to use Potentially Recovered before the authentication is performed. The authentication process is necessary in order to decide if the file is Fully Recovered, Partly Recovered, only Name and Metadata recovered, or Name Recovered. The decision should be based on confidence level after testing or trying to falsify different scenarios or claims.

B.3 Method

We use a generic automated approach to identify a potential set of timestamps within a specified threshold. We then record the byte positions in the image file where the set of timestamps were found. The approach is generic because it will identify the metadata structures in any file system that uses two or more timestamps of a user defined size to describe the temporal information of a file or directory. Since our approach is based on identifying equality between sequences of bytes, we do not require a start or end time for the timestamps. This approach will increase the false positive rate, but our semantic parsers attempt to exclude false positives by verifying if each timestamp location has a valid metadata structure for a specific file system.

As a proof of concept, we have added support for the recovery of metadata based on the identified timestamps in the Ext4 and NTFS file systems.

B.3.1 General Potential Timestamp Algorithm Description

We first describe the general potential timestamp algorithm at a high-level overview. A motivating factor for this algorithm is that often one or more MAC timestamps are identical. Furthermore, for file system entries in NTFS, ReFS, and ExtX the timestamps are closely co-located together in the metadata structure. Let m be the length in bytes of the potential timestamp, let T be an array of bytes of the data being searched. The user will define the length m (ExtX requires $m = 4$ and NTFS requires $m = 8$), as well as the length k of bytes to be searched after the potential timestamp, which we refer to as the search threshold. The crux of this search approach is that every non-overlapping m bytes in our binary data T is considered a *search keyword*, and we look for repetitions of this size m byte sequence within the subsequent k byte threshold window following the keyword. If the given byte pattern occurs one or more times within this threshold window, then we have identified a potential timestamp.

The mechanics of the search algorithm are based on the *sliding window* approach as is often found in malware analysis. The search begins at $T[0]$, in which the first m bytes are taken to be a potential timestamp, which contains the values $T[0 : m)$. We then check if this m byte keyword is equivalent to every non-overlapping m bytes in $T[m : (m + k))$, and keep count of how many exact matches have occurred. Given that we are searching for timestamps where at least two of them per metadata structure are equivalent, if no matches are found, we would then advance our search position by m bytes to position $T[0 + m]$. The advancement of m bytes assumes that timestamps will always fall on a multiple of m , and we do plan in implementing an *exhaustive* search functionality which checks for timestamps on every one or two bytes. Such skip sizes were chosen to enhance the speed of search substantially, as the current solution for 8 byte timestamps will process a disk image 8 times faster than an exhaustive search alternative that checks for potential timestamps on every single byte. If one or more matches are found we advance our search position by k bytes to position $T[0 + k]$. In either, case the entire search procedure is repeated from our new search position. This process is repeated for the entirety of the data T , except last k threshold of bytes. The skip size of k was chosen as it is the minimum size to avoid multiple hits for the same metadata structure. Note, for our current implementation k must be a multiple of m , otherwise the bytes being searched will be misaligned with the actual disk image timestamps. Algorithm 1 provides the pseudocode of the basic potential timestamp carving algorithm.

We provide an illustration for further explanation. In Figure B.1, the underlined bytes represent the potential timestamp keyword with $m = 8$, and the brackets represent the threshold of bytes, $k = 24$, being searched for matches.

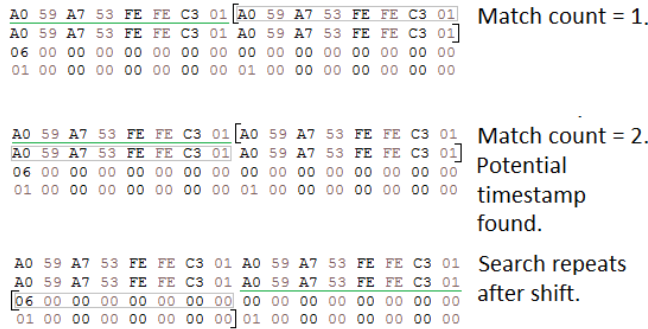


Figure B.1: Visual representation of the search procedure where three matching time stamps are searched for. The underlined byte sequence represents the current byte sequence being tested as a possible timestamp. The subsequent bytes in brackets represent the search threshold for checking matches. The bytes in grey boxes represent checks for matching byte sequences. In the second row, after a second match is found, we advance the search procedure ahead by k bytes, where the process is repeated.

This general search by itself likely produces a large number of false positives, thus we placed an additional condition to improve the algorithm’s precision. We determine if the potential timestamp to be searched for consists of a single repeated byte value, and if so, we skip the search procedure and move forward m bytes. This is to avoid fruitless searches on blocks of repetitive bytes. Examples of such timestamps we wish to avoid are $0x0000000000000000$ and $0xFFFFFFFFFFFFFFFF$.

Here we approximate the time complexity of the worst case search scenario. We assume that the entire disk image could be read into memory at once to simplify our approximation. In this scenario, we are searching a disk image with no sets of co-located bytes that are repeated two or more times (we get no hits). In this fashion, we cannot perform any byte window skips after searching through our search window threshold. We also perform the most generic type of potential timestamp carving, where we do not consider repetitive byte sequences. In this way, we cannot skip any particular keyword byte sequence, since all byte sequences will be considered to be potentially valid timestamps. Thus, every m sequence of non-overlapping bytes on the array of disk image bytes T will have a search procedure performed on it, in which the entire threshold window of size k is searched.

Given this worst case scenario, the computational time complexity is $O\left(\left\lfloor \frac{|T|}{m} \right\rfloor \times \frac{k}{m}\right)$. The integer of the cardinality of T divided by m is the number of byte sequences that have a search procedure performed on it, and it is multiplied by the maximum number of byte matching checks, the threshold of bytes k divided by m where m is a factor of k .

B.3.2 Practical Potential Timestamp Program Details

The general timestamp carving algorithm was implemented in C++, which we refer to as *cPTS*, and is supported by a number of libraries. Since disk images under analysis are likely greater than memory, we use the cross-platform mio memory mapping library². This allows us to read in 1 GB of memory at a time, and read the image as a series of arrays. However, once the potential time stamp carver arrives within the last 4096 bytes of the gigabyte in memory, we load a new gigabyte into memory from the search point relative to the disk, as to handle directory entries that are spread across segments. For converting datetime formats into decimal form, we used the Date library³. The program outputs a text file list of all the potential timestamp locations (in byte offset) that were found.

B.3.3 Semantic parsers

Identifying closely co-located potential timestamps based on equality will provide generic results, but will contain many false positives due to its genericity. Therefore, parsers which utilize the semantics of the expected metadata structures of specific file systems were developed for more accurate automation. Our Python 3 parsers accept the timestamp locations from the generic timestamp carver and the disk image as input, as seen in Figure 2. Our experiments utilize this process.

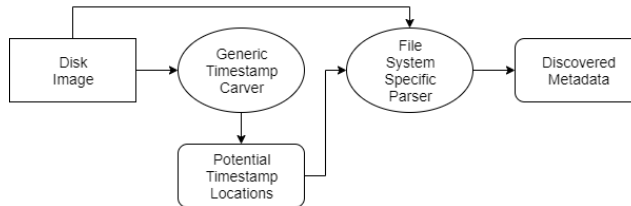


Figure B.2: Diagram for system deployment, used in our experiments.

NTFS semantic parser

The NTFS script assesses if the potential timestamp is within the Standard Information Attribute (SIA), or a Filename Attribute (FNA). To reduce the number of false positives, we exclude any potential timestamps from before the year 1970 and years beyond 2100. The script outputs metadata information contained in the SIA, FNA, and Data Attribute if possible. When attempting to parse out a full MFT entry, we start with the identification of the SIA. Once we identify the location of the SIA header, we use the length of the SIA to see what the header of the next attribute is, ultimately searching for the Data Attribute. If the next attribute header

²<https://github.com/mandreyel/mio>

³<https://github.com/HowardHinnant/date>

is not the Data attribute, and the first byte of the attribute type is less than 0x80, we read the length of the attribute and perform another skip down to the next attribute. Though, if the next attribute is an FNA, we will output its metadata information, and add the byte location of its first timestamp to a list of future timestamps to avoid. This is done so we do not get redundant FNA outputs. Encountering at least one FNA is required to read out a potential Data Attribute we encounter. This is repeated until we find the Data attribute, or abandoned if we identify an attribute type where its first byte is greater than 0x80 or if we have searched more than the length of the potential MFT entry. A limitation of this work is that we do not currently perform MFT entry searching starting from identified File Name Attributes, where their timestamps are more likely to be reliably⁴ found due to their relatively unchanging nature compared to Standard Information Attributes (Cho 2013). Another limitation is that we currently do not support Alternate Data Streams. Relevant MFT entry information is output into the file `NTFSResults.txt`, and if the file is resident, we also include the resident file encoded in ASCII.

Ext4 semantic parser

The Ext4 Python 3 script uses the text file produced by the cPTS tool containing the potential timestamp locations, the disk image, the byte position to where the partition starts, and the assumed block size. For conducting a similar search as was done in the NTFS parser, these parameters and a default static inode size of 256 bytes are the only assumptions we make.

Like the NTFS parser, we use the potential timestamps as anchor points and test for various semantics at local offsets. But now, we also verify information found in likely directory entries. For Ext4, we test all possible offsets backwards for file flags of interest: 0x04 for directories, 0x08 for regular files, and 0x0A for symbolic links. For Ext4 inodes not using extents, we ensure the relative position of bytes 36-39 of the inode are unused. For inodes using extents, we check that the relative position of its extent header magic number is equal to 0xF30A. We then conduct additional tests to increase the likelihood of having discovered an inode, such as using some of Dewald and Seufert (2017) timestamp consistency tests, that the size of the file appropriately fits the sector count, and that the size of the file cannot exceed the size of the disk image. All inodes found to be sufficiently valid have their likely starting points added to a list.

The great difficulty in performing full file extraction in Ext4 is connecting inodes and their directory entries when not relying on superblocks, block group descriptor tables, or inode bitmaps. Such connections will need to be made if these metadata structures are irrecoverable. The inode contains the majority of the metadata for

⁴FNA timestamps are updated mainly on MFT entry creation and on file name change

the file, but its associated directory entry contains the filename and the inode number. The task was then to connect inodes based on their physical position to their actual inode number. We pursued solving this problem solely relying on information we can find locally within or around a validated inode.

Our solution revolves around using the verified inodes of directories that have not been deleted, as this gives us ground truth information about inode numbers and filenames, including the inode number of the directory itself. Verification is performed by following the directory's extent or direct block pointer to its first directory entry and checking if bytes 4-6 are 0x0C0001 (the length of the entry and the first byte of the length of the name). For Ext4, we perform two passes over the disk image. The first pass collects information on valid inodes, creating a dictionary of inode numbers and filenames found in all validated directories, as well as a so-called synchronization list. This synchronization list is a recording of the first validated inode of a directory found per block group, wherein we record the inode's location and inode number. The second pass uses the inode dictionary and the synchronization list to make inode number estimations of validated inodes, outputting the inode number alongside its likely filename.

We can estimate the inode number of potential inodes in two different ways. The first way uses the positions found in the inode synchronization list, where we can then make estimates of the inode numbers of the validated inodes that are in the same block group as the validated directory being used from the inode synchronization list. Assuming that the Ext4 inodes are a static size of 256 bytes, the following is the equation of the estimated inode number e , where dn represents the inode number of the validated directory being used from the inode synchronization list, vl represents the validated inode location, and dl represents the location of the inode of the same directory obtained from the inode synchronization list:

$$e = dn + ((vl - dl)/256) \quad (\text{B.1})$$

Using the inode synchronization list, we can estimate inode numbers prior to encountering the directory its filename and inode number are held in (in case of deletion). During the second pass (while we are estimating inode numbers), when encountering a validated inode of a directory, we update the entry in the inode synchronization list for the current block group we are in. This allows for rather local synchronization of the current inode number.

The second way of estimating inode numbers uses the previous estimations, and the inode dictionary. The first time we make an estimate for a particular inode number, we update its entry in the inode dictionary by adding in the inode's file

version number and created time as parameters. If the inode number did not exist in the dictionary prior to the inode estimation, we simply create an entry with these parameters. Once the entry has been updated or created, it cannot change. The file version number and the created time should be relatively unique per inode, and so when parsing future inodes we check if we have already recorded its file version number and creation time in the inode dictionary, and write out the associated recorded inode number and filename.

We output a text file and csv file, *ExtResults*, where we record pertinent inode and directory entry information. We list both the estimated inode number and filename (using the inode number as a key in the inode dictionary), and the recorded inode number and filename (using the file version number as the key in the inode dictionary).

B.3.4 Experimental setup

We used an external USB thumb drive and wiped the partition⁵ using the tool `dc3dd v. 7.2.646` ([Department of Defence Cyber Crime Center 2012](#)) in macOS Mojave v. 10.14 (Linux could also be used).

```
sudo dc3dd hwipe=/dev/rdisk8s1 hash=md5
```

Listing B.1: Wiping USB thumb drive.

B.3.5 Experiment - NTFS reformatted with exFAT

We formatted the device in Windows 10 using NTFS, where we created 50 files, and for each file type we named them File1, File2, File3,...File10. Five different file types were used, and there were 10 files for each of these file types, where the extensions were added to the filename. Then we reformatted the file system using exFAT. Fragments, or the complete MFT table should still be available. Finally, 10 text files were added to the reformatted image.

The files created by the batch file give us a known basis in order to test precision and recall. We know all the file names and content, as the base forensic image (`ntfsbase.dd`) of the partition was created before reformatting it with exFAT. After the reformatting and the creation of 10 text files, we created a new forensic image of the partition (`nftsexfat.dd`) using `dc3dd`.

We measured the false positive and false negative rates by comparing the carved metadata results with the filenames we found in `ntfsbase.dd`. A false positive is a hit location not found within metadata describing a file or directory, while a

⁵We wiped only the partition, shown in Listing B.1, because macOS gave a resource busy message when trying to wipe the complete raw disk

false negative is a set of timestamps not identified as a hit, but which is located within metadata describing a file or directory. Finally, we calculated the precision and recall (Perry et al. 1955) of the methods implemented.

```
cPTS ntfsexfat.dd 8 24 3
```

Listing B.2: Command to find possible NTFS timestamps.

We used a timestamp size of 8 bytes, a search threshold of 24 bytes to search for equivalent timestamps, where at least 3 timestamps are equal. The output from Listing B.2 was saved to the file cPTS.txt.

```
python3 ntfParser.py cPTS.txt ntfsexfat.dd
```

Listing B.3: Command to identify if the hits are SIA or FNA in a MFT entry, and outputs information from these attributes and from the DATA attribute if possible.

The next step was to assess if each hit was part of a standard information attribute (SIA) or a file name attribute (FNA). Then the script in Listing B.3 identifies the Data attribute and shows the resident data or the non-resident data runs.

Additionally, we tested if X-Ways, EnCase, and Recuva were able to recover the previous NTFS partition or to find unallocated MFT entries using the same forensic image.

B.3.6 Experiment - Previous Ext4 reformatted with NTFS

For the Ext4 experiment we used Linux Mint 18.2 and Windows 10. In Linux we wiped the storage device using the command `shred`, overwriting using zeros. Then we formatted the storage device with Ext4, and mounted it. We created 50 directories with 500 files in each directory. The files were numbered from 1.txt to 25000.txt. The file names correspond with the number of bytes (a's) in each file. The text files were selected because they are more difficult for carving tools to recover, as there is no signature. Therefore, recovery of these text files rely only on metadata recovery. The file system was unmounted, and a ground truth forensic raw image named `expExt4.dd` was created using `dd`. Then it was mounted to a Windows 10 OS, and quick reformatted with NTFS using a 4096 byte cluster size. Then 10 files were created. A raw image was created with the name `Ext4NowNTFS.dd`.

```
cPTS Ext4NowNTFS.dd 4 12 2
```

Listing B.4: Command to find possible Ext4 timestamps.

We used a timestamp size of 4 bytes, a search threshold of 12 bytes to search for equal timestamps, where at least 2 timestamps are equal. The output from

Listing B.4 was saved to the file cPTS.txt.

```
ext4Parser.py cPTS.txt Ext4NowNTFS.dd 0 4096
```

Listing B.5: Command to parse Ext4 inodes.

In Listing B.5 we start at byte offset 0 in the image, the block size is 4096 bytes, blocks per group are estimated to block size * 8 = 32768.

B.3.7 Limitations

We do not know at the start of the investigation if there has been a previous file system. We suggest to search for known signatures of volume boot records/super-blocks, which may document the start of a previous partition. We also suggest to try to recover the partition before attempting our metadata carving approach.

The output results of the prototype should be assessed by file system experts (or expert systems) in order to assess if a file (name, metadata and content) can be fully or potentially recovered. This is called authentication, and it includes an evaluation of the classifier, the results, and finally a confident decision (Casey et al. 2019).

Our prototype tool will not work properly on a manipulated file system where sectors or clusters are removed or added, because the mapping between data runs (extents) and the cluster locations are not in sync. Our approach depends on the existence of metadata structures in the unallocated area of the partition, and we assume that the start of a data run (extent block pointer) is relative to the start of the partition. The prototype also does not consider fixup values found in the last two bytes in each sector in a MFT entry. Since both SIA and FNAs are among the first attributes in a MFT entry, we assume they normally will not be found within a fixup value. We do not consider files that use multiple MFT records if the DATA attribute is not located in the first of these MFT records. The currently implemented semantic parsers do not consider FNAs in directory indexes, but the cPTS tool will locate them. Lastly, we do not consider Ext4 inode record sizes larger than the standard 256 bytes.

We are aware our experiments have a small sample size, and we have not included testing on real forensic images from real criminal cases in order to comply with legislation. We are also aware that we have only tested using specific versions of Linux and Windows, which opens for possible deviations if other OSes are selected. We selected this small sample because it allows us to know the ground truth of the content, which is difficult when using a system volume where the OS is continuously creating and deleting files. Using an unknown source makes it difficult to compute precision and recall, and gives us no control of the different variables that may affect the results.

B.4 Results

The `cPTS` command took 13 seconds to run on a 2 GiB byte `dd` image file. The `ntfsParser.py` took less than 1 second, while the `ext4parser.py` took 8 seconds. This is faster than the runtime performance of [Dewald and Seufert \(2017\)](#) tools, however they additionally exported the file content automatically.

B.4.1 NTFS metadata carving

For each discovered MFT entry, we know the SIA is associated with the FNA because of the distance between them is less than 1024 bytes, which could visually be verified or falsified by interpreting the byte location for the SIA and FNA timestamp hits. The Data attribute belongs to the FNA because we skipped to the next attribute until we found the unnamed Data attribute, which we found within the next 1024 bytes. This means we have the name, metadata and the content, and since we have the data runs, we know this record is non-resident and potentially recoverable. In order to test if the original content can be connected to the metadata, we need to extract the content and perform hypothesis testing. Extraction of the content based on known data runs is described by [Carrier \(2005\)](#).

	TP	FP	FN	Precision	Recall
SIA matches	162	1	0	0.9939	1

Table B.2: Precision and Recall for finding MFT records in `ntfsexfat.dd`

In [Table B.2](#) we focus on files/directories and Standard Information Attributes (SIAs). We know each base MFT entry has one SIA. Since we have 79 files (50 files and the 29 system generated files and directories) in our experiment, we know there must be 79 SIAs in the MFT table. We also know there must be 79 SIAs in `$LogFile`⁶ and 4 SIAs in `$MFTMirr` ([Carrier 2005](#), p. 303). This gives a total of 162 SIAs. We found all 162 SIAs, and only one of the hits was a false positive. We did not have any false negatives for SIAs. Our computation of precision and recall is shown below.

$$Precision = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} = \frac{162}{163} = 0.9939$$

$$Recall = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = \frac{162}{162} = 1$$

⁶The only file system transactions we performed were creating files.

For our simple experiment, it was easy to verify the found MFT records with the known base. However, with a forensic image from a real criminal case hypothesis testing must be performed in order to verify that the data runs found in the MFT entries still can be connected to the file content, and that they are not completely or partly overwritten by another file.

Hits from \$LogFile

In addition to entries in the MFT Table and the MFTMirr, we found MFT entries within the unallocated \$LogFile, but in this case they did not include data runs or resident data. It was interesting to observe that our created files had a Data attribute within the \$LogFile with the size of only 0x18 bytes, which only contains the attribute header. This means that we cannot fully recover files from all MFT entries found in \$LogFile.

B.4.2 Ext4 metadata carving

We were able to recover all inode metadata entries that were not overwritten by NTFS, but reformatting Ext 4 with NTFS in our experiment wiped approximately 20257 inodes from the inode table. When using flex groups, the inode tables are all located continuously in the first block group (Dewald and Seufert 2017), instead of being divided into its corresponding block group. Our current observations show that our approach supports both types, and does not depend on information from the block group descriptors. For each extent found in an inode, an extraction of the file content is easily performed by extracting from the extent start block as well as the number of blocks contained in this extent. A deleted inode will normally have the extents zeroed out, making the inode connection to the file content infeasible. However, since we find hits for duplicated inodes in different physical locations, we may be able to recover the file content by using extents found in these duplicates. We did not extract the files from the overwritten Ext 4 image due to their quantity, and thus are only discovering potentially recovered files.

In order to measure precision and recall, we created the same Ext4 dd images again following the same method as previously described, but in addition we added the real inode number as an attribute to each inode for the 25000 text files and 50 directories we created in the experiment using the `Linux attr` command. This way we could compare our estimated inode number and the recorded inode number with the real inode number included in the attribute.

For the original and reformatted images, we conducted two precision and recall experiments. The first calculates the precision of attributing our recorded or estimated inode number to the true inode number of the inode, and we also calculate the recall of finding our known files with correct attribution. For calculating recall,

if at least one inode per inode number was found and we correctly estimated or recorded its inode number, it counted as a true positive, where the duplicate inodes (with respect to its true inode number) were removed. Table B.3 shows our results for the non-reformatted version of the dd file. A similar process was done for the reformatted dd file, as seen in Table B.5, except rather than calculating the recall we simply record the number of discovered inodes per method of inode number estimation. This was done since we cannot be certain of how many false negatives were either due to our methods, or due to the file system reformatting. Note that since at least 20257 of 25050 inodes were wiped from the table, and we recover 5755 inodes, that we are potentially recovering at least 963 previously overwritten inodes.

The second experiment calculates the precision of our method to correctly classify inodes, whether they were from known files or not. False positives in this case are inode hits that contain junk information. Table B.4 shows the precision from the non-reformatted experiment, and Table B.6 shows the precision of the reformatted experiment. In both experiments, we obtained 100% precision. We cannot calculate the recall in this case, since we cannot know how many inodes (from the inode tables and copies throughout the disk) exist on the image.

	TP	FP	Precision	Recall
Recorded inode matches True inode	77481	194	0.9975	1
Estimated inode matches True inode	27336	50339	0.3519	1
Est or Rec inode Matches True inode	77481	194	0.9975	1

Table B.3: Precision and Recall for finding and attributing iNode numbers for known files in expExt4Attr.dd

TP	FP	Precision
77675	0	1

Table B.4: Precision of iNode classification for non-reformatted image.

	TP	FP	Precision	Files Found
Recorded inode matches True inode	15544	41864	0.2078	4848
Estimated inode matches True inode	7091	50336	0.1235	5755
Est or Rec inode Matches True inode	16553	40874	0.2882	5755

Table B.5: Precision and Files Found for finding and attributing iNode numbers for known files in Ext4AttrNowNTFS.dd

TP	FP	Precision
57426	0	1

Table B.6: Precision of inode classification for reformatted image.

B.4.3 Commercial tools

The results of the tool testing described in this section are shown in Table B.7.

NTFS now exFAT

	EnCase	X-Ways	EaseUS	Bulk_Extra	cPTS
NTFS metadata	N	N	Y?	Y	Y
Ext4 inode	N	N	N	N	Y

Table B.7: Tool testing - Carve for metadata from previous file system when reformatted with another file system.

We created a case in `X-ways v 19.8` and imported the file `ntfsexfat.dd`. Using the feature refine volume snapshot, we selected the particularly thorough file system data structure search, and checked search FILE records everywhere. The X-ways manual describes that this search should be able to find MFT entries from unallocated space. However, the tool did not find any of the MFT entries from the previous NTFS file system.

X-Ways also has a function that should be able to scan for lost partitions, but this feature was disabled. X-ways was not able to detect the previous partition automatically. We created a new image of the complete USB thumb drive, including the MBR. In this case, it was possible to search for lost partitions, but X-Ways did not find the NTFS partition or its MFT entries.

We also tried to carve for file content, and X-Ways was able to carve the contiguous files, but not the tiff files that had two fragments.

EnCase v8.08 was not able to find the MFT records from the previous NTFS partition in `ntfsexfat.dd`. We selected the Full Investigation pathway, which includes the relevant Recover Folders (which should locate hidden files in FAT and NTFS volumes), and the Windows Artifact Parser with MFT Transactions selected. According to the EnCase manual, the Recover Folder option should be able to recover NTFS files from unallocated clusters. Since EnCase is a closed source tool, we do not know how this is implemented. EnCase did find the Backup VBR (Volume Boot Record) when we searched for it using the Partition Finder.

However, it was not possible to recover the partition in disk view.

We tried using `EnCase` to carve for picture files (bmp, jpg, png, and tiff). The content of all 10 bmp files were found, but also 178 extra false positives. Each of the 10 jpg file contents were found, and with no false positives. We missed one of the png files, but found the others. `EnCase` did not find any of the tiff files. This is because `EnCase` searched for the tiff signatures 49492A000A, 49492B00, 4D4D002A, 4D4D002B, while the tiff file created in Windows 10 had a signature 49492A008E.

We tested the storage device we performed the experiment on with `EaseUS Data Recovery Wizard v11.15` (previously named `Recuva`), and it was able to identify all the 50 files and their content. Since this tool is closed source, we assume they performed a partition recovery by using the VBR backup. For partition recovery they only showed the size, the date created and the path. It also carved for files, but the carved files did not include the metadata. Lastly, we found the tool could not correctly carve the fragmented files or the text files.

Ext4 now NTFS

Carving for ASCII text files is not supported by `EnCase v8.08`, and therefore carving for the ASCII text files in the previous Ext4 file system is not possible. However, we tried to carve for 10 different supported file types and measured the run time to be 27 seconds on a 2 GiB disk image. Next we tried 100 different supported file types, which took about 1 minute. Finally, we selected all 349 supported file types, but we canceled the progress after 6 hours.

We performed a test using the tool `EaseUS Data Recovery Wizard v11.15` (previously named `Recuva`), but it did not find any of the 25000 text files within the experiment storage media. All the 16 files it listed were allocated NTFS files, and it did not find any previous Ext4 partition.

Additional testing

In order to identify an expected number of false positives using our tools, we performed additional testing on a real 16 GiB raw image from a cell phone that did not have any MFT records and an 80 GiB Windows 10 machine with no Ext4 inodes. Searching for inodes in the Windows image we found 3 false positives, and searching for MFT entries on the cellphone image we found 8 false positives.

Lastly, we tried running `Bulk_Extractor` on our reformatted images, wherein it found all of the filenames of the MFT entries in both images, but did not recover any of the metadata information from Ext4.

B.5 Discussion

In our experiments, we reformatted one file system with a different file system. If we know it has been reformatted, we can first try to recover the previous file system by assessing the backup VBR or superblocks. This may potentially recover the file system, and we can accurately find files that the other file system did not overwrite. However, if the other file system has overwritten parts of the previous file system, then we may need to use our approach to find the parts that are not overwritten.

B.5.1 Discussion related to NTFS

For the timestamp hits where we found NTFS MFT entries with a resident data attribute, we can reliably connect the metadata and the file content (Casey et al. 2019). This is because the resident data is found within the MFT entry. Normal file carving will not find small ASCII text files that are resident in MFT entries, because these files have no signature within their content.

MFT entries could potentially be found in multiple sources; memory dumps, un-allocated space, in the allocated system files like \$MFT, \$MFTMirr, \$LogFile, hiberfil.sys, etc. The allocated \$MFT should normally be accurate if not manipulated.

Our approach does not need a complete MFT table, nor a complete MFT entry. For instance, we do not use the MFT entry header at all. However, we rely on that the SIA, FNA or Data attributes are co-located within the size of a typical MFT entry. This allows recovery of partly overwritten metadata. Currently, we do not search for a Data attribute if the SIA is not found, but we plan to change this dependency in later releases of the tool. This change will allow detection of FNAs in index entries.

We need to use the \$Bitmap of the new file system to identify which clusters/blocks are in use. If a data run found in a recovered metadata structure uses one or more of the clusters allocated by a file in the new file system, we must assume that the file content is partly overwritten.

It is important that the investigator is knowledgeable about the file systems found when using our approach. First of all, our approach uses the data runs found within the NTFS metadata, and the first data run for the MFT entry is relative to the start of the file system, and we need to use the correct cluster size used by the previous NTFS file system. Furthermore, subsequent data runs are relative to the previous run (Carrier 2005, p. 258).

Since we are proposing a generic approach, we cannot automate the extraction

of files without considering the specific context, which requires context based recovery tools or manual expert assessment. For instance, the storage media could have first had an NTFS file system, and then been reformatted with NTFS or another file system. Then of course the context is different, and must be taken into consideration.

We have shown that popular digital forensic tools, such as the current versions of X-Ways or Encase, do not necessarily find the MFT entries when the NTFS system is reformatted to exFAT. This may incorrectly cause the investigator to utilize file carving, which of course does not include the metadata, but only the file content. Such actions would result in missing pertinent files, and partly recovered fragmented files.

B.5.2 Discussion related to Ext4

If a user deleted files using the command line `rm` tool, or by emptying the trash, some of the important fields in the deleted inodes are set to 0, for instance the total size, the link count, the number of extents, and the extents fields. The timestamps for changed, modified and deleted are set to the deletion time, while the accessed and created are not changed. However, since we may find duplicate inodes in locations outside the inode tables, we may find previous versions of a deleted inode, which can allow us to recover the content and the metadata.

B.5.3 Addressing Our Statistics and Current Challenges

Statistics: Our high precision and recall does not indicate that our tool will find nearly all metadata entries without error, but it indicates that it will work well *given* that the metadata structures include repeated timestamps. When creating the disk images, files were guaranteed to have at least two or three identical timestamps. If our current solution is applied on a realistic disk image, the percentage of identified metadata entries should effectively be the same as the percentage of metadata entries on the disk that have the identical timestamps.

Metadata Remnants: Our approach does not differentiate between MFT records/inodes found in the the MFT/inode table and the instances found in the journal or elsewhere. This is not a limitation, this is a feature since remnant from metadata structures describing files can be scattered across the file system.

Virtual Machines: We assume that not all virtual storage in a Virtual Machine is wiped on creation. This means there could be remnants from metadata from previous host file systems if the area assigned to the virtual storage has previously been allocated to the MFT/inode table or to a previous journal. Our approach will also find these timestamp locations.

Our approach can also be used to identify metadata currently not linked to existing file content, which is important for event reconstruction.

B.6 Conclusion and further work

The aim of this research was to answer the following research questions.

- Can we reliably use time as a generic identifier to carve for file and directory metadata structures in different file systems?
- What is the reliability of recovery of files using the discovered metadata in Ext4 and NTFS?

We have shown that a set of similar timestamps can be used as a form of dynamic signature (magic identifier), and we carve for these by using a simple byte matching algorithm. Then we use file system semantics in order to interpret metadata structures, and manually extract the resident or non-resident files. Finally, a file system expert evaluates the classification, authentication and makes a decision for final classification of the manually recovered files.

We argue that a manual evaluation of the reliability of the connection between the metadata and file content is necessary, and that this assessment is context based and should be manual for non-resident file content. The manual assessment could, however, be supported by automated tools.

Connecting the inode number and the file name is challenging in Ext4 when an inode table is partly wiped. However, connecting the inode metadata with its corresponding file content is still possible even without the correct inode number or file name. On the non-reformatted Ext4 image, we were able to achieve greater than 99% precision when attributing an inode number to discovered inodes, and full recall. For the reformatted Ext4 image, it was possible to achieve greater than 28% precision in correctly attributing inode numbers to found inodes. Of the known 25050 known files and directories in the original image, we were able to recover inodes for 5755 of them. Since at least 20257 of the 25050 inodes were wiped from their inode tables, this means that we are potentially recovering at least 963 inodes.

When accurately connecting the metadata to the file content, we increase the evidential value of the evidence. We should not only use file carving when searching for files in unallocated space, since there may be pertinent metadata structures within unallocated space. As long as metadata structures exist in unallocated space, our generic metadata time carving approach combined with the semantic parsers can be used to connect metadata to file content. Knowledge of the file

system context is necessary in order to assess the accountability of the connection between the metadata carved and the file content recovered.

When extracting inodes, our method had 100% precision for both the original Ext4 image and the reformatted image. Even though our tool outperforms commercial tools given our specific experimental setup, our tool should still be considered a proof of concept prototype.

Support for file systems other than Ext4 and NTFS is left for further work. Automation of file recovery is possible, but requires context aware features. Further research is needed in order to improve the accuracy of connecting Ext4 inode number and file name to the inode entry, especially in the context of partially wiped inode tables.

Input: Raw disk image T as a byte array
Output: Potential timestamp positions (in bytes)
 m # Length of timestamp;
 k # Length of search threshold;
 h # Theshold of matching timestamps;
 $i = 0$ # Byte location;
bool repeatedBytes = False;
while ($i < |T| - k$) **do**
 searchString = $T[i : (i + m)]$;
 decimalDate = *stringToDecimal*(*searchString*);
 repeatedBytes = *checkRepeatBytes*(*decimalDate*);
 if (*!repeatedBytes*) **then**
 matchCount = 0;
 $j = i + m$;
 while ($j < i + m + k$) **do**
 testBlock = *stringToDecimal*($T[j : j + m]$);
 if ($(i\ testBlock == decimalDate)$) **then**
 matchCount += 1;
 end
 $j += m$;
 if ($matchCount \geq (h - 1)$) **then**
 Print Byte Location i ;
 $j = i + m + k + 1$;
 $i += (k - m)$;
 end
 end
 end
 $i += m$;
end

Algorithm 1: Basic Potential Timestamp Carving Algorithm.

Bibliography

- Apple (2004). HFS plus volume format. Last visited 2019-12-20.
- Bhat, W. A. and Wani, M. A. (2018). Forensic analysis of B-tree file system (Btrfs). *Digital Investigation*, 27:57 – 70.
- Carrier, B. (2005). *File System Forensic Analysis*. Addison-Wesley Professional.
- Casey, E., Nelson, A., and Hyde, J. (2019). Standardization of file recovery classification and authentication. *Digital Investigation*.
- Cho, G.-S. (2013). A computer forensic method for detecting timestamp forgery in ntfs. *Computers & Security*, 34:36 – 46.
- Department of Defence Cyber Crime Center (2012). Dc3dd. Last visited: 2019-09-19.
- Dewald, A. and Seufert, S. (2017). Afeic: Advanced forensic Ext4 inode carving. *Digital Investigation*, 20:S83 – S91. DFRWS 2017 Europe.
- Ext4 development team (2019). Ext4 header file. Last visited: 2019-09-11, code from the master development on github.
- Garfinkel, S. L. (2007). Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2 – 12.
- Garfinkel, S. L. (2013). Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:56 – 72.
- Göbel, T. and Baier, H. (2018). Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. *Digital Investigation*, 24:S111 – S120.

- Hamm, J. (2009). Extended fat file system. <https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf>, visited 2018-09-16.
- Hansen, K. H. and Toolan, F. (2017). Decoding the apfs file system. *Digital Investigation*, 22:107 – 132.
- Mccash, J. (2010). Timestamped registry & NTFS artifacts from unallocated space.
- Mueller, L. (2008). Search for windows 64 bit timestamps.
- Nordvik, R., Georges, H., Toolan, F., and Axelsson, S. (2019). Reverse engineering of ReFS. *Digital Investigation*, 30:127 – 147.
- Nordvik, R., Porter, K., Toolan, F., Axelsson, S., and Franke, K. (2020). cPTS carve for potential timestamps. Last visited 2020-03-20.
- Perry, J. W., Kent, A., and Berry, M. M. (1955). Machine literature searching x. machine language; factors underlying its design and development. *American Documentation*, 6(4):242–254.
- Plum, J. and Dewald, A. (2018). Forensic apfs file recovery. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018*, pages 47:1–47:10, New York, NY, USA. ACM.

Appendix C

Publication C: Timestamp prefix carving for filesystem metadata extraction

Timestamp prefix carving for filesystem metadata extraction, Kyle Porter, Rune Nordvik, Fergus Toolan and Stefan Axelsson. In: *Forensic Science International: Digital Investigation Volume 38*. September 2021, Pages 301266. DOI: <https://doi.org/10.1016/j.fsidi.2021.301266>

Abstract

While file carving is a popular and effective method for extracting file content from unallocated space in a forensic image, it can be time consuming to carve for the wide variety of possible file signatures. Furthermore, file carving does not connect the discovered file to its filesystem metadata. These limitations of file carving are the advantages of *Generic Metadata Time Carving*, in which filesystem metadata is searched for by first finding repeated co-located timestamps using a potential timestamp carving algorithm. The potential metadata is verified by a filesystem specific parser, and the pointer within the metadata to the file data may allow for full file recovery. Currently, a limitation of the Generic Metadata Time Carving method is that it will only find metadata records that have multiple equivalent timestamps, thus missing metadata records and files with differing, but very similar, timestamps. Therefore, in order to improve the recall of the Generic Metadata Time Carving methodology, we have designed and implemented a prefix matching potential timestamp carving algorithm. We apply our experiments to realistic NTFS and Ext4 forensic images, in which we compare the precision and recall results for differing prefix lengths. Our results indicate that using prefix-based potential timestamp carving can yield significantly greater recall for extracting filesystem metadata records, with little to no reduction in precision as compared to the original exact potential timestamp carving method.

C.1 Introduction

File carving is an established digital forensics method for extracting files that cannot be found using the filesystem, and while extremely useful it is not without its faults. When applying popular file carving tools such as Scalpel ([Richard III and Roussev 2005](#)), one must attempt to search for all possible file signatures that are relevant to the case, which not only makes the search process more time consuming,¹ but more importantly, the file signature database may be incomplete. Omitted file signatures means missing files when carving.² File carving also does not have an automated method for connecting filesystem metadata to the discovered file (assuming the metadata still exists on disk) ([Dewald and Seufert 2017](#), [Nordvik et al. 2019](#)), and has difficulty dealing with fragmentation ([Garfinkel 2007](#)).

¹While Scalpel uses a modified version of the single-pattern string matching algorithm Boyer-Moore ([Boyer and Moore 1977](#)) (as of 2018 ([Bayne et al. 2018](#))) and causes header-footer matching to run in $O(sn)$ time where s is the number of header-footer signatures and n is the length of the data being processed, [Zha and Sahni \(2010\)](#) created *FastScalpel* which uses the multi-pattern string matching algorithm ([Aho and Corasick 1975](#)) that performs header-footer matching in linear time.

²In the case of general data carving methods (for example, the Bulk Extractor ([Garfinkel 2013](#))), omitted regular expressions may mean missed data such as credit card numbers, social security numbers, etc.

These limitations of file carving are the advantages of *Generic Metadata Time Carving* (GMTC) by Nordvik et al. (2020). GMTC is a metadata carving method that uses a simple string matching algorithm, a *potential timestamp carver*, to search for equivalent and closely co-located byte sequences in order to find potential filesystem metadata record timestamps. After the locations of the potential timestamps are found, a filesystem specific parser either accepts or rejects the surrounding content as filesystem metadata. The filesystem metadata record may then be used to retrieve the associated file whether or not the file is fragmented, where the ability to do so is dependent on the filesystem and its policy for deleted files. This technique thus connects filesystem metadata to the file data, enabling at least *metadata and content recovery* (Casey et al. 2019). Furthermore, the method does not depend on file signatures, since timestamps are essentially a dynamic signature for all filetypes.

Use-cases of GMTC and other metadata carving methods include scenarios where the filesystem has been severely damaged or overwritten. Moreover, GMTC can also be used to find metadata records hidden in perfectly functioning filesystems.

Generic Metadata Time Carving has several limitations as well. The largest of which is that the method can only find metadata records that contain precisely equivalent timestamps, thus limiting the recall of discovered files in an image to the same number of metadata records that contain at least 2 or more equal timestamps. In order to improve this limitation, we have created and implemented a new potential timestamp carving algorithm that performs timestamp prefix matching. Allowing for some minor tolerance of difference between potential timestamps, we aim to improve filesystem metadata record recovery recall for GMTC without significantly reducing its precision. We formalize our research questions below:

1. How does the value of the prefix parameter affect the precision and recall of the Generic Metadata Time Carving method?
 - (a) How does the original Generic Metadata Time Carving method compare with the prefix matching implementation?
2. Do the experimental results indicate that Generic Metadata Time Carving, prefix matching or otherwise, may be used in realistic digital forensic scenarios?

We hypothesize that timestamp matching using shorter prefixes will result in more potential timestamp matches, thus increasing recall, while reducing precision, as

is often seen in precision-recall trade-offs.³ Furthermore, due to the time complexity of potential timestamp carving, previous work on this subject, and the size of our data we hypothesize that even relatively large images can be processed in a reasonable amount of time. To test our hypotheses, our prefix matching method of Generic Metadata Time Carving is applied to two NTFS images and one Ext4 image, where the sizes of the images range from 1 GB to 476 GB. For each image, we apply all possible prefix length parameters and record the runtime for the timestamp carver and filesystem specific parser, and also record the number of potential timestamp hits.

We perform a location-based data recovery evaluation to test the performance of our tools' abilities to carve for filesystem metadata records, wherein we measure their precision and recall for extracting records from specific files or regions of the disk (such as from the \$MFT, \$LogFile, and the inode table). Note that we are running our tools on the entire disk images, as the tools are intended to be used, but we only calculate precision and recall for identifying filesystem metadata record hits for specific regions of disk, on a particular partition. We would have liked to obtain the precision and recall of our tools' ability to carve for filesystem metadata records across an entire disk image or partition, but since we did not create the test images we have no way of knowing the ground truth information regarding the locations of all filesystem metadata records on any partition. We additionally determine the files containing any hits for file system metadata records found outside these test spaces.

The prefix-based potential timestamp carver, the filesystem specific parsers, and instructions on how to use the tools are provided on a GitHub repository.⁴

The paper is organized as follows. The [Introduction](#) section has described the objectives and experiments of this paper and the [Related Work](#) section covers past work, such as prominent metadata carving methods and timestamp carving methods. The [Methodology](#) section describes our prefix matching potential timestamp carving algorithm, how it fits into the greater Generic Metadata Time Carving methodology, a description of the experiments, and a description of the location-based data recovery evaluation. The [Results](#) section covers the outcomes of our experiments. The [Discussion](#) section analyzes and synthesizes our results, as well as looks at the limitations of this study. Lastly, we conclude in the [Conclusion and Further Work](#) section.

³In short, precision is the percentage of returned hits that are relevant to the user's search, and recall is the percentage of the total amount of relevant items that were returned.

⁴Timestamp Prefix Carving for Filesystem Metadata Extraction code <https://github.com/TimestampPrefixCarving/Peer-Review>

C.2 Related Work

Metadata carving is a niche field in digital forensics, as opposed to file carving which is better studied, so we have attempted to cover the subject in full. In summary, these methods do not depend on critical filesystem data such as the \$MFT record, superblocks, or group descriptor tables. Carving for metadata is done in a byte-wise manner, and if the pointers in the metadata records to their files have not been deleted, then there is a possibility of full file recovery.

In Figure C.1, we show an abstraction of a disk image with two partitions, which also shows a simplified abstraction of filesystems. The image is a representation of how critical filesystem data such as the MFT table or superblocks keeps track of the filesystem metadata records. For details on traditional file carving, see the Forensics Wiki (forensicswiki.xyz 2012).

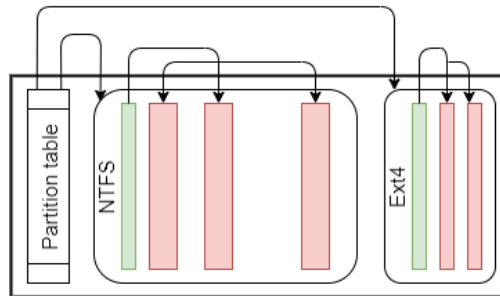


Figure C.1: An abstraction of a simple disk image, partitions, and filesystems. The large encompassing rectangle is the entire disk image, the furthest left rectangle with internal lines is the partition table that points to the partitions, and the other rectangles with rounded corners are partitions. Each partition has a filesystem, where the green rectangles represent filesystem critical data structures such as the \$MFT record (and its mirror), superblock, or group descriptor table. These help keep track of the filesystem records (for example, inodes or MFT records), which are represented by the red rectangles. *Generic Metadata Time Carving* (Nordvik et al. 2020), and our work, attempts to find the red blocks without help from the green blocks. For a more complete picture of the general filesystem structure, see the work by Carrier (Carrier 2005).

C.2.1 Metadata Carving

One of the first works that scientifically studied metadata carving was done by Dewald and Seufert (2017). They exclusively carve for inodes in Ext4, where they intentionally made their images' superblocks and group descriptor tables unusable. Their method of byte-wise search uses search patterns that are expected to be found in inodes such as file flags, extent header magic numbers, and tests the relationships between the timestamps ensuring their validity. In their experiments,

they tested a variety of combinations of the inode attributes to search for, where the effectiveness of the combination of patterns is dependent on the case. Similar work was performed by [Plum and Dewald \(2018\)](#), where they carved for nodes using different combinations of object type and subtype.

Our work extends the work by [Nordvik et al. \(2020\)](#), wherein they created the Generic Metadata Time Carving method. The method considers every non-overlapping m length sequence of bytes as a search keyword, where this keyword is checked for equivalency against every non-overlapping m length sequence of bytes in a k length search window directly following the keyword. If the keyword matches one or more of the sequences of bytes in the search window, then the location of the potential timestamp is recorded. One notable aspect of their byte-wise search approach to timestamps is that it does not require the user to set a minimum or maximum date they are searching for, only a guess as to what the length of the timestamp is and the filesystem that is suspected. For instance, it is required that the length m of the timestamp must be defined as either 4 or 8. After a list of potential timestamp locations are compiled, another scan over the disk image is performed that is filesystem specific. Using the list of potential timestamp locations, they directly access each location on the image and check if specific byte offsets relative to the timestamp location fit the profile of the metadata record being searched for. Examples of such expected features are Standard Information Attribute (SIA) or Filename Attribute (FNA) flags for NTFS, or the extent header magic number for Ext4. Once the metadata record is identified as a positive hit, the metadata information can easily be read out, including resident files for NTFS records, dataruns from NTFS Data Attributes, and extents and block pointers from Ext4.

The potential timestamp carver by [Nordvik et al. \(2020\)](#) works generally, but to date, their filesystem specific parsers only support NTFS and Ext4. For their experiments, they created disk images with a known set of files for each filesystem, where each filesystem was then damaged by reformatting the image with a different filesystem. Their results for the NTFS experiment achieved greater than 99% precision in identifying MFT records and full recall in retrieving files known to the original filesystem. For Ext4 they obtained 100% precision in identifying inodes, but only retrieved about 23% of the inodes known to the original filesystem. Their experiments however only included metadata records that had multiple timestamps that were exactly the same, thus our work intends to apply a prefix matching version of their approach to more realistic datasets.

The first notable reference to byte-wise timestamp carving appears to be from [Mc-cash \(2010\)](#), wherein he used an EnScript from [Mueller \(2008\)](#) to discover MFT records, indexes, and registry keys. The timestamp carving methodology essen-

tially allows the user to input a date or range of possible dates, the EnScript converts the dates into their NTFS byte format, and the possible byte sequences are then searched for. To improve the precision of the tool, there is an option to search for contiguous potential timestamps.

C.2.2 Related Methods of Data Retrieval

We briefly touch upon some tools that do not strictly use filesystem metadata extraction, but whose functionality is similar.

One tool that focuses on Ext4 file recovery via non-traditional means is Ext4Magic (Maar 2014). The basics of the tool is that it uses journal blocks with an old but functional deleted inode. The inode will hopefully point to datablocks which have not been reused for a different file.

Bulk Extractor by Garfinkel (2013) gathers relevant forensic features such as email addresses, phone numbers, credit card numbers, and more by parsing through a disk image in a single scan block by block. A benefit of the tool is that it truly is filesystem agnostic, and can analyze different parts of the disk in parallel.

C.3 Methodology

In this section we first describe our prefix-based potential timestamp carving algorithm and how it fits into Nordvik et al. (2020)'s Generic Metadata Time Carving workflow, and then describe our experimental and evaluative methodologies.

C.3.1 Prefix-Based Potential Timestamp Carving Algorithm

The prefix-based potential timestamp carving algorithm, like the exact matching version by Nordvik et al. (2020), is used to identify the byte offset locations of potential filesystem metadata record timestamps from across an entire disk image. The algorithm outputs the offsets from the beginning of the image to a text file. Our prefix-based potential timestamp carving algorithm adds unique elements to the timestamp carving algorithm and source code from Nordvik et al. (2020). We briefly review the original algorithm, as understanding their work is imperative for understanding our own contributions. Their basic assumption is that timestamps within filesystem metadata records are typically co-located close to each other, and often two or more timestamps are identical.⁵

The search procedure for these algorithms is based on the *sliding window* approach, where we have some byte-stream T representing the forensic image being searched, and we let m be the length of a timestamp. Potentially, almost every

⁵Filesystem metadata records often record their timestamps consecutively, and oftentimes actions on a file (creation, access, modification, etc.) update several of its timestamps simultaneously.

non-overlapping m bytes of the forensic image is tested as a *candidate timestamp*, and used as a *keyword*. This test requires a user defined value k , which is the length of bytes for a search window following each candidate timestamp. If the candidate timestamp passes the test, then it is considered to be a *potential timestamp*. The search begins at $T[0]$, with the first candidate timestamp being $T[0 : m - 1]$. This candidate timestamp is then compared to each non-overlapping m byte sequence within the k length window for equivalency. We refer to this process as the *timestamp equivalency test*, and these byte sequences as *test sequences*. If the number of matches is greater than or equal to the threshold $h - 1$, h being the number of required matching timestamps within a metadata record set by the user, then the position of the candidate timestamp (now potential timestamp) is recorded, the search skips ahead by k , and repeats the search procedure. The definitions given in this paragraph and the timestamp equivalency test are illustrated in Figure C.2. If the candidate timestamp is not found to be a potential timestamp, then the search only skips ahead by m , and the search procedure repeats. The search continues until the last k bytes of T . Full details of the algorithm can be found in Nordvik et al. (2020).

```

16 AC 9A 90 C2 FE CC 01 [44 E1 DD FD C2 FE CC 01]
44 E1 DD FD C2 FE CC 01 [A6 6F BA DA C3 FE CC 01]
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 00 90 00 00 00 00 00 00 00 00 00 03 00

```

Figure C.2: For 8 byte timestamps, the *candidate timestamp* is highlighted with green, and the *test sequences* are highlighted in blue. The search window is indicated by the brackets. The *timestamp equivalency test* simply checks how many times the candidate timestamp matches the test sequences. If the number of matches is greater than or equal to the threshold $h - 1$, where h is the number of required matching timestamps within a metadata record set by the user, the candidate timestamp is deemed a *potential timestamp*.

Our major contribution in this work is the modification of the timestamp equivalency test. In most cases, timestamps that are stringologically similar should also be temporally similar. Our implementation of the timestamp equivalency test simply tests if the p most significant bytes, which we refer to as the prefix, of a candidate timestamp is equivalent to the prefixes of the test sequences.

An example of such a search is shown in Figure C.3. The prefix of the timestamp (the most significant bytes) is least likely to change when a timestamp is updated, and typically holds information regarding the timestamp's month and year. We argue this form of prefix matching is more suitable as an approximation metric than other popular metrics such as the Hamming (Hamming 1950) or edit distance (Levenshtein 1966), since they do not consider the order in which the matching errors occurred. Furthermore, the method for prefix matching is algorithmically simple.

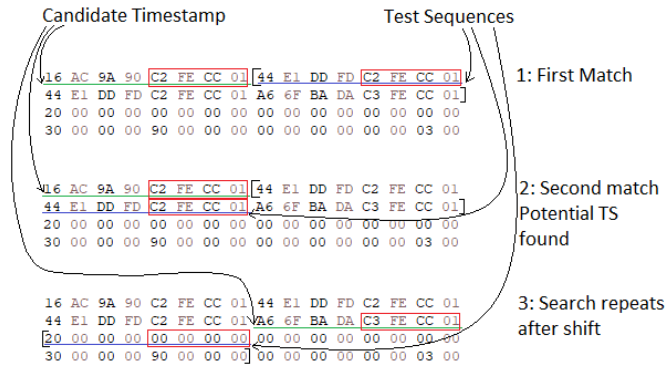


Figure C.3: Hex dump with highlights to illustrate the timestamp prefix matching search procedure. The byte sequence underlined in green represents the current candidate timestamp, and those underlined with blue are test sequences. The brackets represent the candidate timestamp’s search window. The red boxes represent the little-endian prefixes that are being compared for equivalency. The first two examples show matches, despite the fact the candidate timestamp does not equal the subsequent ones. If three matching timestamps are required ($h = 3$), the third example shows the advancement of the search by k bytes, and begins to repeat the entire procedure.

For testing if a candidate timestamp is a potential timestamp, the original algorithm by Nordvik et al. (2020) converted the candidate timestamp byte sequence and the test sequences to their big-endian forms for using them as unsigned long longs, and we do this as well. For most operating systems and filesystems, timestamps are recorded in little-endian, but utilizing them in a program in big-endian is generally more useful. In order to test whether the candidate timestamp and the test sequences have an equivalent prefix of length p (that the p most significant bytes of the timestamps are the same) we need only XOR the timestamps in their big-endian form, and shift the resulting value to the right by $8 * (m - p)$ bits. The shift to the right removes the $m - p$ least significant bytes from the result of the XOR, and if the remaining value is 0, then the prefixes must match. If the prefixes match, the count of matching timestamps for the candidate timestamp is increased by 1. Algorithm 2 explicitly describes this process. The prefix matching algorithm is only a component within the prefix-based potential timestamp carving algorithm (see Algorithm 2), which is shown in C.8.

We add one extra condition that must be met when applying prefix-based timestamp carving. The original method requires that candidate timestamps cannot be sequences of repeated bytes, as this filters out very common byte sequences such as 0x0000 and 0xFFFF. These common byte sequences would otherwise generate many false positive timestamp matches. We keep this condition, but add that a

Input: Big-endian unsigned long long forms of candidate timestamp x and test sequence y

Output: Number of matches for the candidate timestamp either increases or stays the same

m # Length of timestamp;

p # Size of prefix;

$xorResult = x \oplus y$;

if $((xorResult \gg (8*(m-p))) == 0)$ **then**

$matchCount += 1$;

end

Algorithm 2: Prefix matching algorithm.

candidate timestamp's most significant bytes cannot be 0. This is due to the fact that there are many byte sequences which in big-endian start with non-zero values, but end with zeros, which will cause our algorithm to identify many potential timestamps where the candidate timestamp is a non-zero byte sequence, but the prefixes being tested are not. We consider this to be a fair assumption to make, as most timestamps' most significant bytes are non-zero. Both conditions can be found in Algorithm 2 within C.8.

Since the modification of the potential timestamp carving algorithm by Nordvik et al. (2020) primarily only changes the timestamp equivalency test by a constant number of steps, it implies that the time complexity of the prefix-based potential timestamp carving algorithm must remain the same. While Nordvik et al. showed that a general potential timestamp carver would run in nonlinear time (their worst case scenario omits the repeated byte timestamp check and conversion from a string of characters to a 64-bit data type), the implementation of the potential timestamp algorithm effectively runs in linear time, dependent on the length $|T|$ of the disk image. This is because length m is limited to a choice of 4 and 8 (timestamps are stored in an unsigned long long type), and because the search window length k in most practical situations would never exceed the size of a block or cluster.

Note, the potential timestamp carving algorithm assumes that timestamps fall on 4 or 8 byte boundaries, so metadata that is unaligned will be missed. We also note that if timestamps are recorded in big-endian, prefix-matching will not work.

C.3.2 Generic Metadata Time Carving and the Filesystem Specific Parsers

The potential timestamp carving algorithm described in the previous section is only the first step of the Generic Metadata Timestamp Carving (GMTC) methodology. The produced list of potential timestamp locations in general will be extremely

large, and thus referring to many “false positive timestamps” identified across the full disk image. In the GMTC method, the list of potential timestamp locations is then fed into a filesystem specific parser, and the parser checks the offsets on the disk image given by the potential timestamp list in an attempt to verify if the potential timestamp is contained within a filesystem metadata record. The output of the parser is a .csv and .txt file containing data from the suspected records. More or less, a filesystem specific parser acts as a filter. We briefly describe the strict verification tests performed by the NTFS and Ext4 parsers. The majority of stated parser tests were in the original work by [Nordvik et al. \(2020\)](#)

The NTFS parser first checks if the date of a potential timestamp falls within the year range of 1970 and 2100. If so, it then checks all possible offsets behind the potential timestamp location for Standard Information Attribute (SIA) or Filename Attribute (FNA) attribute header flags. If one of these flags are found, we make an assumption of where the attribute begins. We can use the identified attribute lengths to navigate from one attribute to the next, where we require to start from the SIA, then hop to an FNA, and then if possible the Data attribute. Furthermore, the MFT attributes encountered must be in numerical order (as given by their header flags). Any attributes encountered for an assumed record must occur within a 1024 byte space. The parser also only reports an MFT record if the identified filetype extracted from an FNA is one of four possible types. The accepted filetypes are: "File", "Directory", "Index View", or "Directory and Index View". More information regarding MFT data structures can be found in the work by [Carrier \(2005\)](#).

The Ext4 parser is more complicated due to the fact that inodes are small, have far less features to strictly identify than MFT records, and that the parser attempts to connect the inodes to a filename and inode number. Both the inode number and filename are in a different data structure than the inode. The first step for validating potential timestamps is to check the possible offsets from the potential timestamp to the filetype nibble at the start of the inode. We only allow for three different types: "Regular Files", "Directories", and "Symbolic Links". The offset to one of these values dictates our guess to where the inode begins. If the extent flag is set, and the offset to the extent header magic number is 0xF30A, or if there is no extent flag and the offsets from the beginning of the inode 0x24 to 0x27 are 0, we continue our validation tests. The total size of the file is checked to see if it corresponds to the total amount of blocks it is occupying, if the total size of the file is less than the size of the image, and if the relationships between the timestamps are valid. For instance, we check if the deleted value is not 0, then it must be greater or equal to both the modified and created time. The steps so far are the validation checks done in the preprocessing phase, as we need to gather information to try

to connect inodes to their filename and inode numbers. If the inode passes the initial preprocessing and validity checks, it is fully processed. The timestamps are checked to ensure they fall within the years 2000 and 2020 (the deletion timestamp being the exception). For more information about Ext4 data structures and inodes, see [Ext4 \(and Ext2/Ext3\) Wiki \(2019\)](#).

C.3.3 Experimental Methodology

We use our prefix-based Generic Metadata Time Carving (GMTC) method on three realistic forensic images. We first apply our novel prefix-based potential timestamp carving algorithm on the images, where the output of the algorithm creates a text file list of the locations of the potential timestamps in byte offsets from the beginning of the image. This list is then input into one of the two pre-existing but modified filesystem specific parsers (NTFS or Ext4), where the output of the parser is a .csv and .txt file with data from the discovered metadata records. We identified a few bugs in the original filesystem specific parser scripts by [Nordvik et al. \(2020\)](#), so we updated them so that we may achieve more complete and accurate results.

The three images being tested are a 1 GB NTFS image, one 59.5 GB Ext4 image from a real device, and one 476 GB synthetic NTFS image. The small NTFS image is from NIST's Deleted File Recovery page (DFR-13) ([NIST 2017](#)), the Ext4 image was extracted by the authors from a real Samsung S8 mobile phone, and the large NTFS image is the "Lone Wolf" forensic image, available from Digital Corpora ([Moore et al. 2018](#)). Notably, these images are not guaranteed to have at least two equivalent timestamps per metadata record, unlike the work by [Nordvik et al. \(2020\)](#).

For each image we try all possible sizes, p , of the prefixes of the candidate timestamps that are required to be equivalent to the prefixes of the test sequences. Since it is possible that the prefix of the candidate timestamp can be the length of the timestamp itself, we are also comparing the precision-recall performance of the original GMTC method to our method.

We clarify some items regarding our testing and evaluation methods. The prefix-based GMTC methodology (the potential timestamp carving followed by the filesystem specific parser) is applied to the entire disk image for our tests. Thus we obtain potential timestamp locations and filesystem metadata records from across entire disk images. However, since we have no ground truth information regarding the number and location of all file system metadata records across the disks, we cannot evaluate the precision and recall of our tools across an entire disk. Thus, we limit our precision and recall evaluations to specific areas or files on the disk, where we

can easily retrieve the number and location of records. Examples of such files or regions of disk include the MFT table or inode table for a particular partition. This is explained more in depth in the next subsection.

Other data we record are the number of potential timestamps logged by the prefix-based potential timestamp carving algorithm, the time required by this algorithm and the filesystem specific parsers, and the number of metadata records extracted that were outside the \$MFT, \$LogFile, or inode table and which file they were found in.

C.3.4 Precision-Recall Location-Based Data Recovery Evaluation

Ideally, our experiments would measure the precision and recall of our tools' ability to carve all filesystem metadata records from an image or partition, but this is infeasible since we have no reliable method of obtaining ground truth knowledge of every single offset of every single file system metadata record. Thus, while we still run our tool on entire disk images, we focus on our tools' precision and recall for carving filesystem metadata records from specific files or regions of disk where record offsets are more easily obtainable. We also determine the files on the same partition that contain the hits for file system metadata outside the precision-recall evaluated files or data structures.

For NTFS, we measure the precision and recall for carving MFT records from the \$MFT of the partition of interest, and we perform another precision and recall evaluation for carving MFT records from the \$LogFile. To obtain ground truth knowledge of the offsets to MFT records in each file, we used The Sleuthkit to export these files from the NTFS image with `icat`, search the file for FILE signatures, and check if the 0x38 byte offset from the FILE signature equals the Standard Information Attribute flag (0x10000000). For the \$LogFile, we also check the 0x78 byte offset from the FILE signature for the cases that an MFT record is divided between \$LogFile pages, where the beginning of each page has header that begins with the signature "RCRD". Using The Sleuthkit's `istat` command we also obtained the clusters that the \$MFT and \$LogFile occupy, so that we can translate the files' logical offsets to MFT records into physical offsets on the disk. All the discovered offsets for MFT records in the \$MFT matched possible locations of records given by the clusters output from the `istat` command on the \$MFT.

We refer to these ground truth offsets to MFT records as *Condition Positives*. Formally, a Condition Positive is the knowledge that at address A , there exists a filesystem metadata record. It is *conditioned* on the fact that we are limiting our precision-recall evaluations to the regions of the disk occupied by a specific file or ranges of disk space. By running our prefix-based GMTC method on the entire

disk image, we obtain a large set of byte offset locations that our tools detect as the locations of filesystem metadata records.⁶ We refer to the offsets identified by our tools as *Test Positives*. One can think of our filesystem specific parsers similar to that of a classifier when they filter potential timestamp locations, which declares at address A we detect a filesystem metadata record of minimum length L . The value L for inodes and MFT records is 256 bytes (the minimum length of an MFT record includes the record header of length 0x38, Standard Information Attribute of length 0x60, and a minimum length Filename Attribute of 0x68). A Test Negative is simply that our tools do not detect a filesystem metadata record at address A .

For Ext4, we measure the precision and recall for carving inodes from the inode table of the partition of interest. To determine the Condition Positives in the inode table, we use the Sleuthkit's `fls -r` command to dump all files to a list (wherein we add the root directory and Journal with inode numbers 2 and 8 respectively). Using The Sleuthkit's `fsstat` command, we determine which blocks the inode table occupies, and which inodes are in which fragment of the inode table. Using this information (inode numbers provide a 256 byte multiple offset into their respective inode table fragment), we can calculate the physical positions of each inode offset from the beginning of the disk. We would have liked to perform similar tests on the Ext4 Journal, but we would need a more certain method of identifying Condition Positives other than performing a string search for the extent signature 0xF30A.

We reiterate that we limit the Condition Positives to the regions of disk where the precision and recall is being measured. If a Test Positive is also a Condition Positive, then the result produced by the GMTC tools is a true positive. That is, our tools detected a filesystem metadata record at address A with minimum length L , and the beginning of a record truly begins at address A . A false positive occurs if we obtain a Test Positive at some address B within the region of disk under examination, where according to our list of Condition Positives no record exists. If there are Condition Positive addresses that do not have a matching Test Positive address, then our tools have produced a false negative, a miss.

We use the typical precision and recall measures for our analysis, as seen in the equations below.

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

⁶The Ext4 parser reports the byte offsets to the beginning of the inode, whereas the NTFS tool reports the byte offsets to the potential timestamp identified by the potential timestamp carver. Thus for MFT records, we have to consider the set of all possible locations of the beginning of the record with respect to the identified Standard Information Attribute timestamp.

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

When calculating precision and recall, it is possible that after accounting for all the Test Positives located in the disk image regions such as the \$MFT, \$LogFile, or inode table that there may still be a large number of Test Positive hits that are still unaccounted for elsewhere on a partition. To find where these extra records come from, we use The Sleuthkit’s `istat` command to list the blocks/clusters (we refer to these as “blocks” from here on out) allocated to files known to the inode or MFT table, where the files’ records are filtered with respect to our calculated Condition Positives. For each list of blocks extracted from the `istat` output, we create a list of block ranges that a file occupies, which also accounts for fragmentation. We then build a Python dictionary of such values where the key is the record number, and the values associated with a key are the block ranges of the file, and the file’s name. It is then possible to create a derived version of this dictionary, where the key is a starting block of a particular file fragment, and the values associated with the key are the ending block of the file fragment, as well as the file’s name and number. When this dictionary is ordered numerically, and our Test Positives are ordered by their offsets numerically, we can quickly search through all the file fragments to identify where our remaining hits lie.

C.3.5 Specifics of NTFS Experiments

The 1 GB NTFS image is the 13th test case (`dfr-13-ntfs.dd`) from NIST’s Deleted File Recovery page (NIST 2017). This test case has performed random filesystem activity, so the timestamps of the MFT entries are rarely all equal. When running our prefix-based timestamp carving algorithm we set the length of the timestamps $m = 8$, the search window $k = 24$, and the required number of matching timestamps to $h = 3$ (the same parameters used by Nordvik et al. (2020)). We carved for timestamps for all possible prefixes p , from 1 to 8.

For this image, we only performed the location-based data recovery evaluation on the \$MFT of the partition starting at sector 128, as we did not identify any MFT records in the \$LogFile. We verified the lack of full MFT records in the \$LogFile by running the `LogFileParser` by Schicht⁷ on the file. Transactions in the LogFile where the Redo Operation or Undo Operation has the status of “InitializeFileRecordSegment”, and the other Redo or Undo Operation has the status of “Noop” indicates that the transaction contains an entire MFT record (Cowen and Seyer 2013). We found no such transactions.

⁷<https://github.com/jschicht/LogFileParser>

The experiment using the 476 GB Lone Wolf forensic image (available from Digital Corpora (Moore et al. 2018)) focuses on the “Basic Data Partition” for the precision and recall evaluations, the largest partition on disk. Our timestamp carving experiments for the Lone Wolf image use the same parameters as the DFR-13 image.

We performed the location-based data recovery evaluation on the \$MFT and the \$LogFile on the LoneWolf image’s partition.

C.3.6 Specifics of Ext4 Experiments

The Ext4 experiment uses a dump of a Samsung S8 mobile phone running Android, where we specifically focus on the “SYSTEM” partition’s inode table for the precision and recall calculations. The User partition was encrypted, and SYSTEM partition was the second largest partition on the image. The image was created by first flashing the recovery partition using the TWRP Recovery image (TWRP 2019), and then using an ADB bridge executing a combination of netcat and dd commands in order to acquire the raw image. The recovery image method is described in detail by Son et al. (2013) and Vidas et al. (2011). We ran our prefix-based timestamp carving algorithm on the image with the same parameters as those used by Nordvik et al. (2020), where the length of the timestamps m was set to 4, the search window $k = 12$, and the required number of matching timestamps to $h = 2$. We carved for timestamps for all possible prefixes p , from 1 to 4.

The Ext4 parser also requires a few additional parameters, which are assumptions that assist in attempting to connect inodes to their filename and inode number. Using the Sleuthkit, we obtained the blocksize of 4096 bytes (which is the default blocksize (Ext4 (and Ext2/Ext3) Wiki 2019)), and the byte offset of 225968128 to the partition. Thus, the parser only examines the disk from this offset onward.

C.3.7 Computer Specifications

A Mac with the following specifications was used to run the timing experiments.

- OS: MacOS Catalina v 10.15.4
- Processor: 4.2 GHz Quad-Core Intel Core i7
- Memory: 64 GB 2400 MHz DDR4
- Storage: APPLE SSD SM0128L 3.12 TB, PCI-express, a hybrid, where 128GB is pure SSD, and 3 TB is SATA. Sequential Read: 952 MB/s, sequential write 57 MB/s. Random read 0.9 MB/s, and random write 50 MB/s.

While running the tools we did not activate any other resource demanding processes. However, it is always possible that the OS performed additional scheduled tasks. We used the tool DiskMark⁸ v2.2 to measure the read/write speed.

C.4 Results

Overall, our results show that by reducing the size of the prefix p of a timestamp in the timestamp equivalency test, a much higher recall for filesystem metadata record extraction can be achieved using the Generic Metadata Time Carving (GMTC) method as compared to the exact timestamp matching approach. To our surprise, the precision of the metadata extraction was not reduced by decreasing the size of the matching prefix p , and remained at 100% for all experiments. We go through each disk image we tested, showing the results of the individual precision-recall tests over specific areas of the disk, and the timing results for the potential timestamp carver and parser for that particular image. All timing experiments were run twice, and the listed runtimes are their averages. We also describe the files on the evaluated partition that contained filesystem metadata records that were outside the \$MFT, \$LogFile, and inode table.

C.4.1 Small NTFS Image

The results for carving MFT records from the \$MFT from the 1 GB NTFS image's partition beginning at sector 128, as seen in Table C.1, show that applying prefix matching of timestamps greatly increases the recall, and appears to maintain the exact matching Generic Metadata Time Carving method's 100% precision. The exact matching GMTC ($p = 8$) only obtained 8.8% recall for finding MFT records, whereas decreasing p to 3 and less achieved a 97.9% recall. The number of Test Positives identified over the entire partition for different values of p is shown in Table C.2. The true positives account for most of the Test Positives found over the entire partition, but three had gone unaccounted for. It transpired they were the \$MFTMirr, \$LogFile, and \$Volume records found in the \$MFTMirr file.

This increase in recall was not without its trade-offs, as seen in Table C.3. Upon decreasing p from 8 to 4, the number of identified potential timestamp locations increased by three magnitudes. While this did not appear to unduly influence the timestamp carving algorithm, the time required for the filesystem parser increased more than 100 fold.

C.4.2 Ext4 Samsung S8 Image

The results for carving inodes from the Ext4 image's SYSTEM partition's inode table are shown in Table C.4. Like the small NTFS image, we achieved 100%

⁸<https://inchwest.com/diskmark/>

p	True Positives	False Positives	False Negatives	Precision	Recall
8	21	0	218	1	0.088
7	21	0	218	1	0.088
6	126	0	113	1	0.527
5	164	0	75	1	0.686
4	219	0	20	1	0.916
3	234	0	5	1	0.979
2	234	0	5	1	0.979
1	234	0	5	1	0.979

Table C.1: Precision and recall for carving MFT records from the \$MFT from the 1 GB NTFS image's partition beginning at sector 128 with $p = 1, 2, \dots, 8$. The \$MFT had 239 Condition Positives.

p	8	7	6	5	4	3	2	1
Test Pos. Count	24	24	129	167	222	237	237	237

Table C.2: Test Positive count over entire partition from the 1 GB NTFS image, where $p = 1, 2, \dots, 8$.

precision in identifying inodes, where the recall increased for carving inodes from the inode table as the prefix length value of p decreased. However, the increase in recall was quite minor, only increasing by about 3%. We discuss our theories of why the precision and recall were so high for the Ext4 experiment in the Discussion section.

Table C.5 shows the Test Positive counts of detected inodes found over the entire partition, with respect to the prefix length p being used. All test positive hits that were not discovered in the inode table were discovered in the Journal where, when the timestamp prefix length $p = 1$, we detected 1924 inodes.

In terms of computational performance, Table C.6 exposed trends regarding the timestamp carving program when working with large files. Larger prefixes p caused the timestamp carver to take longer to complete, but not by too much. Unlike the small NTFS image experiment, the number of potential timestamp locations only increased by about one magnitude going from $p = 4$ to $p = 1$. The time required to run the filesystem parser appears to have an approximately linear relationship between the number of potential timestamp carving locations, as the time required to run at $p = 1$ is about 10 times as slow as using a prefix size of $p = 4$.

p	# PTS Locations	TS Carve Time (s)	Parser Time (s)	Total Time (s)
8	892	8.177	0.049	8.226
7	3143	8.132	0.082	8.214
6	4311	8.128	0.104	8.232
5	3638	8.131	0.106	8.237
4	2056322	8.451	6.59	15.042
3	2056629	8.413	6.689	15.102
2	2056636	8.459	6.659	15.117
1	2061768	8.4	6.622	15.019

Table C.3: Generic Metadata Time Carving performance for the entire 1 GB NTFS image with $p = 1, 2, \dots, 8$. PTS stands for “Potential Timestamp”.

p	True Positives	False Positives	False Negatives	Precision	Recall
4	6766	0	670	1	0.910
3	6766	0	670	1	0.910
2	7004	0	432	1	0.942
1	7004	0	432	1	0.942

Table C.4: Precision and recall for carving inodes from the inode table from the SYSTEM partition in the 59.5 GB Ext4 Samsung S8 image with $p = 1, 2, 3, 4$. The inode table had 7436 Condition Positives.

C.4.3 Large NTFS Image

The results for carving MFT records from the \$MFT and \$LogFile from the Lone-Wolf image’s Basic Data Partition are seen in Table C.7 and Table C.8 respectively. Again, we achieved 100% precision in identifying MFT records, both for the \$MFT and \$LogFile (we encountered no false positives with respect to our Condition Positive lists). The recall results reflect previous trends. Using exact matching timestamp carving we only achieved 41.6% recall for carving MFT records from the \$MFT, and allowing for smaller timestamp prefix matching caused increasingly higher recall. The point of diminishing returns appeared to have occurred at $p = 2$, where about 97.2% recall was achieved. The recall results are quite different for carving MFT records from the \$LogFile, as the recall hovered around 87% despite the value of p .

Table C.9 shows the total number of test positives found over the entire partition, and after filtering out the Test Positive hits found in the \$MFT and \$LogFile, there were still a large number of hits left unaccounted for. When discussing where these hits were found on the partition, we focus on the results for $p = 1$, since each value of p larger than this should be a subset of the $p = 1$ results. In total, there were

p	4	3	2	1
Test Pos. Count	8470	8470	8928	8928

Table C.5: Test Positive count over entire SYSTEM partition from the Ext4 Samsung S8 image, where $p = 1, 2, 3, 4$.

p	# PTS Locations	TS Carve Time (s)	Parser Time (s)	Total Time (m:s)
4	10630945	1401.78	45.73	24:07.51
3	26228387	1397.36	115.25	25:12.60
2	41610448	1369.78	186.43	25:56.21
1	97555603	1266.81	452.42	28:39.22

Table C.6: Generic Metadata Time Carving performance for the entire 59.5 GB Samsung S8 image with $p = 1, 2, 3, 4$. PTS stands for “Potential Timestamp”, m for minutes, and s for seconds. Note, the Ext4 parser skips the first approximately 210 MB.

91157 test positive hits that were yet to be accounted for. Using the dictionary we created that contained the allocated cluster ranges of all known files on the partition, we were able to discover where these potential MFT records were coming from, as seen in Table C.10. The \$MFTMirr contained the usual records of \$MFT, \$MFTMirr, \$LogFile, \$Volume. Four different boot.sdi files (with the filenames "boot.sdi, boot.sdi") each contained 42 filesystem metadata record hits, where a boot.sdi file is essentially a small partition of its own with completely irrelevant MFT records. It is used as a Ramdisk which can be shown with the *bcddedit* command (Active KillDisk 2021). Lastly, we have the two Volume Shadow Copies⁹ that contained 90985 detected MFT records.

In terms of timing performance, the large NTFS experiment mostly behaved as expected (see Table C.11). Like in the Ext4 timestamp carving experiment, the run-times for all values of p were similar, but experiments with lower values of p took less time. What was rather surprising was the relatively small increase in potential timestamp locations that were found by $p = 1$ versus $p = 8$, given the size of the image. The increase was only by a factor of about 4.77, quite a deal

⁹Using The Sleuthkit’s (versions 4.4.1 and 4.10.1 tested) *istat* command for Volume Shadow Copies (VSC) will show that the file only occupies a single cluster, having a large non-zero size, and an *init_size* of 0. This error has been seen before: <https://github.com/sleuthkit/sleuthkit/issues/466>. Why we bring this up is that relying on the Python dictionary we created for cluster ranges of files will be incorrect for the VSCs. To address this, we used the given cluster as the start of a VSC’s range, and added the size of the file to obtain the end of its range. To ensure this unfragmented region of disk was truly a VSC file, we performed the following. *icat -s* will output a VSC entirely, and we took the MD5 hash of the VSC files. We then took MD5 hashes of the unfragmented disk regions defined by the byte ranges we were using for the VSCs. The hashes of the files and the regions of disk were identical.

p	True Positives	False Positives	False Negatives	Precision	Recall
8	59422	0	83538	1	0.416
7	59422	0	83538	1	0.416
6	72284	0	70676	1	0.506
5	95193	0	47767	1	0.666
4	120482	0	22478	1	0.843
3	129220	0	13740	1	0.904
2	139022	0	3938	1	0.972
1	139082	0	3878	1	0.973

Table C.7: Precision and recall for carving MFT records from the \$MFT of the Basic Data Partition from the 476 GB LoneWolf NTFS image with $p = 1, 2, \dots, 8$. The \$MFT had 142960 Condition Positives.

p	True Positives	False Positives	False Negatives	Precision	Recall
8	2251	0	353	1	0.864
7	2251	0	353	1	0.864
6	2251	0	353	1	0.864
5	2251	0	353	1	0.864
4	2263	0	341	1	0.869
3	2263	0	341	1	0.869
2	2267	0	337	1	0.871
1	2267	0	337	1	0.871

Table C.8: Precision and recall for carving MFT records from the \$LogFile of the Basic Data Partition from the 476 GB LoneWolf NTFS image with $p = 1, 2, \dots, 8$. The \$LogFile had 2604 Condition Positives.

less than the increase of magnitudes we saw before. A possible reason for this is that the Lone Wolf image is a synthetic image that was only being used for some months. Stranger still, were the parser times over all possible values of p . Given the previous results, we should have seen parser times drastically increase as p decreased. This did not happen, as seen by the fact that the parsing time for $p = 1$ was on average less than most other values of p , and this is despite the fact that the experiment for $p = 1$ had about 46 million more potential timestamps to check than the $p = 8$ experiment. Since both runs of the parser produced such similar results, at the moment we can only guess that some aspect of the parser script handles things inefficiently. A major difference between the NTFS parser and the Ext4 parser is that the Ext4 parser uses a Python memory mapping library¹⁰ to

¹⁰<https://docs.python.org/3/library/mmap.html>

p	8	7	6	5	4	3	2	1
Test Pos. Count	108852	108852	134227	169588	204467	218559	232425	232506

Table C.9: Test Positive count over entire Basic Data Partition from the large LoneWolf NTFS image, where $p = 1, 2, \dots, 8$.

File	Record Number	Test Positive Count
\$MFTMirr	1	4
boot.sdi	21992	42
boot.sdi	21993	42
boot.sdi	21994	42
boot.sdi	21995	42
Volume Shadow Copy 1	96066	51795
Volume Shadow Copy 2	123530	39190

Table C.10: Files containing the remaining Test Positives not found in the \$MFT or \$Log-File of the Basic Data Partition, where $p = 1$. The number associated to each file indicates how many MFT records were found in that particular file.

handle the parsing of large files, while the NTFS parser has handcrafted code to handle large files.

We note that our tools found no Test Positives (detected hits of filesystem metadata records) in unallocated space for any of the disk images.

C.5 Discussion

Here we analyze our results, consider why we may have missed extracting some metadata records, the limitations of our research, and finally answer our research questions.

C.5.1 Analysis: Small NTFS Image

The small image from NIST ([NIST 2017](#)) purposefully created chaotic actions on the system, thus creating MFT records with erratic timestamps. The missed MFT records from the MFT table when $p = 1$ were the \$MFT, as the Standard Information Attribute timestamps were 0, and 4 other records that did not contain File Name Attributes. The NTFS parser requires a File Name Attribute to be present.

The only other interesting item to note is explaining why the number of potential timestamp locations jumped drastically from $p = 5$ to $p = 4$. The *dfr-13-ntfs.dd* image fills sectors not occupied by an MFT entry with repeated byte sequences of

p	# PTS Locations	TS Carve Time (s)	Parser Time (s)	Total Time (hr:m:s)
8	12235330	7324.28	1761.83	2:31:26.11
7	17426880	7322.61	2177.43	2:38:20.04
6	23192352	7291.95	2793.94	2:48:05.89
5	30135982	7279.97	2266.49	2:39:06.45
4	32353209	7286.00	2218.84	2:38:24.83
3	33625310	7296.31	2596.24	2:44:52.55
2	46791325	7298.88	1617.81	2:28:36.68
1	57934625	7137.19	1707.40	2:27:24.59

Table C.11: Generic Metadata Time Carving performance for the entire 476 GB NTFS image with $p = 1, 2, \dots, 8$. PTS stands for “Potential Timestamp”, *hr* for hours, *m* for minutes, and *s* for seconds.

either 0x2A or 0x5A, and the beginning of each sector has a message describing how the sector is or is not used. The combination of this message and the repeated byte sequences creates a large occurrence of valid potential timestamps. Such a situation would be unusual for more realistic images.

C.5.2 Analysis: Ext4 Samsung S8 Image

The location-based data recovery evaluation for carving inodes from the inode table of the Ext4 Samsung S8 image performed suspiciously well, having 100% precision and and 91% or greater recall. The high recall for extracting inodes from the inode table may indicate that the SYSTEM partition had fairly static files. Again, we would have liked to run the tests on the User partition, which would have included real user behavior, but it was encrypted.

The 432 false negative inodes from the inode table were entirely comprised of Symbolic Links. While the GMTTC method by [Nordvik et al. \(2020\)](#) and our work is said to consider symbolic links (and will catch some), the Ext4 parser always assumes that at offset 0x28 from the start of the inode will be direct blocks or the start of the extents. However, this is an incorrect assumption, as a symbolic link will be stored at this offset if the string is less than 60 bytes long ([Ext4 \(and Ext2/Ext3\) Wiki 2019](#)).¹¹

C.5.3 Analysis: Large NTFS Image

Other than the strange runtimes of the NTFS parser, the results from the experiments on the large NTFS image were in line with what we had seen in the previous images. Three items of interest are worth discussing: The high recall of MFT records carved from the \$LogFile, false negative MFT records, and Test Positives

¹¹https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Symbolic_Links

found outside the \$MFT and \$LogFile.

The high recall of at least 86.4% of MFT records found in the \$LogFile can be attributed to the fact that full MFT records only occur in \$LogFile transactions with “InitializeFileRecordSegment” operations, which means a new file is being created (Schicht 2018). When a new file is created, all the timestamps for the Standard Information Attribute (SIA) are updated (Knutson and Carbone 2016). This implies all the timestamps for the MFT records should be the same or nearly the same. Decreasing the timestamp prefix length p from 8 to 1 increased recall by less than 1%, which shows that some of the timestamps were indeed slightly different. The implications of these results is that the exact matching GMTC method will work well for carving MFT records from the \$LogFile. However, as Nordvik et al. (2020) previously observed, the majority of records recovered from the \$LogFile contained no datarun information, where we only identified 11 records that did.

It would appear that most of the 3878 false negative MFT records in the \$MFT were those that needed to have non-resident attributes. This was expected, as the NTFS parser does not handle MFT records that are larger than 1024 bytes. Most of the 337 false negative MFT records in the \$LogFile were records that crossed log pages, where a log page header (containing the magic number 'RCRD') split the MFT record somewhere after the Standard Information Attribute. We do however still find MFT records where they are split by a log page header after the MFT record header, and before the Standard Information Attribute.

Of the 91157 Test Positive hits for MFT records that were found in neither the \$MFT or the \$LogFile (see Table C.10), the 90985 hits in Volume Shadow Copies are the most interesting. This is because the Volume Shadow Copies are snapshots of previous states of the partition, and thus may either contain previous states of files and their MFT records, or they may contain MFT records that are now deleted. Furthermore, a large number of Test Positive hits outside the MFT table or LogFile, but within specific regions of disk, may indicate that Volume Shadow Copies even exist on a partition in the first place.

C.5.4 Limitations

The purpose of this work was to show that the GMTC method can be used on realistic images and timestamps, and that the use of a timestamp prefix matching method could greatly improve the method’s ability to extract filesystem metadata records. Here, we address the issues we believe to be the primary limitations of our work.

The first issue is that we are applying the GMTC method to data that does not fit

its more typical use-case. The GMTC method should be applied if the filesystem is damaged or otherwise inaccessible. We are applying the method to perfectly working images, and undamaged filesystems. The reason for this is to understand what the prefix-based GMTC method can *potentially* recover.

Another limitation of this work is that we were not using a user partition for the Ext4 experiments, so the filesystem we were analyzing was likely more static and not as realistic as we would have liked it to be.

The last large limitation of our work is our experiments' unsatisfying explanation of the unflagging 100% precision. However, by looking at our results, we can see that low timestamp prefix lengths do indeed produce many more false positive potential timestamps. For example, reducing the prefix length p from 4 to 1 in the Ext4 experiment increased the number of potential timestamps from approximately 11 million to 98 million. For the Large NTFS experiment, reducing p from 8 to 1 increased the number of potential timestamps from about 12 million to 58 million. The effect of prefix length p on the number of potential timestamps identified for the large NTFS and Ext4 images is shown in Figures C.4 and C.5 respectively, where we also show the number of Condition Positives to illustrate that the count of Condition Positives is only a fraction of the the number of false positive timestamps we may be encountering. According to our filesystem specific parsing experiments, it would seem that the filesystem specific parsers are extremely strict since we encountered no false positive records despite checking for millions of more offsets on the disk image. Likewise, it appears that the roll of the potential timestamp carver is to control the total number of byte offsets that a filesystem specific parser must verify when looking through a disk image for filesystem metadata records (affecting recall), and that the filesystem specific parser ultimately controls the precision of the GMTC method.

We wanted to observe these suspected rolls of the prefix-based potential timestamp carver and filesystem specific parsers empirically, so we conducted a short experiment. This experiment obtains the results of performing the prefix-based GMTC method on an encrypted image, as the data is essentially a large string of random bytes, and also obtain the results of applying the filesystem specific parsers directly on the encrypted image without first performing timestamp carving. Results we were interested in included the number of false positive filesystem metadata records the experiments would encounter, and how long the runtimes for the different experiments were. Our hypothesis was that we would not encounter any false positive records for any experiment, and that Generic Metadata Time Carving should be faster than applying the parsers directly on the image.

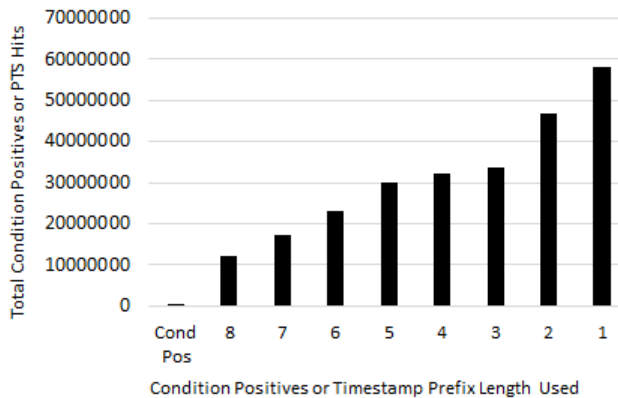


Figure C.4: Histogram comparing the number of Condition Positives we account for on the Basic Data Partition of the 476 GB Lone Wolf image and the number of potential timestamp (PTS) locations identified after carving for all possible prefix lengths.

We encrypted the 59.5 GB Ext4 image with Kleopatra¹² and carved for potential timestamps on the encrypted image using the same parameters as our previous experiments, but only searched for timestamps based on a prefix size of 1 byte. Then both filesystem specific parsers were ran on the encrypted image using their respective potential timestamp locations from the potential timestamp carving. Next, we ran the NTFS and Ext4 parsers over every byte of the encrypted image without using potential timestamp information, with the exception of the first and last 1024 bytes.

For searching for NTFS MFT records, we obtained no false positives for the GMTC experiment or the pure parser experiment. Carving for potential timestamps took approximately 17 minutes, and where 360990 potential timestamps were discovered. Applying the NTFS specific parser on the image with the potential timestamp results took about 7 minutes to run. Applying the NTFS parser directly on the encrypted image took 5.75 hours.

When searching for inodes, we encountered no false positives for the GMTC experiment or the pure parser experiment. The potential timestamp carving took about 17.5 minutes, and we identified 182405692 potential timestamps. When applying the Ext4 specific parser on the image with the potential timestamp results, the parser ran for about 17.5 minutes. Applying the Ext4 parser directly on the encrypted image took about 17.5 hours.

¹²<https://www.openpgp.org/software/kleopatra/>

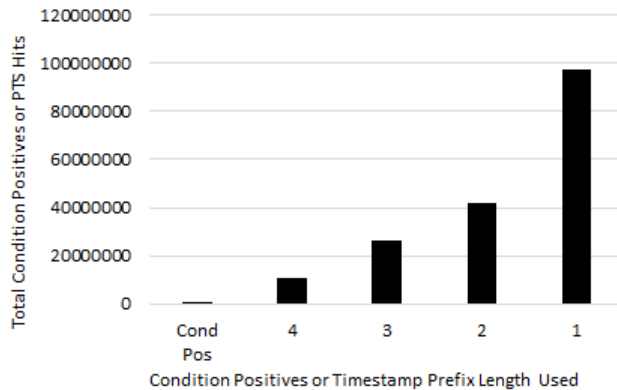


Figure C.5: Histogram comparing the number of Condition Positives we account for on the SYSTEM partition of the 59.5 GB Samsung S8 image and the number of potential timestamp (PTS) locations identified after carving for all possible prefix lengths.

These results further reveal why it is the GMTC method produces little to no false positives. The tests where the parsers are directly executed on the encrypted image show that it is the file system specific parsers that ultimately control the precision of the GMTC method since no false positives were encountered. This implies that the filesystem specific parsers are extremely strict when verifying filesystem metadata records, and that the records themselves are highly structured. However, running the parsers directly on the encrypted image took much longer to run.

With these results we get a clearer picture on how potential timestamp carving effectively acts as a data reduction technique, where its parameters influence the number of potential timestamps returned, going on to influence recall and parser runtime, and that the filesystem specific parsers ultimately control the precision of the GMTC method. We can also see that the other parameters for the potential timestamp carver such as the user defined threshold h of the required number of matching timestamps per record also controls the number of returned potential timestamps. For example, despite both applying a prefix length $p = 1$, using the NTFS timestamp carving settings (requiring $h = 3$ matching timestamp prefixes) only encountered 360990 potential timestamps, whereas carving with the Ext4 settings (requiring $h = 2$ matching timestamp prefixes) encountered 182405692 potential timestamps. However, more research needs to be done to understand all the implications of applying different potential timestamp carving parameters.

C.5.5 Revisiting Research Questions

Below, we answer our research questions based on our results and analysis of the experiments.

How does the value of the prefix parameter effect the precision and recall of the Generic Metadata Time Carving method?

We hypothesized that as the length of the prefix of the most significant bytes, p , of a potential timestamp decreased, that this in turn would increase the recall but reduce the precision of the Generic Metadata Time Carving method. According to our results, the recall for finding metadata records may significantly increase when applying prefix-based timestamp carving, but the precision in our experiments did not decrease when applying prefix-based timestamp carving. In fact, the precision remained at 100% for all possible prefix values. These items require a short discussion.

It seems that the recall reaches a point of diminishing returns once the timestamp prefix length $p \leq 2$, no matter what the filesystem is. We cannot suggest to make the value of p as low as possible either, as reducing $p = 2$ to $p = 1$ increased the number of potential time timestamps locations in the Ext4 experiment by 55.9 million (increasing parser time by over 100%) and in the Large NTFS experiment by 11.1 million. As noted in the Limitations subsection, decreasing p yields more potential timestamp locations, most of which will not be timestamps at all, but also allows for greater filesystem metadata record recall.

Despite producing much greater recall and many more potential timestamps, lowering the prefix-length p did not reduce the precision for carving MFT records or inodes. Our brief further investigations in the Limitations subsection demonstrated that by using the filesystem specific parsers on an encrypted version of the 59.5 Ext4 image produced no false positive record hits. We mention the trade-offs between the potential timestamp carver and parsers when addressing the last research question.

But overall, we can state with confidence, according to our experiments, that decreasing the value of the prefix parameter p can drastically increase the recall of finding metadata records, without much (if any) loss in precision of identifying metadata records.

How does the original Generic Metadata Time Carving method compare with our

prefix matching implementation?

For comparing our GMTC method to the original, we simply set the value of the prefix length, p , of a potential timestamp to its maximum (8 for NTFS or 4 for Ext4). In the small NTFS experiment, the exact matching timestamp method only achieved a recall of 8.8% for carving MFT records from the MFT table, while using the prefix-based method achieved 97.6% recall. Then in the Ext4 experiment, the exact matching timestamp method achieved a 91% recall for carving inodes from the inode table, while using the prefix-based method achieved 94.2% recall. In the Large NTFS experiment, the exact matching timestamp method only achieved a recall of 41.6% for carving MFT records from the MFT table, while, using the prefix-based method achieved 97.3% recall. The precision-recall experiments on the \$LogFile also showed improvement of recall as p decreased, though not nearly as drastic as the MFT Table experiments. In fact, our results indicate that the exact matching GMTC method performs nearly identically on the \$LogFile from NTFS as our prefix-based version, due to the nature of MFT records found within the file. In all experiments, the precision remained a constant 100%.

Our results indicate that the degree of improvement of the recall is dependent upon the temporal variety of the filesystem metadata records. Both the \$LogFile and inode table results showed only a minor improvement in recall since both the data sources appeared to have static records. As the records in the \$MFT from both NTFS images were more often updated, then the improvement in recall was much more significant.

Overall, we have shown that the prefix-based GMTC method can potentially carve a significantly greater number of filesystem metadata records than the original, while maintaining perfect or near-perfect precision for realistic test datasets.

In terms of time and space complexity, the time complexity of the prefix matching algorithm is the same as the exact matching algorithm. Our results show that the timestamp carving times are close to constant for all values of p , but carving with lower values of p will take slightly less time. A limitation of our work is that we did not perform extensive tests on the original GMTC algorithm, thus making statements on the speed of our algorithm versus the original mostly theoretical. Where the prefix-based GMTC method performs worse than the original method, is the space required for the potential timestamp locations produced by the potential timestamp carver, and consequentially the time required by the filesystem specific parsers. For example, the exact timestamp carving on the Ext4 image identified nearly 11 million potential timestamps and the parser took about 46 seconds to run, but carving for timestamps with a prefix length of 1 byte on the same im-

age identified nearly 98 million potential timestamps and the parser took about 7.5 minutes to run. As there were only 7436 inodes in the Ext4 partition's inode table, the grand majority of the potential timestamps are false positive timestamps.

The rather unexpected results of the timing of the NTFS parser for the Lone Wolf image, where the time to parse the image decreased when applying $p = 1$, is likely the result of the implementation of the NTFS parser.

Do the experimental results indicate that Generic Metadata Time Carving, prefix matching or otherwise, may be used in realistic digital forensic scenarios?

In terms of functionality, the prefix-based GMTC method appears practical for carving filesystem metadata records as our experiments achieved recall of approximately 90% or greater. The exact matching GMTC can be practical if time is of primary concern, or one wishes to carve for records in specific files such as \$LogFile (where the existence of dataruns is rare), but for many files or regions of disk this will risk missing many filesystem metadata records.

As filesystem metadata records are highly structured (and often sparse) data, and our filesystem specific parsers run many verification tests, we can understand why our parsers filtered out all of the tested false positive potential timestamps. Further investigations in our Limitations subsection showed that even when running our tools on the encrypted Ext4 image, that we encountered no false positives. The implication is that while potential timestamp carving will allow for greater recall, what ultimately controls the precision are the filesystem specific parsers, and that the precision measured in all cases was 100%.

However, we also showed in the Limitations subsection that by running the parser without potential timestamp information on the disk images took a significantly longer time than the GMTC method. For example, the prefix-based GMTC method took about 35 minutes to fully run on the encrypted Ext4 image when searching for inodes, whereas running the Ext4 parser alone took about 17.5 hours. Applying the prefix-based GMTC method to the encrypted image took 24 minutes to search for MFT records, while running the NTFS parser directly on the image took 5.75 hours.

In terms of time and space both the exact matching and prefix matching GMTC methods are practical. The time taken to carve out potential timestamps on the 476 GB NTFS image was on average just over two hours. Furthermore, the carving time is not much affected by the change in the prefix length p , as was predicted by the fact that the time complexity of the prefix-based timestamp carving method is

the same as the exact timestamp carving method. In general, it appears that as we allow for smaller prefixes in timestamp carving, the time it takes for the filesystem specific parsers to complete increases. The exception to this rule is the NTFS parser for large files, but we believe this to be more of an implementation issue than indicative of a general trend. Even so, the longest time it took for the NTFS parser to scan the Lone Wolf image was about 46 minutes. Thus, the longest time for total analysis of the 476 GB NTFS image was about 2 hours and 48 minutes (as seen in Table C.11).

In summary, there is a performance trade-off that exists for prefix-based Generic Metadata Time Carving, where lowering the prefix parameter p may significantly increase recall, slightly reduces timestamp carving time, but can also significantly increase the filesystem specific parser time due to having the need to validate more potential timestamps.

C.6 Conclusion and Further Work

In this work, we created and applied a timestamp prefix matching version of the Generic Metadata Time Carving (GMTC) method (Nordvik et al. 2020). The GMTC method can be used to carve for filesystem metadata records from a forensic image without the use of the filesystem, and can potentially allow for full file recovery on a damaged or partially overwritten disk. The crux of our contribution was the prefix-based potential timestamp carving algorithm, that only compares the prefixes of length p as opposed to the entire timestamp. This is because stringologically similar timestamps in most cases should be temporally similar as well. We tested the prefix-based method on three realistic forensic images. Two of the images used NTFS, one of the images used Ext4, and they varied in size from 1 GB to 476 GB.

Our location-based data recovery experiments mostly support our hypotheses. First, we have shown that applying timestamp prefix matching to the GMTC method can produce significantly greater recall in carving filesystem metadata records than the exact timestamp matching version. Surprisingly, performing prefix-based timestamp matching did not appear to affect the precision for carving MFT records or inodes from our test data, as we obtained 100% precision for all of our experiments. Further examinations in the Limitations subsection shows that prefix-based potential timestamp carving will increase the number of potentially valid offsets to metadata timestamps, but it is ultimately the filtering done by the filesystem specific parsers that controls the precision. However, running the parsers on an image without prior potential timestamp information will take significantly longer than using a GMTC method. Using the prefix-based Generic Metadata Time Carving method, the potential timestamp carver essentially performs data reduction of pos-

sible MFT record or inode locations for the filesystem specific parsers to check. The method appears to be practical, as our longest experiment on a 476 GB image in total clocked in at about 2 hours and 48 minutes.

Interestingly, changing the size of the matching prefix for the timestamps does not affect the time taken to perform timestamp carving by much. This makes sense as the prefix-based potential timestamp carving algorithm only added a constant number of steps to the original algorithm, therefore producing an algorithm with the same time complexity. Our experiments showed that timestamp carving with lower values of p took slightly less time than experiments with larger p values. On the other hand, reducing the size of the timestamp prefix often greatly increased the time taken by the filesystem specific parsers to extract the metadata records. This is due to the fact that matching for timestamps that are approximately similar results in some magnitudes more of potential timestamps to consider, and thus causing some magnitudes more time to run the filesystem specific parsers. We noted an exception to this rule for the large NTFS image, but this may be due to its implementation and the fact it does not use Python memory mapping libraries as the Ext4 parser does.

Future work for the prefix-based timestamp carving algorithm would be to try to improve its efficiency. Since the algorithm ingests the disk image in a linear fashion, perhaps the efficiency could be improved by using parallel processing to analyze different parts of the disk simultaneously, much like Garfinkel's Bulk Extractor ([Garfinkel 2013](#)). The filesystem specific parsers can also be further optimized.

Our work has shown there needs to be improvements made to the filesystem specific parsers as well, so that they can handle more possible variations to filesystem metadata records. For instance, the NTFS parser needs to be able to handle MFT records with non-resident attributes. For Ext4, there needs to be hard link support, and better support for symbolic links. Then in general, there is also a need for development of parsers of filesystems other than NTFS and Ext4.

C.7 Acknowledgement

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project "Ars Forensica - Computational Forensics for Large-scale Fraud Detection, Crime Investigation & Prevention", grant agreement 248094/O70.

C.8 Prefix-Based Potential Timestamp Carving Algorithm

Note, we do not include the memory mapping aspects to handle large files. For the full potential timestamp carving program, see:

<https://github.com/ TimestampPrefixCarving/ Peer-Review/blob/ main/main.cpp>.

Input: Raw disk image T as a byte array. Parameters:

m # Length of timestamp;

k # Length of search threshold;

h # Threshold for number of required matching timestamps per record;

p # Prefix Length;

Output: Potential timestamp offsets from beginning of image (in bytes) in txt file.

$i = 0$ # Current byte offset from start of image;

while ($i < |T| - k$) **do**

$candidateTS = T[i : (i + m)]$;

 # Boolean holding status of repeated byte sequence check;

$repeat = \text{True}$;

for $b \leftarrow 1$ **to** $m - 1$ **by** 1 **do**

$repeat = repeat \ \& \ (candidateTS[0] == candidateTS[b])$;

end

 # Variable for holding value of a string of characters read little-endian and transformed into a numerical value;

$littleEndian = 0$;

if $m == 8$ **then**

$littleEndian = (candidateTS[0] \ll (8 * 0) | \dots | candidateTS[7] \ll (8 * 7))$;

else if $m == 4$ **then**

$littleEndian = (candidateTS[0] \ll (8 * 0) | \dots | candidateTS[3] \ll (8 * 3))$;

 #If candidate timestamp is not a repeated sequence of bytes, and the prefix value of $littleEndian$ is not 0;

if ($\neg repeat \ \& \ ((littleEndian \gg 8 * (m - p)) \neq 0)$) **then**

$matchCount = 0$;

$j = i + m$;

while ($j < i + m + k$) **do**

$testSequence = 0$;

if $m == 8$ **then**

$testSequence = (T[j] \ll (8 * 0) | \dots | (T[j + 7] \ll (8 * 7)))$;

else if $m == 4$ **then**

$testSequence = (T[j] \ll (8 * 0) | \dots | (T[j + 3] \ll (8 * 3)))$;

 #Our timestamp prefix matching equivalency check;

$xorResult = littleEndian \oplus testSequence$;

if ($(xorResult \gg (8 * (m - p))) == 0$) **then**

$matchCount + = 1$;

end

$j + = m$;

if ($matchCount \geq (h - 1)$) **then**

 #Print Byte Location i ;

$j = i + m + k$;

$i + = (k - m)$;

end

end

end

$i + = m$;

end

Algorithm 3: Prefix-based potential timestamp carving algorithm, using timestamp prefix matching for the timestamp equivalency test.

Bibliography

- Active KillDisk (2021). How to... for killdisk software.
- Aho, A. V. and Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340.
- Bayne, E., Ferguson, R. I., and Sampson, A. (2018). Openforensics: A digital forensics gpu pattern matching approach for the 21st century. *Digital Investigation*, 24:S29–S37. The Proceedings of the Fifth Annual DFRWS Europe.
- Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772.
- Carrier, B. (2005). *File System Forensic Analysis*. Addison-Wesley Professional.
- Casey, E., Nelson, A., and Hyde, J. (2019). Standardization of file recovery classification and authentication. *Digital Investigation*.
- Cowen, D. and Seyer, M. (2013). File system journal analysis. SANS DFIR.
- Dewald, A. and Seufert, S. (2017). Afeic: Advanced forensic Ext4 inode carving. *Digital Investigation*, 20:S83 – S91. DFRWS 2017 Europe.
- Ext4 (and Ext2/Ext3) Wiki (2019). Ext4 disk layout.
- forensicswiki.xyz (2012). File carving.
- Garfinkel, S. L. (2007). Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2 – 12.
- Garfinkel, S. L. (2013). Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:56 – 72.

- Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160.
- Knutson, T. and Carbone, R. (2016). Filesystem timestamps: What makes them tick? *GIAC GCFA Gold Certification*, 11.
- Levenshtein, V. I. (1966). Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710.
- Maar, R. (2014). Ext4magic.
- Mccash, J. (2010). Timestamped registry & NTFS artifacts from unallocated space.
- Moore, Garfinkel, Farrell, Roussev, and Dinolt (2018). 2018 lone wolf scenario. Last visited: 2020-04-16.
- Mueller, L. (2008). Search for windows 64 bit timestamps.
- NIST (2017). Computer forensic reference data sets: Deleted file recovery.
- Nordvik, R., Georges, H., Toolan, F., and Axelsson, S. (2019). Reverse engineering of ReFS. *Digital Investigation*, 30:127 – 147.
- Nordvik, R., Porter, K., Toolan, F., Axelsson, S., and Franke, K. (2020). Generic metadata time carving. *Digital Investigation*, 33:301005. The Proceedings of the Twentieth Annual DFRWS USA.
- Plum, J. and Dewald, A. (2018). Forensic apfs file recovery. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018*, pages 47:1–47:10, New York, NY, USA. ACM.
- Richard III, G. G. and Roussev, V. (2005). Scalpel: A frugal, high performance file carver. In *Proceedings of the Digital Forensic Research Conference (2005)*.
- Schicht, J. (2018). Logfileparser readme. Github.
- Son, N., Lee, Y., Kim, D., James, J. I., Lee, S., and Lee, K. (2013). A study of user data integrity during acquisition of android devices. *Digital Investigation*, 10:S3 – S11. The Proceedings of the Thirteenth Annual DFRWS Conference.
- TWRP (2019). Download twrp-3.3.1-2-dreamlte.img.tar. <https://eu.dl.twrp.me/dreamlte/twrp-3.3.1-2-dreamlte.img.tar.html>.

- Vidas, T., Zhang, C., and Christin, N. (2011). Toward a general collection methodology for android devices. *Digital Investigation*, 8:S14 – S24. The Proceedings of the Eleventh Annual DFRWS Conference.
- Zha, X. and Sahni, S. (2010). Fast in-place file carving for digital forensics. In *International Conference on Forensics in Telecommunications, Information, and Multimedia*, pages 141–158. Springer.

Appendix D

Publication D: It is about time—Do exFAT implementations handle timestamps correctly?

It is about time—Do exFAT implementations handle timestamps correctly?, Rune Nordvik and Stefan Axelsson. In: *Forensic Science International: Digital Investigation Volumes 42/43*. October-December 2022, Pages 301476. DOI: <https://doi.org/10.1016/j.fsidi.2022.301476>

Note: The version included in this thesis is a corrected version, including the corrigendum of the original paper (Nordvik and Axelsson 2023).

abstract

Digital forensic investigations require that file metadata are interpreted correctly. In this paper we focus on the timestamps of the exFAT file system. How these timestamps are written may depend on the implementation of the file system. We have performed experiments using Windows, MacOS and Linux to examine whether the respective file system drivers for exFAT use timestamps in the same manner, and whether they take the directory entry *UTCOffset* fields into account. We have also studied whether the forensic tools: Autopsy, X-Ways Forensics, EnCase Examiner, and FTK Imager interpret the timestamps consistently.

The results show that there are substantial inconsistencies both in the file system implementations, in atomic storage features implemented by applications or their libraries, and in how forensic tools handle these inconsistencies. For the unwary forensic examiner, there is a clear risk of interpreting timestamps incorrectly by a substantial margin.

We conclude that timestamp interpretation during criminal investigations should not be based on the assumption that the file system specifications are followed flawlessly by the file system driver developers or necessarily interpreted and displayed correctly by the digital forensic tools.

D.1 Introduction

During the investigation of criminal cases it is important that timestamps are interpreted correctly. Misinterpreting timestamps may exclude the guilty or implicate the innocent. For instance; a Word document on a USB stick belonging to the suspect with a creation timestamp corresponding to the time the crime was committed may indicate that the suspect was using a computer at the time of the crime to store the file on the USB stick. If traces found on the suspect's home computer also show that the same USB stick was inserted one hour before the time of the crime, and removed a day after, this may indicate that someone was at the suspect's home at the creation of the document. If the crime took place at another address at the same time, this finding may exclude the person as a suspect if the investigation can connect the suspect to the computer. If further hypothesis testing shows that the computer has not been connected to any other network than the home network, and that the computer clock has not been manipulated, this will strengthen the main hypotheses that the suspect or someone else was at the suspect's home address at the time of the crime. However, more detailed hypothesis testing must be performed before this conclusion is firmly drawn.

The above deductions can only be drawn if file systems follow their specifications.

However, these may not be available or only partly available. Specifications may also be misinterpreted by the implementer. For example, a previous study of N-version programming by Knight and Leveson (1986) show that developers tend to make similar mistakes even when they are following the same specification, i.e. the flaws they introduce in the code are not independent of each other. This means that there is little support to trust a tool is correct only by comparing its results with a similar tool (Nordvik et al. 2021). In addition, applications and libraries may impact file metadata by using atomic storage approaches (Suhanov 2022).

In this paper we focus on the exFAT file system. The specifications are available from Microsoft (2021b). The exFAT file system can be used on Windows, MacOS, Linux, and other operating systems (Bretel 2017). Off the shelf removable storage devices are today often pre-formatted with exFAT as it can work on most computers and supports file system volume sizes much larger than the 32 GiB FAT32 limitation (Microsoft 2021a). These removable devices can be mounted with read and write support on all previous mentioned operating systems, all thanks to the usage of the exFAT file system (USB Memory Direct 2022).

For most users of file systems it is not critical that timestamps are accurate. However, when investigating criminal cases the accuracy of times given could be critical in order to answer the **when** question in the 5WH questions (Jeong 2006). Jeong (2006) describes these 5WH questions as “What (the data attributes), Why (the motivation), How (the procedures), Who (the people), Where (the location) and When (the time)”.

When it comes to the investigation of metadata, such as timestamps from file systems, most investigations take for granted that the digital forensic tools are able to parse the file systems that they claim to support, and courts depend on the tools accuracy and reliability (Nordvik et al. 2021). As timestamp interpretations may give a suspect an alibi, the investigation should not rely solely on tool interpretation.

Law enforcement organisations typically have a large backlog of seized devices waiting for acquisition and analysis (Scanlon 2016). Digital forensic investigators use digital forensic tools when investigating criminal cases to increase the efficiency of the investigation. The concept of trust in criminal investigations is discussed by Neale et al. (2022), including erroneous trust in the accuracy of digital forensic tools. Error rates for digital forensic tools are seldom available (Nordvik et al. 2021), and previous research have shown that it is difficult to measure error rates involving all independent variables (conditions) that may impact the dependent variable (here the error rate) (Lyle 2010).

In order to comply with certain human rights, like the right to privacy, family life,

and correspondence (Article 8) ([European Court of Human Rights 2021](#)), law enforcement may utilise a more granular acquisition of files, e.g. by only including files from a specific time range ([European Committee for Standardization 2022](#)). In these cases law enforcement of course depend on accurate timestamp interpretations.

Therefore, one can hardly overemphasise the importance of manual verification of tool findings, and how file system timestamps are interpreted.

The contributions of this paper:

- File system driver developers do not implement exFAT equally.
- When specifications are made available, they are not necessarily followed.
- Digital forensic tools have a tendency to make assumptions about metadata.
- Even when specifications are available, reverse engineering by performing black box testing is necessary.
- Atomic storage implemented by applications or libraries may impact timestamp accuracy.

To the best of our knowledge, it has not been performed experiments including multiple operating systems and multiple driver implementations of the same exFAT file system. Since digital forensic investigators do not necessarily know which operating system a removable device has been connected to, they should not assume that the exFAT specifications were followed by the driver. Even building digital forensic tools to automate file system parsing require a detailed understanding about how the file system drivers store metadata, and this paper will show that it is not necessarily always the case.

D.1.1 Background

The background information presented in this section is based on [Microsoft \(2021b\)](#). The exFAT file system has a volume boot record (VBR) which contains information necessary to find the important metadata structures such as the file allocation table (FAT), the cluster heap (data region), and the cluster of the Root directory. It also defines the size of a sector, cluster, the size of the volume, the number of FATs, the length of each FAT, and percentage of allocated clusters in the cluster heap.

The data region starts at cluster 2 and it is recommended that exFAT implementations place the root directory after the clusters used for the allocation bitmap

and the up-case table. The allocation bitmap defines the allocation status of all clusters in the data region. The allocation bitmap has a corresponding set of directory entries, and its primary directory entry has the type 0x81. The number of allocation bitmaps correspond with the number of FATs, and this is normally 1, or maximum of 2.

The exFAT FAT table is mainly used for fragmented files, which are files that do not use contiguous clusters. When a file becomes fragmented, the stream extension directory entry will set the *NoFatChain* field to zero, meaning the FAT is in use. By using the *FirstCluster* field in the stream extension directory entry, the system identifies the correct cluster start in the FAT, and it can continue to the next cluster in the allocation chain by reading FAT chain. This enables the system to find all fragmented clusters for the file. If the file is not fragmented, the FAT is not in use, by setting the *NoFatChain* to one. Then the system can use the *FirstCluster* and the *DataLength* fields to extract the contiguous clusters for the file.

85 02 B0 7F	30 00 00 00	97 AE 57 54	98 AE 57 54	... ° 0	-@WT@ WT
97 AE 57 54	49 48 F4 F4	F4 00 00 00	00 00 00 00	-@WTIHôôôô	
C0 03 00 0C	5C D5 00 00	00 80 00 00	00 00 00 00	Ä \ö €	
00 00 00 00	43 00 00 00	00 80 00 00	00 00 00 00	c e	
C1 00 45 00	78 00 70 00	65 00 72 00	69 00 6D 00	Á E x p e r i m	
65 00 6E 00	74 00 2D 00	30 00 00 00	00 00 00 00	e n t - 0	

Figure D.1: ExFat set of directory entries.

Binary	
Type 0x85	1 0 0 0 1 0 0 1
Type 0xC0	1 1 0 0 0 0 0 0
Type 0xC1	1 1 0 0 1 0 0 1

Allocated directory entry set

Binary	
Type 0x05	0 0 0 0 1 0 0 1
Type 0x40	0 1 0 0 0 0 0 0
Type 0x41	0 1 0 0 1 0 0 1

Unallocated directory entry set

Figure D.2: ExFat allocation of directory entries.

The root directory contains files and folders and each of them have a set of directory entries, as shown in Figure D.1. Allocated files have a file directory entry

(type 0x85), a stream extension directory entry (type 0xC0), and one or more file name directory entries (type 0xC1), which is a set of allocated directory entries as illustrated in the top part of Figure D.2. The file directory entry contains timestamps, *UTCOffset* fields, file attributes, and a directory entry set checksum. The stream extension directory entry gives information about where the data content (the stream) is stored (*FirstCluster*, *DataLength*), the length of the *FileName*, and a hash of the *FileName* in upper case. The file name directory entry describes the name of a file, and will need $\text{NameLength}/15$ file name directory entries.

In this paper we mainly focus on the timestamps found in file directory entries, as shown in Table D.1. There are three different timestamps in a file directory entry (FDE); Create, Last Modified, and Last Accessed. According to the specifications, to interpret the actual time the timestamp, the 10ms increments, and the UTC offset should be considered (Microsoft 2021b). The UTC offset describes the offset from UTC to local time (including daylight savings adjustments) in 15 minute increments. The 10ms increments are only available for the Create and the Last Modified timestamp, and increases the granularity from 2 seconds to 10 milliseconds.

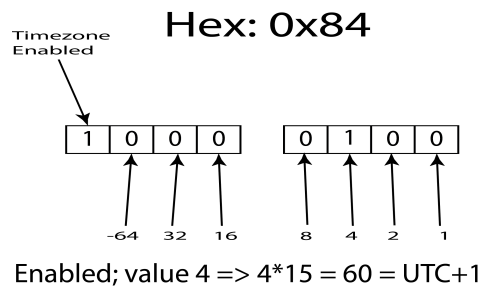


Figure D.3: ExFat timeszone field, from hex byte to UTC offset.

In Figure D.3 we see the *UTCOffset* value 0x84. This value can be converted to the UTC offset used since the timezone enabled bit is set. We do not count the timezone enabled bit, and get the value 0x04, meaning UTC+1:00 because each unit corresponds to 15 minute intervals.

For more information about the exFAT file system please see Schullich (2009).

D.1.2 Research problem

Since the exFAT file system is supported on all main desktop operating systems (Bretel 2017), can we be sure that the specifications are followed by the exFAT file system developers for each of these platforms? Previous research has not tested exFAT implementations on multiple platforms, and our contribution will bridge that gap.

Field Name	Offset	Size	Value example	Comments
EntryType	0x0	0x1	0x85	Regular primary directory entry in use
SecondaryCount	0x01	0x01	0x04	Number of secondary directory entries in this set
SetChecksum	0x02	0x02		A checksum of all bytes (except this field) of directory entry set
FileAttributes	0x04	0x02	0x20	DOS mode: Archive and a file
Reserved1	0x06	0x02	0x00	Reserved
CreateTimestamp	0x08	0x04	0xAA493A54	Created 2022-01-26 09:13:20
LastModifiedTimestamp	0x0C	0x04	0xAA493A54	Modified 2022-01-26 09:13:20
LastAccessedTimestamp	0x10	0x04	0xAA493A54	Accessed 2022-01-26 09:13:20
Create10msIncrement	0x14	0x01		10 ms increments for Create
LastModified10msIncrement	0x15	0x01		10 ms increments for Modified
CreateUtcOffset	0x16	0x01	0xF4	Valid, UTC-3
LastModifiedUtcOffset	0x17	0x01	0xF4	Valid, UTC-3
LastAccessedUtcOffset	0x18	0x01	0xF4	Valid, UTC-3
Reserved2	0x19	0x07	0x00	Reserved

Table D.1: File directory entry (all fields are mandatory)

A new feature of the exFAT compared to the FAT32 is the *UTCOffset* field for each timestamp. Since the specifications describe that the value stored is the offset from UTC to local time including any daylight adjustments (Microsoft 2021b), it is relevant in an investigative context to test if we can find the timezone of the computer used to store the data on the file system by interpreting the file system metadata only. Currently, most Digital forensic tools claim support for the exFAT file system, but is this support accurate and reliable, and can it be validated for law enforcement purposes? Do the latest versions of the tools follow the exFAT specifications, and how do they handle exFAT implementations that do not comply with the exFAT specifications? The problems described above are defined as the following research questions:

- How do current exFAT implementations store timestamps?
- Can we use the UTC offset stored in a directory entry to describe the local time of the computer?
- Do current forensic tools interpret exFAT timestamps differently?

The main hypothesis:

- H_1 : the local time (the base truth) is related to the time stored within the primary directory entry when the timezone offset is valid.

The null hypothesis:

- H_0 : the local time is not related to the timezone offset in the primary directory entry when the timezone offset is valid.

The $\alpha = 0.01$ is the significance level, meaning if less than 1 percent of the observations supports the null hypothesis, then the null hypothesis is falsified, giving additional strength to the corresponding main hypothesis.

We have only defined one main hypothesis that is related to all three research questions, and the main hypothesis is based on an assumption that the exFAT specifications are followed, both by exFAT driver developers and by digital forensic tool developers.

D.1.3 Organisation of this paper

In the [Introduction](#) section we have introduced the importance of interpreting file system metadata, especially timestamps, in an investigative context, and we have given a short introduction to the exFAT file system. In addition, we have defined the research problems. In the [Related Work](#) section we summarise the current state of the art research related to exFAT. In the [Methodology](#) section we describe the methods we used for the experiments for all the supported operating systems (Windows, MacOS, and Linux), and in the [Results](#) section we describe our results. Then we discuss our results in the [Discussion](#) section, and we conclude in the [Conclusion and Further Work](#) section.

D.2 Related Work

[Hamm \(2009\)](#) was of the first to publish documentation about the exFAT file system structures. He described the file system from the view of a Digital Forensic practitioner. The exFAT system is similar to the old FAT systems, but each file has a set of directory entries which describe metadata of files. The file allocation table is mainly used for fragmented files, and an allocation bitmap file for describing which cluster (block) is allocated. [Hamm \(2009\)](#) describes that exFAT was first introduced in Windows CE in 2006, and then in Vista SP1 in 2008. In 2009 Windows XP drivers for exFAT were released. This work was performed before Microsoft released the full specifications, and the work was based on the patent US 20090164440 A1 ([Microsoft 2009](#)), which describes most of the file system structures and their meaning.

[Schullich \(2009\)](#) continued the work from [Hamm \(2009\)](#) and described reverse engineering of the exFAT file system utilising black box analysis, using existing documentation such as patents, examination of other file systems in the FAT family, Google searches, Microsoft knowledge base, and low level examination of the exFAT file system. [Schullich \(2009\)](#) also developed a C program to output metadata structures. Files were created, added, deleted, and added again to observe the effect these operations had on the file system. The output of the C program was compared to the output of native Windows program such as *dir*, *chkdsk*, disk

management, and *Windows Explorer*. [Schullich \(2009\)](#) also describes the internals of exFAT and its metadata structures. The timezone value fields (*UTCOffset*) were found based on experiments and observations, they were not described in the patents. These fields describe the timezone offset in units of 15 minutes.

[Munegowda et al. \(2012\)](#) describe allocation strategies for exFAT and compares them to FAT32. The exFAT file system uses the allocation bitmap (they call it the cluster heap) to search for free clusters. If enough free contiguous clusters are available for allocation, then the “No FAT Chain” is set to 0, the allocation bitmap is set for the new allocated clusters, and the FAT is not used. The stream extension directory entry points to the first cluster. However, if it is not possible to allocate contiguous clusters the “No FAT Chain” is set to 1, the allocation bitmap is updated with the new allocated clusters, and the FAT is used.

Unfortunately, the use of the “No FAT Chain” is misinterpreted by [Munegowda et al. \(2012\)](#), since the specifications from [Microsoft \(2021b\)](#) state that the *NoFatChain* field is set to 1 if the clusters are contiguous, and 0 if the FAT cluster chain is in use.

[Munegowda et al. \(2014\)](#) describe how exFAT can implement directory compaction techniques when its first cluster only has deleted/unallocated entries. In this case the directory entry for this directory should be changed to point to the next cluster, and the previous first cluster should be marked free in the allocation bitmap and in the file allocation table (compaction).

[Ma et al. \(2015\)](#) describe different approaches for data recovery for the exFAT file system. An unallocated file will change the set of directory entries from the types 0x85, 0xC0, and 0xC1, to the types 0x05, 0x40, and 0x41. One approach is using the second directory entry (stream extension directory entry, here type 0x40) of an unallocated file. Read its cluster start, find the start sector and extract the size of the file. If the file is not stored in contiguous clusters and not in the FAT (if damaged), then the file may not be completely recovered. [Ma et al. \(2015\)](#) also describe carving using signatures, and a machine learning approach utilising a Support-Vector Machine (SVM) classification algorithm.

[Vandermeer et al. \(2018\)](#) describe how a set of exFAT directory entries can be unallocated not necessarily as a result of deletion. By combining information from the allocation bitmap it is possible to differentiate between renamed, moved or deleted files. All these scenarios will have a set of unallocated directory entry sets, but only the deletion scenario will also set the bits in the bitmap for the corresponding clusters to zero. If the clusters of an unallocated file is allocated in the bitmap, the file may just as well be moved or renamed. If the file clusters are zeroed out in the

allocation bitmap, then the file is deleted. They also proposed a methodology for recovering deleted files.

Heeger et al. (2021) describe anti-forensic techniques to hide data in the exFAT file system. They suggest the hiding of encrypted data in the *Create10msIncrement* and *LastModified10ms-Increment* fields, by only using six of the least significant bits in these single byte fields. In addition, one of the two most significant bits are set or none of these two bits are set. The *SetChecksum* field in the file directory entry is updated to take the hidden data into consideration. This process is done for all necessary directory entry sets used. Heeger et al. (2021) also suggest another approach called exHide which only uses metadata from deleted files. This approach uses the *Create10msIncrement* field (6 bits), and 1 bit from *CreateTimestamp* and *LastModifiedTimestamp* are used in order to create one byte. In addition, the FirstCluster and file size fields ValidDataLength and DataLength are used (the two latter needs to be equal). A total of 4 bytes are used for hiding for each metadata structure. For the exHide approach the *LastModified10msIncrement* was not used because Windows does not use this field when writing to the File System (Heeger et al. 2021).

D.3 Methodology

We used Linux Ubuntu 20.04 (using exFAT fuse v. 1.3), and Ubuntu 20.04 (using the native kernel exFAT driver), MacOS Monterey/Mojave (both using exFAT driver extension v. 1.4) and Windows 10 as target operating systems (OSes).

We repeated the Linux experiments after removing the exFAT fuse package in Linux Ubuntu 20.04 to enforce the usage of the native kernel exFAT driver.

All experiments are described in this section and illustrated in Figure D.4. The experiments A, C, and D were performed using a bash shell script in MacOS and Linux, and a batch script was used in Windows 10. Experiment B, and E were manually performed to simulate normal user activity. We have shared the scripts and the forensic images (Nordvik 2022).

An overview of all experiments are shown in Figure D.4.

D.3.1 Experiment A - Base

First we wiped the storage device, then we formatted it using the exFAT v. 1.0 file system. We performed experiments by utilising four different timezones, and for each timezone 100 files were created in their own directory on the USB storage device. Before each timezone change we performed an unmount and mount to make sure the data was written to the device. The local time as seen by the user and a timezone index (from 0-3) was encoded as a part of the filenames. After the ex-

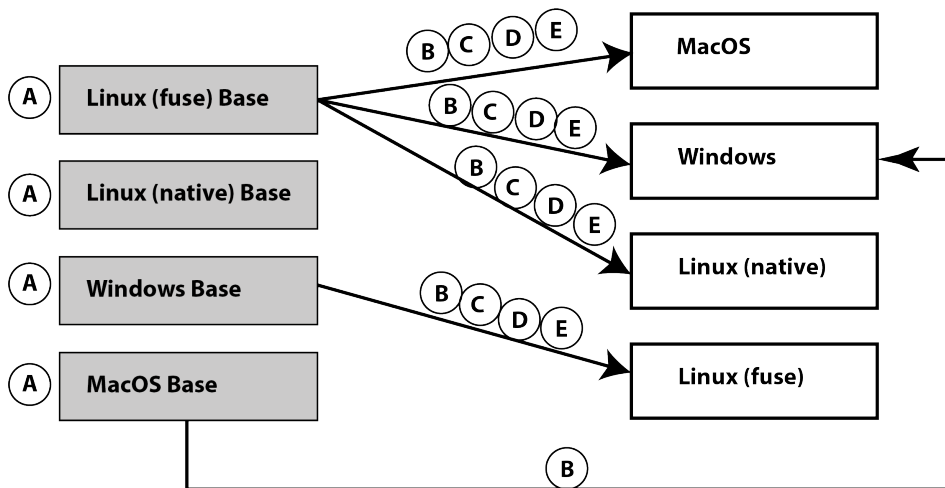


Figure D.4: Overview of all experiments, and all of them have a forensic image associated

periments were performed a forensic image was created for each target OS/driver. Metadata for the created files were observed and timestamp related information was interpreted.

The timezones index includes:

- Europe/Moscow (index 0, UTC+3). Files stored in the directory Experiment-0.
- America/Godthab (index 1, UTC-3). Files stored in the directory Experiment-1.
- Atlantic/Azores (index 2, UTC-1). Files stored in the directory Experiment-2.
- Europe/Oslo (index 3, UTC+1). Files stored in the directory Experiment-3.

For the Windows OS we used similar timezone values;

- Russian Standard Time (index 0, UTC+3). Files stored in the directory Experiment-0.
- E. South America Standard Time (index 1, UTC-3). Files stored in the directory Experiment-1.

- Azores Standard Time (index 2, UTC-1). Files stored in the directory Experiment-2.
- W. Europe Standard Time (index 3, UTC+1). Files stored in the directory Experiment-3.

The following forensic images were created in this experiment:

```
ExFAT-Base-Experiment-A-Linux-NativeExfat.E01  
ExFAT-Base-Experiment-A-Linux.E01  
ExFAT-Base-Experiment-A-MacOS.E01  
ExFAT-Base-Experiment-A-Windows10.E01
```

D.3.2 Experiment B - mounting and unmounting only

We used the Linux (fuse) base forensic image from experiment A, restored to the USB storage device using *ewfmount* and *dd* commands, then the timezone was changed to Europe/Oslo (UTC+1) for MacOS, and W. Europe Standard Time (UTC+1) for Windows, which is different from each base experiment 0, 1, 2. The storage device was mounted on MacOS or Windows 10. We also restored the Windows base forensic image, and mounted and unmounted the USB storage device on Linux.

We did not make any change to any file. Then we unmounted the device and created a forensic image for each target OS/driver.

The following forensic images were created in this experiment:

```
ExFAT-Experiment-B-Linux-MountedLinux-NativeExfat.E01  
ExFAT-Experiment-B-Linux-MountedMacOS.E01  
ExFAT-Experiment-B-Linux-MountedWindows.E01  
ExFAT-Experiment-B-MacOS-MountedWindows.E01  
ExFAT-Experiment-B-Windows-MountedLinux.E01
```

D.3.3 Experiment C - accessing selected files

We selected the Linux (fuse) base image when targeting MacOS, Windows and Linux native exFAT drivers, and we selected the Windows base image when targeting Linux exFAT fuse driver. We restored the base forensic images to the USB storage device using *ewfmount* and *dd* commands, and we re-mounted it on one of the other operating systems after changing the timezone to America/New_York (UTC-5). We opened the files in *TextEdit* on MacOS, *Notepad* in Windows 10, and *Gedit* in Linux and then closed each file. Then we created a forensic image for each target OS/driver. We did not change or save any content.

The following forensic images were created in this experiment:

```
ExFAT-Experiment-C-Linux-LinuxOpenFiles-NativeExfat.  
  ↪ E01  
ExFAT-Experiment-C-Linux-MacOpenFiles.E01  
ExFAT-Experiment-C-Windows-LinuxOpenFiles.E01  
ExFAT-Experiment-C-Linux-WindowsOpenFiles.E01
```

D.3.4 Experiment D - changing the content of all files

We performed experiments to change the files to simulate normal user activity. We selected the Linux (fuse) forensic image from the base experiments, restored to the USB storage device using *ewfmount* and *dd* commands, then re-mounted the device on one of the other operating systems. For every file in each of the 4 directories, we changed the content using the timezone America/New_York (UTC-5) for Linux and MacOS, and Eastern Standard Time (UTC-5) for Windows which is different from each base experiment. The batch script failed to set the timezone in Windows, and instead UTC+0 was used. This is still a difference from each base experiment. We therefore repeated the experiment using the correct timezone. Then we created a new forensic image for each target OS/driver. The metadata changes of the files were observed and documented.

The following forensic images were created in this experiment:

```
ExFAT-Experiment-D-Linux-Linux-Overwrite-Files-  
  ↪ NativeExfat.E01  
ExFAT-Experiment-D-Linux-Windows-Overwrite-Files.E01  
ExFAT-Experiment-D-Linux-Windows-Overwrite-Files-v2.  
  ↪ E01  
ExFAT-Experiment-D-Windows-Linux-Overwrite-Files.E01  
ExFATExperiment-D-Linux-MacOS-Overwrite-Files.E01
```

D.3.5 Experiment E - changing the content of selected files

We selected the Linux (fuse) base image, restored it to the USB storage device using *ewfmount* and *dd* commands, changed the timezone to America/New_York (UTC-5) for Linux and MacOS and Eastern Standard Time (UTC-5) for Windows, and re-mounted it on one of the other operating systems. We changed the content of selected files manually by opening them in TextEdit for MacOS, Notepad in Windows 10 (the timezone had entered daylight time, meaning UTC-4), or Gedit in Linux. Then we wrote a word and saved and closed each file. We only changed the files in the directory Experiment-0.

Then we created a forensic image for each target OS/driver. The following forensic

images were created in this experiment:

```
ExFAT-Experiment-E-Linux-Linux-Overwrite-Files-
  ↪ Manually-NativeExfat.E01
ExFAT-Experiment-E-Linux-MacOS-Overwrite-Manually.E01
ExFAT-Experiment-E-Linux-Windows-Overwrite-Manually.
  ↪ E01
ExFAT-Experiment-E-Windows-Linux-Overwrite-Manually.E01
```

D.3.6 Tool Testing

If the tool supported changing timezone, we adjusted to the timezone stored in the hex dump for each experiment. We tested the following Digital Forensic (DF) tools:

- Autopsy v. 4.19.3 (Windows version)
- FTK Imager v. 4.5.0.3 and v. 4.7.1.2
- X-Ways Forensics v. 20.04 SR-4
- EnCase Examiner v. 22.1

The FTK Imager does not support changing timezone, while Autopsy, X-Ways and EnCase do.

When testing the tools we did not only compare the result from different tools, but we also assessed the results manually in the directory entries of the exFAT file system. This manual verification was necessary since dual tool verification is not a reliable method (Nordvik et al. 2021, Knight and Leveson 1986).

D.3.7 Limitations and assumptions

We assumed that the current implementations of exFAT v. 1.0 store timestamps as localtime, and that each *UTCOffset* field describes the deviation between the local time and the UTC, which the exFAT specifications describes (Microsoft 2021b). However, an implementation may choose not to utilise the *UTCOffset* fields.

In the original version of this paper we assumed that the applications we used to perform the experiments did not implement atomic storage. It was brought to our attention that Gedit (Linux) and other applications may implement atomic storage, and we have confirmed that atomic storage is implemented in Gedit or in one of the libraries used by it. This finding has an impact on the interpretation of our

observations. Hence, we have updated some of our conclusions compared to the original publication.

We did not consider file systems that are manipulated, attacked, or where anti-forensic methods as described by [Wani et al. \(2020\)](#) are used. The experiment methodology described includes the actions that were performed on the USB storage device.

We also assume the exFAT file system interpretation by forensic tools was unreliable until we have verified the findings (Zero Trust ([Neale et al. 2022](#))).

D.4 Results

D.4.1 Experiment A - Creating files on an exFAT storage

Base TZ	Action	Stored TZ	Stored Time	Real TZ	Real Time	Observations
Europe/Moscow	Created	0xF4 (UTC-3)	23/02/2022 21:52	UTC+3	24/02/2022 03:52	100
America/Godthab	Created	0x8C (UTC+3)	24/02/2022 03:53	UTC-3	23/02/2022 21:53	100
Atlantic/Azores	Created	0x84 (UTC+1)	24/02/2022 01:53	UTC-1	23/02/2022 23:53	100
Europe/Oslo	Created	0xFC (UTC-1)	23/02/2022 23:53	UTC+1	24/02/2022 01:53	100

Table D.2: Experiment A Results - MacOS. We can see that stored timestamps use a time-zone offset with switched signs compared to the computer the experiments were executed on

We observed, as shown in [Table D.2](#), that for MacOS the timestamps were stored on disk in UTC-3 when the local time was UTC+3 (Europe/Moscow), and similar were the UTC offset switched from negative to positive for timezones with negative UTC offset.

If we normalise the stored timestamps to UTC-0, we can see that all files are created in the period 24/02/2022 00:52 to 00:53. The files were created using a script, explaining why they were near in creation time.

It was interesting to observe that the latest timezone used on MacOS also changed the last access time for all files, even for files not accessed.

Base TZ	Action	Stored TZ	Stored Time	Real TZ	Real Time	Observations
Russian Standard Time	Created	0x8C (UTC+3)	24/02/2022 21:23	UTC+3	24/02/2022 21:23	100
E. South America Standard Time	Created	0xF4 (UTC-3)	24/02/2022 15:23	UTC-3	24/02/2022 15:23	100
Azores Standard Time	Created	0xFC (UTC-1)	24/02/2022 17:24	UTC-1	24/02/2022 17:24	100
W. Europe Standard Time	Created	0x84 (UTC+1)	24/02/2022 19:24	UTC+1	24/02/2022 19:24	100

Table D.3: Experiment A Results - Win10. We can see that all types of timestamps are stored using the local UTC offset of the computer the experiments were executed on, and that real time is the same as stored time.

[Table D.3](#) shows that the Windows exFAT driver is following the exFAT specifica-

tions when setting the *UTCOffset* fields. In this case the local time (real time) was stored.

Table D.4 shows the results for the Linux Ubuntu experiment with the exFAT fuse driver, and it set the *UTCOffset* fields to 0x00, meaning these fields are not valid because the most significant bit is not set. We observed that the timestamps are stored using localtime. However, it is not possible to interpret what the local time UTC offset was by only assessing the stored timestamps and the *UTCOffset* fields.

Base TZ	Action	Stored TZ	Stored Time	Real TZ	Real Time	Observations
Europe/Moscow	Created	0x00 (Not used)	02/03/2022 16:11	UTC+3	02/03/2022 16:11	100
America/Godthab	Created	0x00 (Not used)	02/03/2022 10:12	UTC-3	02/03/2022 10:12	100
Atlantic/Azores	Created	0x00 (Not used)	02/03/2022 12:12	UTC-1	02/03/2022 12:12	100
Europe/Oslo	Created	0x00 (Not used)	02/03/2022 14:13	UTC+1	02/03/2022 14:13	100

Table D.4: Experiment A Results - Linux Ubuntu 20.04 using exFAT fuse v.1.3. We can see that the timestamps are stored using the local time of the computer the experiments were executed on, and that real time is the same as stored time. However, the *UTCOffset* fields are not in use.

Table D.5 shows the results for the Linux Ubuntu experiment using the native exFAT driver, and it sets the *UTCOffset* fields to 0x80, meaning these fields are valid and is set to UTC+0. The experiment shows that we cannot interpret what the local time UTC offset was by assessing only the stored timestamps and the *UTCOffset* fields.

Base TZ	Action	Stored TZ	Stored Time	Real TZ	Real Time	Observations
Europe/Moscow	Created	0x80	16/03/2022 14:48	UTC+3	16/03/2022 17:48	100
America/Godthab	Created	0x80	16/03/2022 14:49	UTC-3	16/03/2022 11:49	100
Atlantic/Azores	Created	0x80	16/03/2022 14:49	UTC-1	16/03/2022 13:49	100
Europe/Oslo	Created	0x80	16/03/2022 14:50	UTC+1	16/03/2022 15:50	100

Table D.5: Experiment A Results - Linux Ubuntu 20.04 using exFAT native driver. We can see that the timestamps are stored using UTC+0, not the local time of the computer the experiments were executed on. The *UTCOffset* fields are used (set to 0x80).

The Experiment A shows that:

- 400 of 1600 observations show the usage of the 0x00 invalid *UTCOffset* value. Invalid values are excluded from hypothesis testing.
- 400 of 1200 (33 percent) show the usage of 0x80 valid value, even when local time deviates from UTC+0.

- 800 of 1200 (67 percent) observations take the local time into consideration when storing timestamps and UTC offsets.

This means that our main hypothesis is not true for all operating systems.

D.4.2 Experiment B - Mounting exFAT storage

OS	Action	Timestamp	10msIncrement	UtcOffset	Observations	New Directories
MacOS	Mount/ unmount	LA	Not changed	LA (switched sign)	400	.fsevents, .SpotLight-V100
Windows	Mount/ unmount	Not changed	Not changed	Not changed	400	System Volume Information
Linux	Mount/ unmount	Not changed	Not changed	Not changed	400	

Table D.6: Experiment B Results - MacOS. Impact of mounting and unmounting

The result shown in Table D.6 shows that MacOS will change the *UTCOffset* of the last accessed timestamp for all files when a USB storage device is mounted and unmounted. It also shows that when mounted on MacOS the directories *.fsevents* and *.SpotLight-V100* were created. When mounted on Windows, the *System Volume Information* directory was created.

D.4.3 Experiment C - Opening files

OS	Action	Tool used	Timestamp	10msIncrement	UtcOffset	Observations
Linux (fuse)	Open	Gedit	LA using local time	Not changed	Not changed	400
Linux (native)	Open	Gedit	LA using UTC+0	Not changed	All to 0x80	400
MacOS	Open	TextEdit	LM(*) and LA using local time	Not changed	LM and LA (switched)	400
Windows	Open	Notepad	Not changed	Not changed	Not changed	400

Table D.7: Experiment C Results. Impact of opening files

In the Experiment C in Table D.7, we used the timezone *America/New_York* (UTC-5). The last accessed timestamp was changed and stored using UTC-5 (local time), but the *UTCOffset* fields were not changed in Linux exFAT fuse driver. Since we used the Windows base forensic image, and *UTCOffset* fields were not touched in Linux (exFAT fuse) when opening files using *Gedit*, the local timestamp stored does correspond with the preserved *UTCOffset* fields for the last modified and the created, but not necessarily the last accessed. We illustrate this in Figure D.5.

For the Linux native driver we used the base of the Linux exFAT fuse, and here all *UTCOffset* fields were changed to 0x80 when using the Linux exFAT native driver, even though only the timestamp last accessed was changed. This change shows that the native driver interprets the previous value 0x00 *UTCOffset* fields as timestamps stored in UTC+0, which was an incorrect assumption for all our experiments. We illustrate this in Figure D.6.

It was strange that the last modified timestamp was changed when files were opened using *TextEdit* on MacOS, especially because we did not change the con-

Windows Base, experiment-0 (UTC+3)

Create	LastModified	LastAccessed
UTC+3	UTC+3	UTC+3
24/02/2022 21:23	24/02/2022 21:23	24/02/2022 21:23

New UTC offset: UTC-5, then open using Gedit in Linux (exFAT fuse)

Create	LastModified	LastAccessed
UTC+3	UTC+3	UTC+3
24/02/2022 21:23	24/02/2022 21:23	10/03/2022 07:28 LT

Figure D.5: Changes in timestamps and *UTCOffset* fields when opening a file using Gedit in Linux Ubuntu 20.04 exFAT fuse driver. The *LastAccessedTimestamp* is changed using local time (LT), which was UTC-5, however the *LastAccessedUtcOffset* is not changed. In this case the last accessed is inaccurate.

Linux (exFAT fuse) Base, experiment-0 (UTC+3)

Create	LastModified	LastAccessed
LT	LT	LT
02/03/2022 16:11	02/03/2022 16:11	02/03/2022 16:11

New UTC offset: UTC-5, then open using Gedit in Linux (exFAT native)

Create	LastModified	LastAccessed
UTC+0	UTC+0	UTC+0
02/03/2022 16:11	02/03/2022 16:11	17/03/2022 08:40

Figure D.6: Changes in timestamps and *UTCOffset* fields when opening a file using Gedit in Linux Ubuntu 20.04 exFAT native driver. The *LastAccessedTimestamp* is stored as UTC+0, however the *CreateUtcOffset* and *LastModifiedUtcOffset* are also changed to UTC+0, but not the timestamps. In this case the create and last modified timestamps are inaccurate.

tent of any files. The timestamp for last modified was just converted to use the local UTC offset of the computer. The previous last modified timestamp was assumed to be UTC-5 (our local UTC offset) and then the timestamp was converted

Linux (exFAT fuse) Base, experiment-0 (UTC+3)

Create	LastModified	LastAccessed
LT	LT	LT
02/03/2022 16:11	02/03/2022 16:11	02/03/2022 16:11

New UTC offset: UTC-5, then open using TextEdit in MacOS

Create	LastModified	LastAccessed
LT	UTC+5	UTC+5
02/03/2022 16:11	03/03/2022 02:11	11/03/2022 13:18

Figure D.7: Changes in timestamps and *UTCOffset* fields when opening a file using TextEdit in MacOS Monterey/Mojave. The *LastAccessedTimestamp* is stored as UTC+5, even though the real timezone was UTC-5. The *CreateUtcOffset* is not changed, but the *LastModifiedUtcOffset* is changed to UTC+5, trying to convert LT from UTC-5 to UTC+5. In this case the last modified timestamp is inaccurate.

to UTC+5. This must fail since the Linux base did not use *UTCOffset* fields, and the UTC-5 assumption was wrong. We illustrate this in Figure D.7.

Linux (exFAT fuse) Base, experiment-0 (UTC+3)

Create	LastModified	LastAccessed
LT	LT	LT
02/03/2022 16:11	02/03/2022 16:11	02/03/2022 16:11

New UTC offset: UTC-5, then open using Notepad in Windows 10

Create	LastModified	LastAccessed
LT	LT	LT
02/03/2022 16:11	02/03/2022 16:11	02/03/2022 16:11

Figure D.8: Changes in timestamps and *UTCOffset* fields when opening a file using Notepad in Windows 10. Nothing was changed. In this case the *LastAccessedTimestamp* is inaccurate.

In Windows no change at all was registered when just opening files in Notepad. We illustrate this in Figure D.8.

D.4.4 Experiment D and E: Changing exFAT files on multiple OSes

The results are shown in Table D.8. We can see that Windows set the *LastModified10msIncrement* to 0x00 when changing the files in Windows, and the last modified and last access timestamps and the corresponding *UTCOffset* fields are updated. We did not see any traces of atomic storage in the Windows experiments using the command line or Notepad. However, an app that uses atomic storage (safe store) will leave traces in the exFAT file system when changing a file; for instance when we performed a small test using MS Word v. 16.0.16130.20218 we observed the original unallocated file, two unallocated temporary files, and the renamed new allocated file. In this context we also saw proof of file system tunneling, meaning the created timestamp was preserved in the new changed file even when using atomic storage. This is due to that the original file was unallocated and the temporary file was renamed to the same name as the original (Chen 2023). We did not observe this kind of usage in Notepad.

Linux (exFAT fuse driver) only changes the last modified timestamp and the *LastModified10msIncrement* (values 0x00 or 0x64), but not the *UTCOffset* fields when changing the files using the bash script for appending more text in each file. The last accessed timestamp was not changed. However, when using Gedit to change the file content in Linux, it set all timestamps to the change time using the local time of the computer and sets all *UTCOffset* fields to 0x00. This is because the Gedit application use atomic storage which on content change creates a new temporary file with a set of new timestamps. If this temporary file is successfully written, then the old file will be unlinked, and the temporary file will be renamed with the filename of the old unlinked file. The remnants of the delinked file may exist as a set of unallocated directory entries, or they may be overwritten. In the latter case the old metadata of the file is lost.

The *10msIncrement* fields are also updated. Since all timestamps are equal, it looks like the file was created at the time it actually was only changed. The observations are similar when using the Linux Ubuntu native exFAT driver, except that the *UTCOffset* fields are set to 0x80, and that not only 0x64 and 0x00 are used for the *10msIncrement* fields. For both Linux drivers, the original create timestamp may be lost.

MacOS also behaves differently if the content is changed using piping in a bash script, or if the files are changed by manually opening and changing the files in TextEdit. The latter will even try to set the *UTCOffset* for the created timestamp,

assuming the original timestamp was stored with the same timezone offset as the local computer (in our case UTC-5), which was an incorrect assumption in our case. This resulted in a changed created timestamp in addition to the changes of last modified and last accessed. We have not identified that these changes could be a result of atomic storage within TextEdit. The created timestamp is changed based on an assumption that the invalid UTC offset is equal to whatever local timezone *UTCOffset* is used by MacOS. We also observed that additional fork files were created when changing the files in TextEdit, but not when using the bash script. These fork files are pre-pended with `._` and may contain metadata information that describes which app was used to change the file. A fork is a named attribute used normally in HFS or APFS that contains a stream of data, and is similar to alternate data streams in NTFS (Wani et al. 2020).

OS	Action	Tool used	Timestamp	10msIncrement	UtcOffset	Observations
Windows	Change	»	LM and LA	LM (0x00)	LM and LA	400
Windows	Change	Notepad (manual)	LM and LA	LM (0x00)	LM and LA	100
MacOS	Change	»	LM and LA	LM	LM and LA	400
MacOS	Change	TextEdit (manual)	C*, LM and LA	LM	C, LM and LA	100
Linux (fuse)	Change	»	LM	LM (0x00 or 0x64)	Not changed	400
Linux (native)	Change	»	LM	LM	All is set to 0x80	400
Linux (fuse)	Change	Gedit (manual)	C, LM, and LA	C and LM (0x00 or 0x64)	All is set to 0x00	100
Linux (native)	Change	Gedit (manual)	C, LM, and LA	C and LM	All is set to 0x80	100

Table D.8: Experiment D and E Results - Changes in Timestamps, 10msIncrement and *UTCOffset* fields in the exFAT file directory entry when changing the files on Windows, MacOS or Linux. The C* means a special case where an invalid *UTCOffset* for the Created is interpreted incorrectly and then the Created timestamps is converted using the switching feature of MacOS.

D.4.5 10msIncrement fields

Another result we observed was the usage of the 10ms granularity fields; *Create10msIncrement* and *LastModified10msIncrement*. Based on the results in Table D.9 we can verify that Windows 10 does not update the *LastModified10msIncrement* on change, as described by Heeger et al. (2021). However, MacOS does update these fields. We also observed that Linux update these fields, and we observed that both *10msIncrement* fields either had the value 0x00 or 0x64 for the exFAT fuse driver. The latter value 0x64 is 100 in decimal, meaning in this context 1000 ms or 1 second. However, the native exFAT driver used by Ubuntu 20.04 updates the *10msIncrement* fields similar to MacOS.

D.4.6 Tool testing

Law enforcement require tools that give accurate results, and interpret timestamps correctly, else any incorrect results may impact a criminal case. Therefore, we will show how different Digital Forensic tools show timestamps from the exFAT file

OS	Create10msIncrement	LastModified10msIncrement	Observations
Windows	In use	Not used	400
Mac OS	In use	In Use	400
Linux	In use	In Use	400

Table D.9: Experiment Results - Usage of the 10ms granularity fields in the exFAT file directory entry when using Windows, MacOS or Linux.

system, in the context of the above mentioned experiments.

Autopsy

Base TZ (Index)	Type	Stored TZ	Stored Time	Autopsy TZ	Autopsy Time	Observations
Europe/Moscow (0)	Created	0xF4 (UTC-3)	23/02/2022 21:52	UTC-3	23/02/2022 17:52	100
America/Godthab (1)	Created	0x8C (UTC+3)	24/02/2022 03:53	UTC+3	24/02/2022 05:53	100
Atlantic/Azores (2)	Created	0x84 (UTC+1)	24/02/2022 01:53	UTC+1	24/02/2022 01:53	100
Europe/Oslo (3)	Created	0xFC (UTC-1)	23/02/2022 23:53	UTC-1	23-02-2022 21:53	100

Table D.10: Experiment Results - MacOS and Autopsy v. 4.19.3

When testing Autopsy we adjusted the timezone for each experiment in order to match the timezone used for storing the timestamp. The date/time should match between stored time and the time shown in Autopsy, and the results are shown in Table D.10. We can see that experiment index 2 matches where both timezones are using UTC+1. We also need to take into consideration that Autopsy was initially set to the timezone Europe/Oslo (UTC+1) in standard time when adding the forensic image. Therefore, we found that Autopsy interprets the exFAT timestamps using this initial timezone as the stored local time. In the other experiments we saw that Autopsy interpreted all stored timestamps as the initial local time (here UTC+1), and then tries to convert it to a timezone selected in Autopsy options view tab. This assumption about the current timezone was incorrect in most of our experiments. For instance, in order to change to UTC-3, Autopsy tries to subtract -4 from the stored timestamp, since it assumes the stored timestamp is given in UTC+1. For UTC+3 Autopsy adds 2 hours to get from UTC+1 to UTC+3. The only place it gives the same timestamp is when the UTC offset is the same as the assumed timezone. However, even in the latter case it is inaccurate, because Atlantic/Azores is using UTC-1 in standard time as seen in Table D.10.

When we added the Windows exFAT forensic image to Autopsy, we adjusted the initial value to Europe/Moscow (UTC+3). Autopsy tried to adjust the timezone based on the initial UTC+3 that it interpreted as the local time stored. This assumption is only correct for the Experiment-0 files.

FTK Imager

Base TZ (Index)	Type	Stored TZ	Stored Time	FTK TZ	FTK Time	Observations
Europe/Moscow (0)	Created	0xF4 (UTC-3)	23/02/2022 21:52	UTC+0	24/02/2022 00:52	100
America/Godthab (1)	Created	0x8C (UTC+3)	24/02/2022 03:53	UTC+0	24/02/2022 00:53	100
Atlantic/Azores (2)	Created	0x84 (UTC+1)	24/02/2022 01:53	UTC+0	24/02/2022 00:53	100
Europe/Oslo (3)	Created	0xFC (UTC-1)	23/02/2022 23:53	UTC+0	23/02/2022 00:53	100

Table D.11: Experiment Results - MacOS and FTK Imager v. 4.5.0.3

In Table D.11 we were not able to adjust the timezone shown by FTK Imager. Instead, the tool converted the timestamps to UTC+0. The conversions from stored timestamp to UTC+0 was correct. However, when adding the linux base image from the exFAT fuse driver with *UTCOffset* fields not valid, then the Created, Modified and Accessed are set to N/A (Not Applicable). The latter approach is fine, since it is infeasible to show the date and time when the UTC offset fields are not valid. However, showing the timestamps with a LT (Local Time) would have been better.

X-Ways Forensics

Base TZ (Index)	Type	Stored TZ	Stored Time	X-Ways TZ	X-Ways Time	Observations
Europe/Moscow (0)	Created	0xF4 (UTC-3)	23/02/2022 21:52	UTC-3	23/02/2022 21:52	100
America/Godthab (1)	Created	0x8C (UTC+3)	24/02/2022 03:53	UTC+3	24/02/2022 03:53	100
Atlantic/Azores (2)	Created	0x84 (UTC+1)	24/02/2022 01:53	UTC+1	24/02/2022 01:53	100
Europe/Oslo (3)	Created	0xFC (UTC-1)	23/02/2022 23:53	UTC-1	23/02/2022 23:53	100

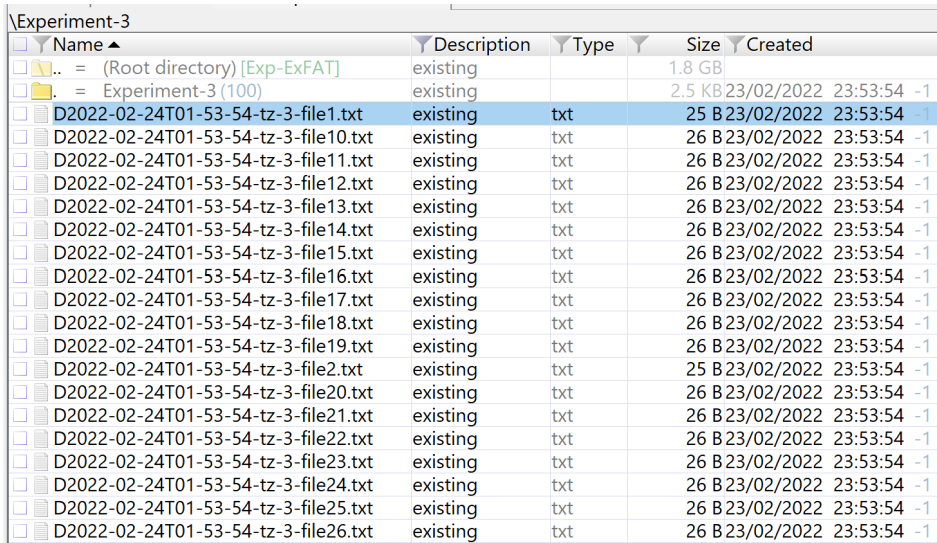
Table D.12: Experiment Results - MacOS and X-Ways Forensics v. 20.04 SR-4

Table D.12 shows the timestamps correctly using the same UTC offset as they were stored. X-Ways also displays the applied UTC offset after each timestamp, as shown in Figure D.9 and the hex representation of the first file in Figure D.10. X-Ways converts the exFAT timestamp correctly using any selected timezone. When X-Ways interprets *UTCOffset* fields with a mix of valid and invalid values, it tries to convert all values using the stored UTC offset to compute the selected timezone UTC offset. However, converting an invalid UTC offset value to a timezone UTC offset is based on an assumption about the previous stored UTC offset.

EnCase Forensic

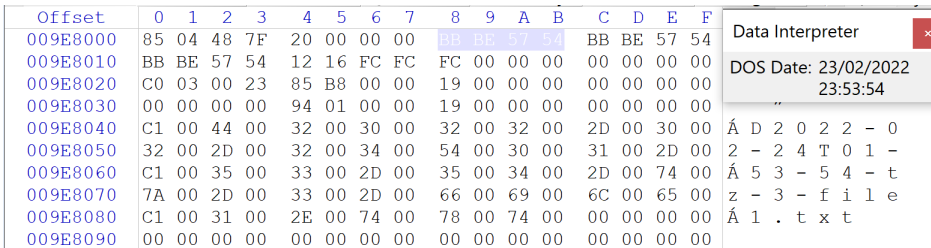
Base TZ (Index)	Type	Stored TZ	Stored Time	EnCase TZ	EnCase Time	Observations
Europe/Moscow (0)	Created	0xF4 (UTC-3)	23/02/2022 21:52	UTC-3	23/02/2022 21:52	100
America/Godthab (1)	Created	0x8C (UTC+3)	24/02/2022 03:53	UTC+3	24/02/2022 03:53	100
Atlantic/Azores (2)	Created	0x84 (UTC+1)	24/02/2022 01:53	UTC+1	24/02/2022 01:53	100
Europe/Oslo (3)	Created	0xFC (UTC-1)	23/02/2022 23:53	UTC-1	23/02/2022 23:53	100

Table D.13: Experiment Results - MacOS and EnCase Forensic v. 22.1



Name	Description	Type	Size	Created
.. = (Root directory) [Exp-ExFAT]	existing		1.8 GB	
. = Experiment-3 (100)	existing		2.5 KB	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file1.txt	existing	txt	25 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file10.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file11.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file12.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file13.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file14.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file15.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file16.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file17.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file18.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file19.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file2.txt	existing	txt	25 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file20.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file21.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file22.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file23.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file24.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file25.txt	existing	txt	26 B	23/02/2022 23:53:54 -1
D2022-02-24T01-53-54-tz-3-file26.txt	existing	txt	26 B	23/02/2022 23:53:54 -1

Figure D.9: ExFat timezones using stored UTC-1 offset for Experiment3 on MacOS and the X-Ways directory listing



Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
009E8000	85	04	48	7F	20	00	00	00	BB	BE	57	54	BB	BE	57	54
009E8010	BB	BE	57	54	12	16	FC	FC	FC	00	00	00	00	00	00	00
009E8020	C0	03	00	23	85	B8	00	00	19	00	00	00	00	00	00	00
009E8030	00	00	00	00	94	01	00	00	19	00	00	00	00	00	00	00
009E8040	C1	00	44	00	32	00	30	00	32	00	32	00	2D	00	30	00
009E8050	32	00	2D	00	32	00	34	00	54	00	30	00	31	00	2D	00
009E8060	C1	00	35	00	33	00	2D	00	35	00	34	00	2D	00	74	00
009E8070	7A	00	2D	00	33	00	2D	00	66	00	69	00	6C	00	65	00
009E8080	C1	00	31	00	2E	00	74	00	78	00	74	00	00	00	00	00
009E8090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

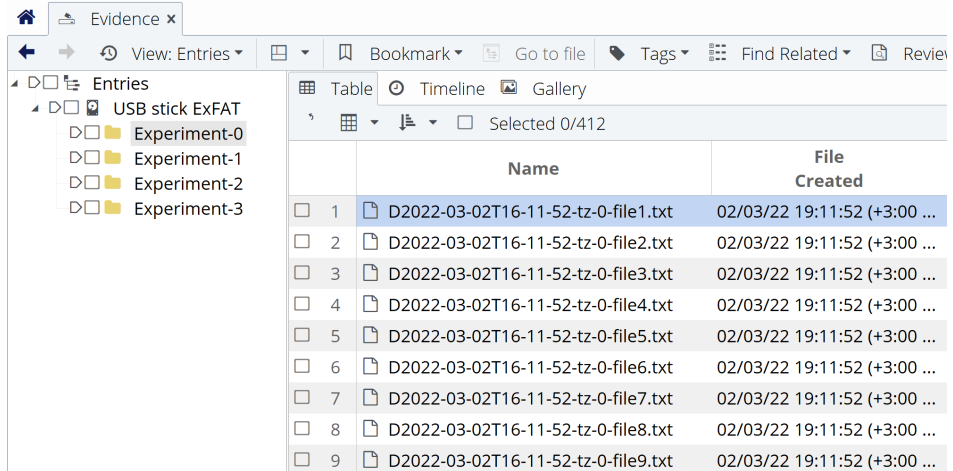
Figure D.10: ExFat timezones using stored UTC-1 offset for Experiment3 and the X-Ways for D2022-02-24T01-53-54-tz-3-file1.txt

EnCase shows the timestamps in the selected timezone taking the *UTCOffset* fields into consideration, as shown in Table D.13. When it comes to interpreting an exFAT filesystem created by the Linux Ubuntu exFAT fuse driver, EnCase interprets the stored timestamp as UTC+0 and tries to convert to the selected timezone, even though the *UTCOffset* fields have the 0x00 value (not valid). This is only accurate if the local time of the Linux computer was UTC+0, which it was not in all our experiments.

When there were mixed values in the *UTCOffset* fields, EnCase correctly showed timestamps where *UTCOffset* fields contained valid values, but failed if these values were invalid (0x00). For instance, Encase correctly showed the ones with value

0x80 using the selected timezone in EnCase, however the 0x00 value was wrongly interpreted as if the timestamps are stored using UTC+0.

Based on the experiments we can see that EnCase can be validated for exFAT as long as the *UTCOffset* fields are valid. If they are not valid, then EnCase seems to make an assumption about the UTC offset that may be wrong.



The screenshot shows the EnCase interface with a table of file entries. The table has columns for 'Name' and 'File Created'. The entries are numbered 1 through 9 and all have a creation time of 02/03/22 19:11:52 (+3:00 ...).

	Name	File Created
1	D2022-03-02T16-11-52-tz-0-file1.txt	02/03/22 19:11:52 (+3:00 ...
2	D2022-03-02T16-11-52-tz-0-file2.txt	02/03/22 19:11:52 (+3:00 ...
3	D2022-03-02T16-11-52-tz-0-file3.txt	02/03/22 19:11:52 (+3:00 ...
4	D2022-03-02T16-11-52-tz-0-file4.txt	02/03/22 19:11:52 (+3:00 ...
5	D2022-03-02T16-11-52-tz-0-file5.txt	02/03/22 19:11:52 (+3:00 ...
6	D2022-03-02T16-11-52-tz-0-file6.txt	02/03/22 19:11:52 (+3:00 ...
7	D2022-03-02T16-11-52-tz-0-file7.txt	02/03/22 19:11:52 (+3:00 ...
8	D2022-03-02T16-11-52-tz-0-file8.txt	02/03/22 19:11:52 (+3:00 ...
9	D2022-03-02T16-11-52-tz-0-file9.txt	02/03/22 19:11:52 (+3:00 ...

Figure D.11: ExFat timeszones using stored UTC+3 offset for Experiment0 from the Linux Base exFAT fuse image and the EnCase

We can see this interpretation in Figure D.11 where the local time stored was 02.03.2022 at 16:11 (UTC+3), but wrongly interpreted as UTC+0 because of the 0x00 values in the *UTCOffset* fields, and then EnCase adds 3 hours to convert to UTC+3, which is incorrect in this context.

D.5 Discussion

The observations show that exFAT on Windows uses the local timezone offset including any daylight settings without switching signs, accurately storing the timestamp using the UTC offset of the local time of the computer. The MacOS experiments show that timestamps are not stored in the local time, instead it converts the UTC offset by switching signs. The Linux experiments using exFAT fuse driver shows that the timezone *UTCOffset* fields are not in use, and that the timestamp is stored using localtime, while the Linux native driver and *UTCOffset* fields are set to 0x80 (UTC+0) and the timestamps are stored using UTC+0 for the native exFAT driver.

The implementation used by MacOS will not make timestamps inaccurate when

showing the files from these experiments in Windows. Windows File Explorer interprets the exFAT file system correctly, meaning File Explorer will take the time-zone *UTCOffset* field into consideration before converting it to the local time used by the local computer. An example from Experiment-3 is shown in Figure D.12 for File Explorer. The same is true when mounting an exFAT storage device from Windows to MacOS, except that MacOS changes the Last Accessed timestamp and the *LastAccessedUtcOffset*. However, both MacOS and Windows take the *UTCOffset* fields into account and show them in the local time of the computer.

When it comes to Linux Ubuntu 20.04 exFAT fuse driver, no tools examined in this experiment will know what the timezone offset of the local time was for each file created. If the files are changed by using Gedit in Linux the *UTCOffset* are set to 0x00 and all timestamps are changed to the time of the update. This is due to the atomic storage features used by Gedit. Therefore, the original created date may be lost if no remnants of previous metadata entries are found. Linux Ubuntu 20.04 native exFAT driver store the time in UTC+0 no matter what the timezone was, and this is an implementation where the knowledge of the local time of the computer is not preserved. It is not necessarily an incorrect method, and it does not impact how the timestamps are presented in other OSes and in Digital Forensic tools. However, they are not following the specifications (Microsoft 2021b).

Digital Forensic tools interpreting the timestamps should describe that the timestamps are stored as the localtime whenever the *UTCOffset* fields are not in use, and investigators cannot assume anything about which timezone was in use for a particular file when using the Linux exFAT fuse driver. If the *UTCOffset* fields was not in use, any change of timezone using a Digital Forensic tool should not change the time shown, but continue using the local time. However, if *UTCOffset* fields are valid, then Digital Forensic tools should change to the selected timezone utilising the necessary computation based on the stored timestamp and *UTCOffset* fields. Further, it should not be assumed that the *UTCOffset* fields only use one timezone offset.

If we know that an exFAT storage device has only been used on a MacOS, we can describe the local time of the computer (UTC offset for the timezone and any daylight settings) by switching the sign again. We can also find the last registered MacOS local computer UTC offset used by checking the *LastAccessedUtcOffset* field. On the other hand it will be difficult to assess if the timezone offset was initially set by a Windows or a MacOS computer. For instance, a MacOS using UTC-1 will store the timestamp in UTC+1, and a Windows computer using UTC+1 will store timestamps in UTC+1. In this scenario we have *UTCOffsets* of 0x84 on all timestamps, though they were actually running on two different timezones.

Name	Date created	Date modified	Date accessed
D2022-02-24T01-53-54-tz-3-file1	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file2	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file3	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file4	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file5	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file6	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file7	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file8	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file9	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file10	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file11	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file12	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file13	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file14	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file15	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file16	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file17	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53
D2022-02-24T01-53-54-tz-3-file18	24/02/2022 01:53	24/02/2022 01:53	24/02/2022 01:53

Figure D.12: ExFat timestzones using Europe/Oslo timezone (UTC+1) for Experiment-3 and the Windows 10 computer.

The existence of a *.fsevents* and *.Spotlight-V100* directory within the root directory is an indication of MacOS usage, while existence of a *System Volume Information* directory within the root directory is an indication of Windows usage.

If only mounting and dismounting a USB storage device with exFAT file system on a MacOS, then the *LastAccessedUtcOffset* field will be updated and stored using the switching signs method on every file on the device. However, the Created and Modified timestamps and their corresponding *UTCOffset* fields will not be updated. Therefore, it is common that a storage device used on both on Windows and MacOS will include timezone offsets that deviate within the same directory entry, even when using the same timezone. It is also important to note how easy it is to change timestamps unintentionally by connecting a storage device to a MacOS without using a write blocker.

D.5.1 Rules for updating timestamps

Timestamp	Specs	Driver compliance
CreateTimestamp	On creation	Windows 10
LastModifiedTimestamp	Modifying cluster content	Windows 10, Linux
LastAccessedTimestamp	Modifying or reading cluster content	MacOS

Table D.14: Rules for updating timestamps - compliance

Table D.14 shows which operating system exFAT driver complies with the ex-

FAT specifications for updating the different timestamps. The MacOS and Linux changes the timestamp for creation when using *TextEdit* or *Gedit* to change the content of a file. The latter because of atomic storage utilised by the application or its library. The former due to misinterpreting invalid UTCOffsets. This means the timestamps are changed by the driver when using MacOS, and it may already be inaccurate before parsing and interpretations of Digital Forensic tools. Most of the drivers update the last modified on change, but MacOS may update the last modified on open only. The last accessed timestamp is updated on open for all exFAT drivers except on Windows when using *Notepad* to open files. In Linux when using a bash script to append content the last accessed is not updated for both exFAT drivers, only the last modified timestamp.

This unequal behaviour impact the investigation, and therefore it must be emphasized that it is important to understand which OS driver has been used in order to interpret the findings correctly.

D.5.2 10ms granularity

Even though the exFAT specifications describe that the *LastModified10msIncrement* field should be updated when updating any clusters used by the stream extension directory entry, or when changing the *ValidDataLength* or *DataLength* fields ([Microsoft 2021b](#)), we observed that this was not implemented in Windows and implemented in MacOS.

The stego-only approach proposed by [Heeger et al. \(2021\)](#) will fail if the storage device is mounted and files are updated using MacOS or Linux, since this approach is using the *LastModified10msIncrement*. However, their exHide approach will work since they are not using this field and are utilising only unallocated directory entries.

D.5.3 Patterns

The three operating systems used in our experiments show distinct patterns which can be used to identify the OS used for a particular exFAT storage device. The most clear pattern is when the *UTCOffset* fields have the 0x00 value, meaning it is used by the Linux exFAT fuse driver. The other Linux exFAT fuse pattern is that the *10msIncrement* fields have the value 0x00 or 0x64. We should not see any System Volume Information directory or *.fsevents* and *.SpotLight-V100* directories if the USB storage is only used on Linux. We only know that the local time is used to store the timestamps when using the exFAT fuse driver, we do not know which timezone was used. The Linux exFAT native driver uses 0x80 (UTC+0) always, but other OSes using the GMT timezone will also use 0x80, and therefore this is not a good pattern to identify Linux.

MacOS uses all *UTCOffset* fields and both *10msIncrement* fields. In addition it creates the directories *.fseventsd* and *.Spot-Light-V100*. If no *UTCOffset* fields are 0x00 and the System Volume Information directory is not present in the root directory, then we know MacOS has been used. However, the native Linux driver may also have been used, but in this context if all files are using another timezone than GMT, then we know MacOS has been used. Another sign is the usage of fork files when using GUI apps to change files. The latter is very interesting since we can see which app was used to change the files. When only MacOS has been used, we can switch the stored *UTCOffset* sign to find the local time of the computer used when creating or updating a file.

Windows updates all *UTCOffset* fields and *Create10msIncrement*, and the *LastModified10msIncrement* is set to 0x00 on creation. In addition the directory System Volume Information is created in the root directory. On change the last modified and last accessed timestamps are updated, and the last modified is updated for the *10msIncrement* field, which is set to 0x00. If no *UTCOffset* fields have the value 0x00, and the directories *.fseventsd* and the *.SpotLight-V100* are not present, and the System Volume Information directory is present, and all files have 0x00 for the *LastModified10msIncrement*, but uses the *Create10msIncrement*, then we know that Windows has been used. When Windows is the only OS used, then we can find the local time used by the computer by using the *UTCOffset* fields.

D.5.4 Challenges

The MacOS exFAT driver will try to update the Create timestamp when changing a file manually using TextEdit. The driver makes an assumption that the timezone must be equal the local time of the Mac computer if it finds an invalid value in the *UTCOffset* fields. This may or may not be true, and if wrong will effectively change the created date to a wrong time.

Linux (both drivers) changed all timestamps when changing files using *Gedit* due to the atomic storage features implemented by *Gedit* or a library, using the local time when the change happened (3 equal timestamps). A text document created a year ago, will get a new set of equal timestamps for create, last modified, and last accessed when changing the file using *Gedit*. If the digital forensic investigator identify that the exFAT storage device has been used on Linux, we can only identify previous metadata if remnants of previous directory entry sets are found.

In Experiment E we manually changed 100 files in the directory Experiment-0 using *Gedit* and the Ubuntu Linux exFAT native driver. The base image was from the exFAT fuse driver. We observed one unallocated temporary file—*.goutputstream-8KK4I1*—which corresponds to the metadata for the allocated file *D2022-03-02T16-*

11-55-tz-0-file100.txt, and we also found the latter file in an unallocated directory entry with the previous timestamps. We found the unallocated file *.goutputstream-A6L5811* in the same directory, which corresponds with the allocated file *D2022-03-02T16-11-55-tz-0-file1.txt*, but not the corresponding unallocated file. In addition, we found a single unallocated File Name Directory Entry with last part of its name; *99.txt* in this directory. This means that 98 of the 100 changed files lost all their original metadata due to atomic storage implemented by the *Gedit* application. One file lost all previous metadata, except the last part of the name.

D.5.5 Tools

In this section we discuss if different tools can be validated for law enforcement usage. With tool validation we mean if the tool is appropriate for its intended usage (ISO 2017). Our aim is that the tool developers improve tools where we have found inaccuracy. This also means that in future releases of these tools, the interpretation may have been improved.

Autopsy

Autopsy interpreted that exFAT has stored the timestamp as the local time initially set when adding the forensic image into Autopsy, and does not consider the timezone offset in the directory entry. If the initial given local time does not match the stored local timezone *UTCOffset* for each timestamp, then it will yield erroneous results. Setting the initial local timezone correctly requires the DF investigator to verify the timezone *UTCOffset* manually in a hex viewer, but Autopsy cannot support files with multiple timezone offsets stored on the same file system.

Based on these findings we assess that Autopsy v. 4.19.3 (Windows version) cannot be validated for interpreting exFAT timestamps.

FTK-imager

FTK Imager displays the timestamps in UTC+0 by taking the *UTCOffset* fields into consideration. FTK Imager can be validated for interpreting exFAT timestamps as long as the timestamps shown are interpreted as UTC+0 by the digital forensic investigator. It is not suitable to use for a storage that have been using the Linux exFAT fuse driver, since it will not show any timestamps because of the non valid *UTCOffset* field values.

Another issue is that FTK Imager v 4.5.0.3 (version used in this paper) and 4.7.1.2 (a newer version) do not show all directories, the *.fseventsd* and the *.SpotLight-V100* are not shown.

These versions of FTK Imager cannot be validated for interpreting the exFAT file

system as long as it does not show all files or directories.

X-Ways

X-Ways displays the timestamps correctly in the timezone selected by the investigator, and it takes the stored *UTCOffset* fields into consideration when adjusting the time to the selected timezone. X-Ways also show the UTC offset used after each timestamp. X-Ways can be validated for interpreting exFAT timestamps as long as all *UTCOffset* fields are the same within the same directory entry.

If the *CreateUtcOffset* is 0x00 and the *LastModifiedOffset* and *LastAccessedUtcOffset* is a valid UTC offset, then it will try to show all timestamps in the selected timezone. However, it cannot know the local time of the timestamp using *UTCOffset* value 0x00, and any conversion must be based on assumptions. This is especially important when there are mixed *UTCOffset* values, where one or more contain the value 0x00.

EnCase

EnCase displays timestamps correctly if the *UTCOffset* fields have valid values. The assumption made by EnCase is that the *UTCOffset* field value 0x00 means UTC+0, but this is a wrong assumption. If the stored timestamp was stored using UTC+3, then the accuracy is 3 hours off.

EnCase can be validated for interpreting exFAT timestamps when the *UTCOffset* fields contain valid values.

D.6 Conclusion and Further Work

- How do current exFAT implementations store timestamps?

In Windows 10 the exFAT specifications ([Microsoft 2021b](#)) are followed by storing timestamps using the UTC offset of the local computer, including any daylight settings. MacOS has their own method of storing exFAT timestamps that switches the UTC sign and store the time accordingly. Linux Ubuntu 20.04 when using the exFAT fuse driver sets the *UTCOffset* fields to 0x00, which means the fields are not in use. Linux Ubuntu 20.04 native exFAT driver uses the *UTCOffset* fields, but sets them always to 0x80 (UTC+0). We also observed that graphical user interface apps could update the create timestamps in Linux to the time of modification time due to atomic storage features implemented in the application or by a library used by the application, or adjust it in MacOS making assumptions about the invalid UTC offset previously registered.

- Can we use the UTC offset stored in a directory entry to describe the local

time of the computer?

If the exFAT storage device has only been used on MacOS computers, we can switch the sign of the *UTCOffset* fields and find the local UTC offset used by the MacOS computer for a specific timestamp. If the storage device has only been used on a Windows computer, we can interpret the local UTC as equal to the *UTCOffset* field for a specific timestamp. However, if mix usage between Windows and MacOS then it may be more difficult. In Linux it is not possible to know what UTC offset were used, but still the local time is used for storing the timestamps when using the exFAT fuse driver, and UTC+0 when using the native exFAT driver. We may be able to find remnants from the previous metadata, but these are not always available.

We were not able to falsify our null hypotheses, because 33 percent of the valid *UTCOffset* observations in Experiment A showed that timestamps were stored using UTC+0, and only 67 percent were stored related to the local time. This means our main hypothesis is wrong for Linux Ubuntu 20.04 native exfat driver, but correct for the Windows and the MacOS exFAT driver.

- Do current forensic tools interpret exFAT timestamps differently?

The four different ways of storing exFAT timestamps between MacOS, Windows and Linux do impact tools that take the timezone *UTCOffset* fields into consideration (FTK Imager, X-Ways, and EnCase). Unfortunately, Autopsy does not consider the *UTCOffset* fields stored in the directory entry and uses the given timezone when adding the forensic image as the local time used for storing the timestamps. EnCase does not interpret exFAT with a non-valid *UTCOffset* field correctly, but make an assumption that the value 0x00 means UTC+0, which is incorrect in most cases. FTK Imager convert all timestamps to UTC+0 taking the *UTCOffset* fields into consideration. If one or more of the *UTCOffset* fields contains a non valid value, it only shows the timestamps for the valid *UTCOffset* fields. Unfortunately, FTK Imager does not show all directories. X-Ways take *UTCOffset* fields into consideration, and if these fields are all invalid it describes that local time (LT) is being used. However, X-Ways does not handle a mix of valid and invalid *UTCOffset* values, it then makes assumptions about the non-valid value in order to convert all timestamps of a file to the selected timezone.

It is not just the tools that may interpret exFAT differently, but also the different file system drivers may incorrectly change timestamps. MacOS makes an assumption that the created time uses the local time of the MacOS when the *UTCOffset* fields

are invalid, and updates the Create time when changing a file using TextEdit by switching the UTC offset and storing the time accordingly. If the assumption is wrong, then the created time is stored incorrectly.

Finally, the conclusion is that exFAT timestamps stored by file systems may be unreliable, especially when used on multiple OSes, and that Digital Forensic tools may even interpret reliable dates in an unreliable way. We recommend using X-Ways to interpret exFAT, and use patterns to identify which OS has been used in order to make an accurate interpretation of the timestamps. In addition, the digital forensic expert should be aware that some applications may implement atomic storage of files, which effectively modifies all timestamps by creating a new file on any change. The previous file metadata is unallocated and often overwritten.

As further work we suggest observing other file systems that can be used on multiple OSes, to assess if the drivers store timestamps equally, and if Digital Forensic tools interpret the timestamps accurately and reliably. Further, we recommend law enforcement to reassess criminal cases where exFAT and timestamps have been an important evidence to make sure innocent persons have not been convicted based on misinterpreted timestamps. We also suggest studying the impact of atomic storage have on accuracy of timestamps using different file systems.

Acknowledgements

We would like to thank the anonymous reviewers for their suggestions and Maxim Suhanov who pointed out mistakes in our original version of this article, and also made us aware of that recent versions of FTK Imager does not display all directories.

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project "Ars Forensica - Computational Forensics for Large-scale Fraud Detection, Crime Investigation & Prevention", grant agreement 248094/O70.

Bibliography

- Bretel, J. (2017). Operating systems and file systems compatibility. <https://www.7dayshop.com/blog/operating-systems-and-file-systems-cross-compatibility-windows-apple-linux-playstation-xbox-android/>.
- Chen, R. (2023). The apocryphal history of file system tunnelling. <https://devblogs.microsoft.com/oldnewthing/20050715-14/?p=34923>, visited 2023-03-13.
- European Committee for Standardization (2022). Cen Workshop Agreement (CWA): Requirements and Guidelines for a complete end-to-end mobile forensic investigation chain. https://www.cencenelec.eu/media/CEN-CENELEC/CWAs/RI/cwa17865_2022.pdf.
- European Court of Human Rights (2021). European convention on human rights. https://www.echr.coe.int/Documents/Convention_ENG.pdf.
- Hamm, J. (2009). Extended fat file system. <https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf>, visited 2018-09-16.
- Heeger, J., Yannikos, Y., and Steinebach, M. (2021). Exhide: Hiding data within the exfat file system. In *The 16th International Conference on Availability, Reliability and Security, ARES 2021*, New York, NY, USA. Association for Computing Machinery.
- Ieong, R. S. (2006). Forza – digital forensics investigation framework that incorporate legal issues. *Digital Investigation*, 3:29 – 36. The Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06).

- ISO (2017). ISO/IEC 17025:2017 General requirements for the competence of testing and calibration laboratories. <https://www.iso.org/standard/66912.html>.
- Knight, J. C. and Leveson, N. G. (1986). An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109.
- Lyle, J. R. (2010). If error rate is such a simple concept, why don't i have one for my forensic tool yet? *Digital Investigation*, 7:S135–S139. The Proceedings of the Tenth Annual DFRWS Conference.
- Ma, G., Wang, Z., and Cheng, Y. (2015). Recovery of evidence and the judicial identification of electronic data based on exfat. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 66–71.
- Microsoft (2009). Us patent us 20090164440 a1. <https://ppubs.uspto.gov/pubwebapp/>, visited 2022-03-01.
- Microsoft (2021a). Default cluster size for NTFS, FAT, and exFAT. <https://support.microsoft.com/en-us/topic/default-cluster-size-for-ntfs-fat-and-exfat-9772e6f1-e31a-00d7-e18f-73169155af95>.
- Microsoft (2021b). exFAT file system specification. <https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification>.
- Munegowda, K., Raju, G., and Raju, V. M. (2014). Directory compaction techniques for space optimizations in exfat and fat file systems for embedded storage devices. *International Journal of Computer Science Issues (IJCSI)*, 11(1):144.
- Munegowda, K., Raju, G. T., and Raju, V. M. (2012). Cluster allocation strategies of the exfat and fat file systems: A comparative study in embedded storage systems. In Kumar M., A., R., S., and Kumar, T. V. S., editors, *Proceedings of International Conference on Advances in Computing*, pages 691–698, New Delhi. Springer India.
- Neale, C., Kennedy, I., Price, B., Yu, Y., and Nuseibeh, B. (2022). The case for zero trust digital forensics. *Forensic Science International: Digital Investigation*, 40:301352.

- Nordvik, R. (2022). Exfat forensic images for timestamp testing. <https://data.mendeley.com/datasets/krjsmdc65h/2>, visited 2023-03-13.
- Nordvik, R. and Axelsson, S. (2023). Corrigendum to “it is about time—do exfat implementations handle timestamps correctly?” [forensic science international: Digital investigation 42–43 (2022) 301476]. *Forensic Science International: Digital Investigation*, 45:301542.
- Nordvik, R., Stoykova, R., Franke, K., Axelsson, S., and Toolan, F. (2021). Reliability validation for file system interpretation. *Forensic Science International: Digital Investigation*, 37:301174.
- Scanlon, M. (2016). Battling the digital forensic backlog through data deduplication. In *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*, pages 10–14.
- Schullich, R. (2009). Reverse Engineering the Microsoft Extended FAT File System (exFAT). <https://www.giac.org/paper/gcfa/570/reverse-engineering-microsoft-exfat-file-system/106672>, visited 2022-03-01.
- Suhanov, M. (2022). Do researchers handle exfat volumes correctly? <https://dfir.ru/2022/12/18/do-researchers-handle-exfat-volumes-correctly/>, visited 2023-03-07.
- USB Memory Direct (2022). Do I Need to Format a New USB Flash Drive? <https://www.usbmemorydirect.com/blog/need-format-new-flash-drive/>.
- Vandermeer, Y., Le-Khac, N.-A., Carthy, J., and Kechadi, T. (2018). Forensic analysis of the exfat artefacts.
- Wani, M. A., AlZahrani, A., and Bhat, W. A. (2020). File system anti-forensics – types, techniques and tools. *Computer Fraud & Security*, 2020(3):14–19.

ISBN 978-82-326-7824-2 (printed ver.)
ISBN 978-82-326-7823-5 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)



NTNU

Norwegian University of
Science and Technology