# Improving Foundations of File Recovery

## A Digital Forensics Perspective on File Fragmentation, Timestamps, and JPEG Validation

Vincent van der Meer

Dit proefschrift komt voort uit een Promotiebeurs voor Leraren met projectnummer 023.012.047, gefinancierd door de Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO).

Gedrukt door: ProefschriftMaken | proefschriftmaken.nl
Omslagontwerp: Wendy Schoneveld | `@wenzidee`
ISBN: 978-94-6510-001-2

Dit boek is gedrukt op gerecycled, $CO_2$-neutraal papier.

# Improving Foundations of File Recovery

**A Digital Forensics Perspective on File Fragmentation, Timestamps, and JPEG Validation**

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Open Universiteit
op gezag van de rector magnificus
prof. dr. Theo Bastiaens
ten overstaan van een door het
College voor promoties ingestelde commissie
in het openbaar te verdedigen

op 5 september 2024 te Heerlen
om 16:00 uur precies

door

Vincent van der Meer

geboren op 8 januari 1982
te Delft

# Voorwoord

*'Wij denken dat je het kunt.'*

Deze woorden bleven bij me resoneren. Zojuist had ik met Roger, mijn lector, gesproken over de vacature van PhD Researcher & Docent in Digital Forensics. Hij wilde weten of ik de vacature had gezien en had overwogen. Uiteraard had ik die gezien: die had direct mijn interesse gewekt! Een PhD-traject was de uitdaging waar ik naar op zoek was, en het forensische vakgebied intrigeerde me enorm. Jeroen, een beoogd begeleider van het traject, had ik ook al leren kennen, en tussen ons was er direct een klik geweest. Dus waarom had ik dan nog niet gereageerd op de vacature?

Hoe gaaf een promotietraject in digital forensics ook mocht lijken, ik moest ook realistisch zijn over wat me te wachten stond. Het vakgebied was nieuw voor mij en de slagingskansen voor buitenpromovendi zijn ronduit laag. En wanneer je het parttime onderzoek moet combineren met het docentschap én een mooi gezin, snap je misschien waar mijn terughoudendheid vandaan kwam.

Ondanks allemaal redenen om het niet te doen, ben ik toch het avontuur aangegaan. Ik ben begonnen aan een uitdaging waar ik de omvang nog niet van kon overzien, dankzij dat mooie blijk van vertrouwen.

# Dankwoord

Allereerst wil ik graag mijn dank uitspreken aan de leden van de beoordelingscommissie voor hun tijd en waardevolle feedback die hebben bijgedragen aan de verbetering van dit proefschrift.

Marko, jij was mijn promotor bij de start van dit traject. Mede dankzij jouw steun heb ik een succesvolle subsidieaanvraag gedaan en is mijn promotietraject uit de startblokken geschoten. Als je gezondheid het had toegelaten, weet ik zeker dat ik met jou de eindstreep zou hebben bereikt.

Harald, jij hebt de rol van promotor overgenomen. Dat heb je op een zorgvuldige en integere manier gedaan. Je hebt een belangrijke inspanning en bijdrage geleverd aan het eindresultaat. Daar ben ik je dankbaar voor.

Hugo, mijn dagelijks begeleider en co-promotor. Vanaf de start ben je mijn begeleider geweest. Ik heb je leren kennen als hartelijk persoon met een gezellige Brabantse inslag, en als iemand die de lat hoog legt. Werkend aan het onderzoek hebben we niet alleen veel dagen, maar ook veel avonden samen doorgebracht. Op kantoor, aan mijn keukentafel en aan jouw keukentafel. Het commitment dat je aan het promotietraject (en dus ook aan mij) hebt gegeven was groot. Jouw credo was: als jij hard werkt, werk ik net zo hard mee. En dat heb je waargemaakt. We hebben geknald! En het is zeker niet altijd makkelijk geweest, maar je hebt me ook gesteund op belangrijke momenten. Zo had ik je ineens aan de lijn voorafgaand aan een conferentiepresentatie, of wist me meteen te vinden als we een (harde) *reject* op een ingediend artikel ontvangen hadden. Dank je wel voor onze jaren van samenwerking, ik kijk daar met plezier op terug.

Dan is het tijd om mijn andere begeleider en co-promotor, Jeroen, te bedanken. Met jouw aanstelling als lector in Digital Forensics, was je natuurlijk een beetje de aanstichter van dit traject. Ik ben blij dat onze paden elkaar hebben gekruist. We hebben in de afgelopen jaren veel mogen samenwerken, en ik zie dat als een groot voorrecht. Ik heb veel van je geleerd en genoten van je beschouwingen over de gebeurtenissen in de wereld. Je hebt een aanstekelijk optimisme. Zelfs wanneer we in de diepste krochten van het JPEG-formaat hopeloos verstrengeld waren geraakt, was er nooit enige vorm van twijfel bij je te bespeuren. Je had een rotsvaste overtuiging dat deze problemen oplosbaar waren, mits we het onderwerp maar écht konden doorgronden. Pas in die periode leerde ik de echte betekenis daarvan kennen, en dat is een bijzondere, waardevolle ervaring geweest. Ik had nooit gedacht dat ik de resultaten kon boeken die we geboekt hebben.

Harm, jij was mijn begeleider en linking pin vanuit het Nederlands Forensisch Instituut. Ik heb onze vrijdagochtendgesprekken zeer kunnen waarderen. Je bracht een belangrijke forensische invalshoek aan op mijn onderzoek, en tegelijkertijd moedigde je me aan om fundamenteel of theoretisch onderzoek niet uit de weg te gaan. Als ik twijfel had bij een onderzoeksonderwerp, kon jij juist overtuigd zijn dat het meerwaarde had. Je steun en inzichten hebben me vaak geholpen om door te zetten.

Ook wil ik graag Dion bedanken, die mij als teamleider Forensische Software Engineering hartelijk heeft ontvangen. En natuurlijk de andere NFI-collega's, die altijd vriendelijk, geïnteresseerd en meedenkend zijn geweest.

Tijdens mijn promotieonderzoek heb ik het voorrecht gehad om met een groot aantal getalenteerde, enthousiaste (en inmiddels voormalig) studenten te mogen samenwerken, in de vorm van stage- of afstudeerbegeleiding. Daarom een hartelijk dank je wel aan Dewi, Jasper, Johannes, Mart, Nick, Remco, Robbert, Yvonne en Zowie. In het bijzonder wil ik Guy, Jelle en Laurent bedanken. Jullie afstudeerwerk, samen met jullie extra inspanningen na de afronding van de projecten, heeft geleid tot een publicatie waar jullie co-auteur van zijn geworden. Een fantastisch resultaat!

Wat mij brengt naar mijn collega's van Zuyd Hogeschool. Roger, dit traject is gestart met jouw blijk van vertrouwen, en dat vertrouwen ben ik al die jaren blijven voelen. Je hebt mij als lector de ruimte gegeven om mijn traject naar eigen inzicht vorm te geven en te organiseren. Terugkijkend denk ik dat dat een heel belangrijke succesfactor voor mijn traject is gebleken.

Peter, je hebt me als faculteitsdirecteur niet alleen aangemoedigd om dit traject aan te gaan, ook gedurende de hele rit kon ik op je steun rekenen als ik daar om vroeg. Out-of-the-box-oplossingen lijk je zo uit je mouw te kunnen schudden, met zowel een lach als een serieuze ondertoon. Ik wist dat ik op je kon rekenen als dat nodig was. Dank je wel daarvoor!

En er zijn nog veel meer Zuyd-collega's die mij gesteund hebben. Wanneer er een paper deadline naderde (en ik elk beschikbaar uur aan het schrijfproces besteedde) of wanneer ik weg was voor een conferentiebezoek, dan werden er voor mij toetsen en casussen nagekeken en lessen overgenomen. Zonder morren, altijd, door iedereen. Wauw! Ik heb ook altijd een mooie interesse gevoeld van mijn collega's: met plezier heb ik jullie bij de koffieautomaat deelgenoot kunnen maken van mijn PhD-avonturen, en hebben we samen elk geaccepteerd paper kunnen vieren met vlaai. Ik dank jullie allemaal voor jullie steun en interesse, en in het bijzonder wil ik graag noemen: Chris, Hans, Isolde, Josianne, Kees, Luuk, Marc, Marlou, Maud, Miguel, Roel, en Roland.

Een bijzonder woord van waardering voor Eddy. Je hebt een visualisatie ontworpen en geautomatiseerd, wat vele ontwerpiteraties heeft gekost, maar waarbij je altijd bereid was mee te werken. Ik ben er trots op dat jij co-auteur bent van een van de papers.

Nog een bijzondere vermelding wil ik voor Bianca maken: naast dat ik heerlijk met je heb kunnen lachen, heb jij mij alle administratieve ondersteuning geboden die al die jaren bij dit PhD-traject kwam kijken. Voor dit niet-standaard traject kwamen we nogal gauw buiten de gebaande paden uit. Voor jou leek dat nooit een probleem, jij wist van alles voor elkaar te boksen. Dank je wel voor al je inzet!

Als laatste Zuyd-collega een speciaal woord voor Marcel. Er zijn denk ik weinig mensen die echt snappen wat een traject als dit betekent. Ik heb het daarom altijd heel fijn gevonden dat we (als PhD'ers en lotgenoten) steun en begrip bij elkaar konden vinden. We hebben onze hoogte- en dieptepunten van dit traject kunnen delen samen. Dank je wel daarvoor! Wat is het ongelofelijk gaaf dat we beiden ook de eindstreep hebben weten te bereiken.

Dan wil ik nog mijn oud-collega's Geny, Laurent en Emiel bedanken voor de fijne samenwerking die we hebben gehad, en voor hun steun aan mij om dit promotietraject te beginnen.

Dan richt ik mij graag tot mijn vriendenkring, waar ik niet iedereen kan noemen, maar de volgende personen zeker niet mogen ontbreken. Aukje, dank je wel voor je nuchtere, vrolijke kijk op het leven. Ik heb onze koffiemomentjes zeer kunnen waarderen. Heleen, wat een lol hebben we gehad, en wat fijn dat we elkaar na al die jaren nog weten te vinden. Al was het maar met een (veel te laat) felicitatie-berichtje! Monique, jij bood me een luisterend oor, en we hebben mooie gesprekken gevoerd. Daar ben ik je dankbaar voor. Jodi, wat ben ik blij met jouw vasthoudendheid. Steevast vroeg jij naar hoe het met m'n PhD ging. En 'prima' was daarbij geen bevredigend antwoord. Dank je wel voor je steun en vriendschap. Stefan, je bent een heerlijke gozer met een flinke bak humor. Daar ga ik voorlopig geen genoeg van krijgen! Jody, er is geen onderwerp waar wij niet samen goed mee uit de voeten kunnen. Heerlijk om met jou altijd wel een boom op te kunnen zetten. We hebben nog wel een bos of wat in het vooruitzicht! En over al mijn vriendschappen kan ik zeggen: ik kijk met een glimlach en dankbaarheid terug, en met plezier vooruit! We gaan nog een hoop gave, mooie momenten hebben samen!

Dan wil ik mijn ouders graag bedanken. Voor het warme nest waarin ik ben opgegroeid, waarin ik nieuwsgierig kon zijn en de vrijheid voelde om de wereld te gaan verkennen. En mijn broer en zus, voor onze mooie tijd toen en nu, en voor alle talloze potjes Mario en Mario Kart die we samen gespeeld hebben.

Vivian, thank you for proofreading some of my papers, and for getting me on that plane to Sydney. That was an experience I will never forget.

En dan ben ik aangekomen bij mijn eigen gezin. Allereerst mijn prachtige kinderen: Pepijn, Hidde, en Feis. We hebben al wat mooie avonturen beleefd samen. En nu mijn eigen PhD-avontuur erop zit, is er weer ruimte voor wat nieuws. Wie weet welke Zelda-games we nog gaan spelen, of welke bergen we nog gaan beklimmen. Ik weet wel dat ik er zin in heb! En weet dat ik jullie altijd zal steunen op jullie pad, waarheen dat ook zal leiden.

Tot slot, Dieuwertje, wil ik jou bedanken. Dit PhD-traject heeft mij, en ons, meer gekost dan ik had verwacht. Zonder jouw steun zou ik de eindstreep niet hebben bereikt. Ik ben onwaarschijnlijk trots dat we samen de eindstreep hebben weten te behalen. Dank je wel voor je steun en je liefde. Het is een voorrecht om jou mijn partner te mogen noemen.

Vincent van der Meer
Landgraaf, juni 2024

*A PhD is not so much an intelligence test,*
*but a test of your endurance and adaptability,*
*along with your creativity and ability to be systematic*
*in the face of chaos.*

# Contents

# Chapter 1

# Introduction

The field of digital forensics encompasses numerous specializations, one of which is the retrieval and restoration of deleted computer files. Potentially, it is precisely these deleted files that can contain crucial evidence, and in the case of digital imagery, significantly contribute to the identification of perpetrators and victims. The successful recovery of deleted files is a complex task: comprehensive searches for deleted files can feasibly only be conducted in an autmated way, yet the performance of such software remains a critical concern. Strict deadlines often exist in legal proceedings, setting a limit on the file recovery phase [AvdB11]. The success rate in finding and restoring deleted files is influenced by multiple factors, the most significant being the manner in which files are stored in a file system, and in-depth, applicable knowledge about forensically relevant file formats. This dissertation is dedicated to improving these foundations of file recovery.

The chapter begins with an introduction to digital forensics, highlighting its brief yet rapidly evolving history. Then, it lays the groundwork with an overview of file storage, deletion, and recovery. Subsequently, the research questions are presented, along with the relevant context in which this PhD project was conducted in. Finally, an outline of the thesis is provided, which guides the reader through the structure and key contributions of the work.

## 1.1 An introduction to digital forensics

Forensic research is dedicated to answering legal questions through the scientific method. This field encompasses trace evidence analysis specifically conducted to support the discovery of truth within criminal investigations, testing hypotheses about events that might have occurred. These traditionally include a real-world location of a (suspected) crime scene, but in a digitally connected world, it extends to virtual spaces across the internet, mobile devices, and cloud storage.

Digital forensics shares its foundations with other forms of forensic investigation. However, it is distinguished by the specific techniques and technologies required for acquiring and analyzing data from digital sources.

> **Digital Forensics:** The application of science to the identification, collection, examination, and analysis of data (distinct pieces of digital information) while preserving the integrity of the information and maintaining a strict chain of custody for the data [tech-KCG06].

Digital traces can be found on a variety of media, ranging from traditional hard drives and removable media like USB sticks and SD cards, to data stored on smartphones, wearables, automotive systems, distributed across networks and the cloud. Each of these sources presents unique challenges to the forensic community due to their diversity and rapidly evolving technologies.

The data itself is just as varied. Direct user data such as emails, documents, images, and videos can be crucial for an investigation, but it is often the supporting data—log files, metadata, timestamps, and encryption keys—that enable the reconstruction of activities and events. Identifying the origin, authenticity, and integrity of this data is a key task for digital forensic investigators.

### Golden Age of digital forensics

The late 1990s and early 2000s can be regarded as the 'Golden Age' of digital forensics. The internet was still emerging, and the digital footprint of everyday life was still relatively small. Broadband internet was the exception rather than the norm, social media had yet to make their breakthrough, and mobile phones were primarily used for calling and texting.

In most households, there was a single desktop PC, often equipped with Windows '95 or '98. These computers ran a limited set of software, making it easier for forensic experts to locate and extract relevant data. Outlook Express and Internet Explorer were the standard software for many PC users at the time, and this homogeneity made predicting where and how information was stored much less complex.

Storage space was a valuable commodity at that time. Hard drives had a capacity that we would now consider minimal, often no more than a few gigabytes. This meant that users had to be selective about what they kept, which limited the amount of potential forensic data. The limited storage and lack of cloud services resulted in a situation where, during a house search, the places to look for digital traces were clear: the local drives of the computer.

### An Era of Crisis

The period following the 'Golden Age', often depicted as an 'Era of Crisis' [Gar10], is characterized by the disruptive impact of a rapidly digitizing society on digital forensic investigation practices. With the increasing prevalence of digital technologies, such as smartphones and tablets, which reach a wide range of target groups, digital traces are being generated everywhere by a diverse audience.

These devices have not only become more numerous, but also more powerful and complex. Each device is accompanied with an evolving ecosystem of apps, each with their own and changing file formats, making the collection and analysis of potential evidence more complicated. The challenges are further compounded by increasing storage capacities and the use of cloud services, forcing investigators to look beyond just local hard drives.

Additionally, new privacy laws, like the GDPR in the EU, bring extra legal and ethical challenges for the collection and processing of digital evidence. This period is also marked by the emergence of new technologies, such as pervasive encryption, which significantly complicate the forensic process. In different ways, these all contribute to the fact that data is less readily accessible to investigators.

This combination of factors underscores the importance for forensic investigators to continuously update their methods and techniques, in line with the ever-changing digital landscape. Ongoing research in various fields is essential, and although it is uncertain where the next major breakthrough will occur, it is clear that the field of digital forensics cannot afford to fall behind.

## 1.2 Storage and recovery of computer files

In order to understand how computer files can be recovered after their deletion, it is important to understand how they are stored and what happens when a user deletes a file, and why file fragmentation hinders the recovery of files.

### 1.2.1 How files are stored on a storage device

Files on a storage device are stored in a structured way, usually in a format determined by the file system. This system keeps track of where data resides on the storage medium. The file system uses a central index with metadata for each file, including the name, size, date, and location of the data clusters on the disk. The file system uses this index to read and write the file data. In computers running the Windows operating system, NTFS (New Technology File System) is the default file system, and this central index is known as the Master File Table (MFT), see Figure 1.1.

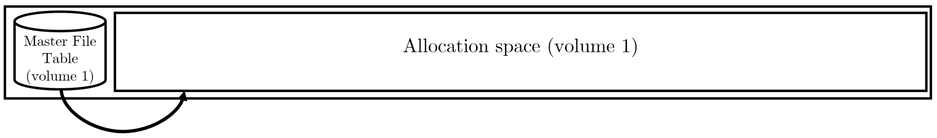

Figure 1.1: NTFS volume on a storage device

### 1.2.2 File fragmentation

Fragmentation is a common occurrence in file systems where files are split into non-contiguous fragments and scattered across different areas of a storage device. This typically happens when files are modified, enlarged, or deleted, leading to irregular free space distribution on the device. Consequently, when the file system attempts to write a new file and cannot find a sufficiently large contiguous space, it will break the file into fragments, see Figure 1.2.

> **File fragmentation** refers to the condition of a file system where a single file is divided into multiple separate pieces, or fragments, that are stored in non-adjacent locations.

These fragments are written in blocks, with 4096 bytes being the default block size, meaning that each fragment of a file will be 4 kB or a multiple thereof. This fragmentation can impact the performance of the storage device, as the read/write head (in the case of traditional hard disk drives) has to move to different parts of the disk to access a single file.

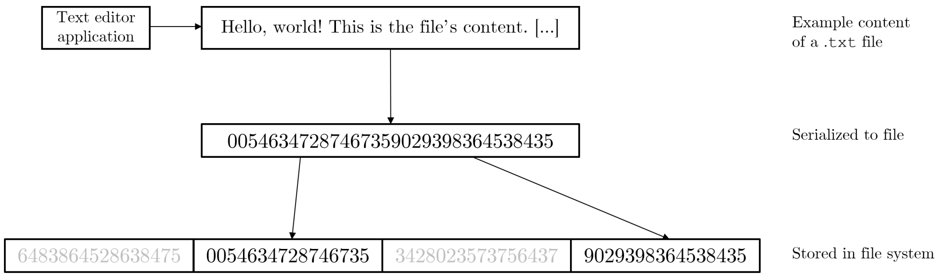

Figure 1.2: File storage and fragmentation example

### 1.2.3 File deletion and recovery

When a file is deleted, the file system typically only removes its index and marks the space it occupied as available for new data. The actual data often remains intact until overwritten, presenting a window for potential recovery. This situation arises not because file systems are designed with file recovery in mind, but due to efficiency: completely erasing data takes more time and resources.

The type of storage device influences how long deleted data remain recoverable. On hard disk drives (HDDs), deleted data persists until physically overwritten. However, solid-state drives (SSDs) may autonomously manage and erase unused data blocks to optimize performance and extend lifespan, potentially reducing the window for data recovery [HMK21].

**File carving:** File carving is the process of reassembling computer files from fragments in the absence of file system metadata [web-Wik23].

**File carver:** Software designed to reconstruct files from raw data on storage media without file system metadata, utilizing file carving techniques to identify and piece together file content.

To recover a deleted file, file carvers often assume that files are not fragmented and search for distinctive starting and ending markers to piece the file back together. This process is relatively straightforward for non-fragmented files, as it only requires correctly identifying the start and end of the file.

However, for fragmented files, which are spread out across the disk in pieces, the challenge significantly increases. The absence of information regarding the file's fragmentation status renders accurate reassembly difficult, often leading to extensive and impractical search times or failure to locate the fragments entirely.

Note that while file carving can retrieve the contents of a file, it does not recover its metadata. This metadata, such as timestamps and allocation data, is stored within the Master File Table in NTFS systems and is often lost upon file deletion.

### 1.2.4 File format specific recovery

The recovery of deleted or corrupted files is significantly influenced by their specific file format, each having a unique structure and layout. These unique characteristics can either aid or complicate the recovery process. Features like internal consistency checks, length fields, and internal pointers in some file formats can provide strong clues about which data fragments belong to a particular file and their correct sequence.

From a forensic perspective, certain file formats are particularly noteworthy. For instance, photographs have always been crucial pieces of evidence. Although there are multiple formats for storing visual data (such as BMP, PNG, GIF), JPEG has emerged as the de facto standard for photographs. However, it is this specific format that poses distinct challenges in recovery. Unlike other formats, JPEG lacks many features like internal consistency checks that could facilitate its recovery. This absence of recovery supporting features means that recovering JPEG files, especially when they are fragmented, requires more sophisticated techniques and a deeper understanding of the structure of the format.

## 1.3 Research questions

This thesis aims to improve the foundations of file recovery, addressing the evolving challenges in digital forensics as described in sections 1.1 and 1.2. Therefore, the main research question of this thesis is:

*How to identify and leverage unexplored potential in file recovery?*

We address this research question by focusing on specific aspects of file recovery from a digital forensic perspective. We first consider the current state of file fragmentation. As explained, file fragmentation introduces major challenges for file carving. To assess the necessity of considering file fragmentation in file carving, we conduct an empirical study to reveal how files are currently fragmented on real-world Windows-based systems. In this empirical study, we collect and examine a large collection of systems in the wild. This should be done in a privacy-friendly manner, since we should not reveal any personal information of the users or owners of these systems. Next, we consider the reconstruction of potential file histories. We study to what extent file history can be recovered from file timestamps. Finally, we study how file carving of JPEG files can be improved, since JPEG files are particularly noteworthy from a forensic perspective and the JPEG file format introduces major challenges for file carving.

To investigate this, we examine the structure and recoverability of files across different perspectives:

- **File system** – Understanding the entire file system's impact on file recovery processes.

- **File metadata** – Analyzing how an individual file's metadata can aid in recovery efforts.
- **File format** – Research file format specific properties to maximize file recovery opportunities.

Based on this, we have formulated the following sub-research questions to address the main research question.

**RQ1:** How can data on file fragmentation be collected in a privacy-friendly manner?

**RQ2:** How are files on real-world, in-use Windows systems fragmented?

**RQ3:** To what extent can file history be recovered from file timestamps?

**RQ4:** To what extent can entropy-coded JPEG data be validated?

**RQ5:** How effective are the newly identified approaches to JPEG fragmentation point detection in real-world JPEGs?

Within the context of the whole file system, our focus is directed toward research questions RQ1 and RQ2. There was no data or study available on file fragmentation data, the most recent study was published in 2007 [Gar07], and likely contained data from predominantly PC's running Windows 95, Windows 98 and Windows 2000 (due to the time frame of the data collection). Therefore, there was a need for a modern, large, real-world data-collection on file fragmentation. RQ1 addresses this, considering the fundamental challenge of preserving the privacy of device owners. Once an appropriate data collection tool is developed and a dataset is obtained, RQ2 delves into the degree of file fragmentation across various storage devices and file types. It further distinguishes between different fragmentation patterns, providing insights into which patterns are (most) prevalent, and must therefore be addressed by file recovery methods.

For the forensic analysis of individual files, we turn our attention to research question RQ3, which examines the extent to which a file's history can be reconstructed from its system metadata. Shifting to the file metadata perspective, RQ3 investigates the potential of file timestamps in reconstructing a file's history. This query highlights the critical role of metadata in recovery efforts and explores how timestamps can unveil the potential lifecycles of a file.

Transitioning to the file format specific analysis, RQ4 and RQ5 delve into the JPEG format. RQ4 focuses on bit-level correctness—a notably neglected aspect in current studies. With these theoretical findings, RQ5 investigates the extent to which these identified approaches are effective when realized in a functioning artifact and tested on a large-scale JPEG dataset.

Each subsequent chapter addresses one of the subquestions, aiding in answering the central research question. Chapter 2 addresses RQ1, Chapter 3 addresses RQ2, Chapter 4 addresses RQ3, Chapter 5 addresses RQ4, and Chapter 6 addresses RQ5.

## 1.4 Thesis overview and contributions

As an article-based dissertation, each chapter is based on one peer-reviewed publication. The following publications form the core of this work:

- The paper titled *File Fragmentation in the Wild: a Privacy-Friendly Approach*, at IEEE International Workshop on Information Forensics and Security (WIFS), in 2019 ([WIFS19]). The main contributions are:
  - Establishing a set of requirements to safeguard privacy during the collection of file information.
  - Designing a filtering wrapper for Fiwalk, a commonly used forensics tool, based on these requirements.
  - Measuring file fragmentation in the wild through a case study of 220 in-use, privately-acquired laptops.
- The paper titled *A contemporary investigation on NTFS file fragmentation*, at the Digital Forensics Research Conference (DFRWS-APAC), in 2021 ([DFRWS-APAC21]). The main contributions are:
  - The file fragmentation corpus contains significantly more files than previous studies, with (among others) over 1 million .jpg and 87,000 .docx files, marking a 2 to 10-fold increase.
  - Novel metrics are introduced to assess the complexity of fragmentation, including the degree of internal fragmentation and the degree of out-of-orderedness.
  - Analysis of various fragmentation characteristics, including the relationship with file size, used volume space, fragmentation by file extension, gap size for files fragmented into two parts, distribution of the number of fragments, correlation between fragmentation and disk size, and the impact of disk type (primary/secondary) on fragmentation.

  - Among the findings is the discovery that the average degree of out-of-order fragmentation of files is substantial, carrying significant implications for the field of digital forensics.

- The paper titled *Reconstructing Timelines: from NTFS Timestamps to File Histories*, at the International Workshop on Digital Forensics (WSDF), held in conjunction with ARES, in 2023 ([WSDF23]). This paper was awarded the Best Research Paper Award of the workshop WSDF @ ARES 2023. The main contributions are:
  - A method for determining all possible histories of a file based on a set of operations, encompassing (1) identifying the impact of each operation on timestamps, (2) reversing these impacts to ascertain the applicable constraints for inversion, and (3) reasoning back from the current file timestamps by aligning them with these reversed effects.
  - Development of a proof-of-concept toolchain tailored for the NTFS file system, enabling a visualization of possible histories (sequences of operations) deduced from the file's existing timestamps.

- The paper titled *JPEG File Fragmentation Point Detection Using Huffman Code and Quantization Array Validation*, at the International Workshop on Digital Forensics (WSDF), held in conjunction with ARES, in 2021 ([WSDF21]). The main contributions are:
  - Introduction of a novel method for validating JPEG files through detection of invalid Huffman codes and quantization array overflows.
  - A validation algorithm that incorporates these techniques, underpinned by an analysis of file fragmentation and JPEG structure.
  - A practical example illustrating the algorithm's application, followed by an evaluation of its effectiveness, limitations, and considerations in the context of existing research.

- The paper titled *Problem solved: a reliable, deterministic method for JPEG fragmentation point detection*, at the Digital Forensics Research Conference (DFRWS-EU), in 2024 ([DFRWS-EU24]). This paper was awarded the DFRWS EU 2024 Best Paper Award. The main contributions are:
  - Expansion, implementation, and efficacy measurement of our previously introduced fragmentation point detection algorithm, leading to the creation of a JPEG validator.

  - Compilation of a sizeable, relevant dataset covering all JPEG variants occurring in the real world, to test the validation methods.
  - Evaluation of the performance of the validator in detecting fragmentation points, demonstrating an exceptional success rate of 99.997% for the most common JPEG encoding.

### 1.4.1 Collaborating institutions and roles

This PhD research was conducted as an external candidate at the Open Universiteit. The candidate worked as a lecturer at Zuyd University of Applied Sciences and as a researcher at the Data Intelligence research group. Furthermore, he served as an external employee at the Netherlands Forensic Institute during the research period. The candidate was awarded a Teacher Doctoral Grant by the Netherlands Organisation for Scientific Research (NWO), ensuring the continuity of the research.

### 1.4.2 Personal Contributions

This section provides an overview of the PhD candidate's contributions, outlined on a per-publication basis following the CRediT (Contributor Roles Taxonomy) guidelines [web-Els24]. Each CRediT role listed here represents a substantial personal contribution, but does not imply sole responsibility. Multiple authors may have contributed in similar or different roles.

- [WIFS19]: **Conceptualization, Investigation, Writing – Original Draft, Supervision**
  I conceptualized and designed the Artifact [artefact-Dol19]. The data collection, conducted under my supervision, included ethical reviews and participant briefing. As the first author, I co-authored this paper and presented the results at the International Workshop on Information Forensics and Security in 2019.

- [DFRWS-APAC21]: **Data Curation, Writing – Original Draft**
  I conducted a thorough analysis on the dataset, uncovering all potentially relevant results and their representations. As the first author, I co-authored this paper and presented the findings at the Digital Forensics Research Conference APAC in 2021.

- [WSDF21]: **Investigation, Writing – Original Draft**
  For this two-author paper, I collaboratively engaged in both the research and

the writing processes. I presented the results at the International Workshop on Digital Forensics in 2021.

- [WSDF23]: **Writing – Original Draft, Visualization, Supervision**
  I conceptualized and validated a visualization tool [artefact-vdAke21], that complements the previously constructed artifact under my supervision [artefact-Bou21]. I co-authored the paper and presented the results at the International Workshop on Digital Forensics in 2023.

- [DFRWS-EU24]: **Software, Validation, Data Curation, Writing – Original Draft**
  I constructed the JPEG dataset [dataset-vdMee22] and all subsets used for both validating the validator and addressing research questions. I co-developed and tested the JPEG validator [artefact-vdBvdM23]. As the first author, I co-authored the paper and presented the results at the Digital Forensics Research Conference Europe in 2024.

### 1.4.3 Relations between publications and other research components

Figure 1.3 illustrates the interconnections between supervised student projects, artifacts, datasets, and the publications resulting from this research. As can be seen, many research opportunities were explored. Sometimes, the results of projects led to artifacts, datasets, and contributed to publications. This visualization demonstrates the exploratory, result-driven approach that was employed during this PhD project.

### 1.4.4 Contributions to society

In the role of both a part-time teacher and researcher, there were two explicit objectives aligned with the applied research: (1) contributing to student education and (2) contributing to society. The course on Reverse Engineering Data, directly linked to the research, was among the subjects taught. Additionally, courses related to applied statistics and research design also benefited from the accumulated research experience.

Beyond the teaching responsibilities, 12 student projects were supervised, including internships and graduation assignments at both Zuyd University of Applied Sciences and the Open Universiteit, integrating these projects into the broader context of the PhD research.

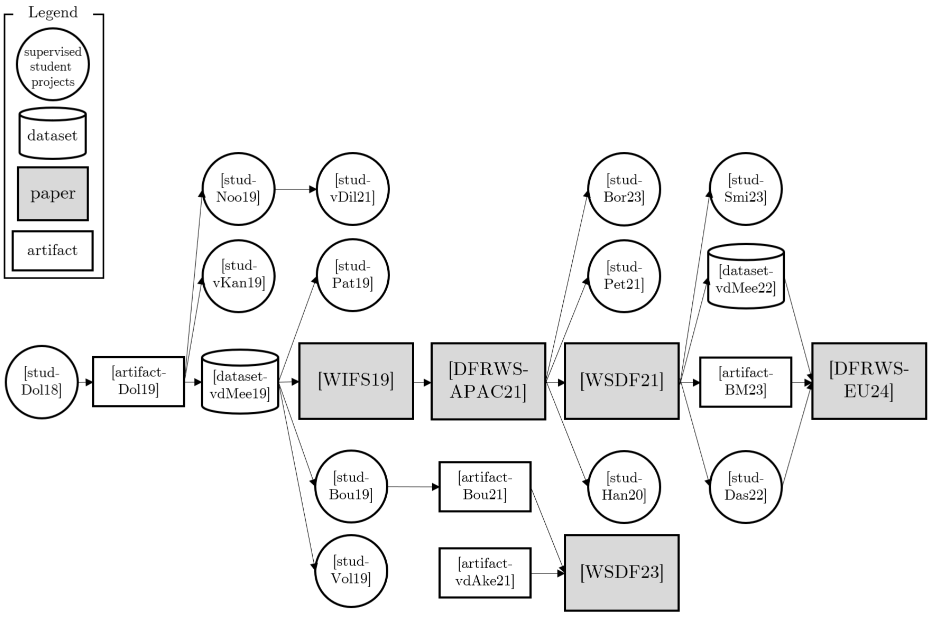

Figure 1.3: Relation between artifacts, datasets, supervised student project and publications

- Zuyd University of Applied Sciences
  - 4 internship assignments
  - 2 graduation projects (BSc)
- Open Universiteit
  - 4 Bachelor's projects (BSc)
  - 2 Master's projects (MSc)

In 2018 at Zuyd University of Applied Sciences, Dols [stud-Dol18] conducted a graduation project exploring the possibilities of measuring file fragmentation in a detailed and privacy-friendly manner on Windows computers, leading to the first version of the artifact [artefact-Dol19]. Inspired by the work of Karresand et al.[KAD19], Patti [stud-Pat19], undertook an internship project that investigated volume visualization using fractals. Van Kan [stud-vKan19] carried out an internship that investigated the creation of a privacy-friendly, low-level access method for examining storage devices on Android phones.

At the Open Universiteit, three students simultaneously started their bachelor's projects in 2019. Vollebregt [stud-Vol19] investigated file dating based on physical location on a disk, inspired by the work of Bahjat and Jones [BJ19]. Noordzij [stud-Noo19] designed and implemented a method to synthetically age a file system in order to create realistically fragmented file systems. Following this, Van Dillen [stud-vDil21] evaluated existing file system ageing tools, building on the work of Noordzij and Hueting (not supervised by the author). Bouma [stud-Bou19] researched file timestamps, examining how file operations impact timestamps and which sequences of file operations may have led to the current set of timestamps.

The supervision at the Open Universiteit also extended to two master's projects: Peters [stud-Pet21] focused on file format structures for file recovery, particularly the PST (Outlook) format, and Borgers [stud-Bor23] sought to identify differences between NTFS drivers and their interactions with storage devices.

Lastly, at Zuyd University of Applied Sciences, the graduation project of Hanegraaf [stud-Han20], who explored file recovery on solid-state drives, and the internships of Dassen [stud-Das22], who automated the extraction and storage of JPEG metadata, and Smitz [stud-Smi23], who benchmarked existing file carvers for their effectiveness in recovering fragmented JPEG files, were supervised.

### Public outreach

Engaging in public outreach offered additional avenues to disseminate research findings to broader audiences. These opportunities, ranging from industry conferences to public science fairs, facilitated the contribution of insights from the work beyond the academic sphere. The following events stand out:

- Futurum, 2017 [web-Fut17], Heerlen.
- Nederlands ICT in het Onderwijs Congres [web-NIO18], 2018, Leeuwarden.
- Learning and Innovation in Resilient Systems [web-OU23a], 2019, Heerlen.
- APG devConf [web-Dev22] (Keynote), 2022, Online.
- IllionX DevDays, 2022, Arnhem.
- NFI Science Fair, 2023, The Hague.
- PROMIS'23 [web-OU23b] (Keynote), 2023, Utrecht.

### 1.4.5 Thesis outline

This dissertation is divided into two main parts. The first part, titled ”File fragmentation and timestamps”, focuses on the collection of a comprehensive dataset containing metadata about file fragmentation and timestamps. It also discusses the findings, artifacts, and publications that emerged from this research. The second part, titled ”JPEG validation”, delves into the JPEG file format. It introduces and evaluates a novel approach for validating JPEG images, aiming to enhance the success rate of recovering fragmented JPEG files.

**Part I – Chapter 2: Collecting real-world data on File Fragmentation.** Considering the current scarcity of real-world data on file fragmentation, this chapter establishes the foundation for gathering a contemporary dataset on the subject. It details the design, implementation, and publication of the artifact [artefact-Dol19], as well as an evaluation of its design objectives following a data collection exercise.

**Part I – Chapter 3: A Contemporary Investigation of NTFS File Fragmentation.** This chapter introduces novel metrics to express the degree of file fragmentation, facilitating a clear comparison with previous studies and enabling detailed reporting on the current dataset. Additionally, it provides insights into various characteristics of fragmentation, including analysis by file extension and disk type.

**Part I – Chapter 4: Reconstructing Timelines: From NTFS Timestamps to File Histories.** This chapter studies to what extent file history can be derived from timestamp information. To this end, it provides a methodology for deriving and describing a file’s previous timestamp values, and introduces two artifacts. One artifact [artefact-Bou21] that implements this methodology, a second artifact [artefact-vdAke21] that visualises the resulting tree in a timeline-format. This demonstrates the practical applicability of the approach.

**Part II – Chapter 5: JPEG File Fragmentation Point Detection using Huffman Code and Quantization Array Validation.** Building upon the insights from file fragmentation findings, particularly the often overlooked out-of-order fragmentation not addressed in current file-carvers, this chapter presents a new theoretical approach to JPEG file validation, with algorithms applicable to a wide range of JPEGs (both baseline and progressive). This method is designed to significantly enhance future file carving efforts in restoring deleted fragmented JPEG files.

**Part II – Chapter 6: Problem solved: a reliable, deterministic method for JPEG fragmentation point detection.** This chapter presents a JPEG validator [artefact-vdBvdM23] that implements the JPEG validation mechanisms introduced in the previous chapter. The validation mechanisms are extensively tested on a large-scale JPEG dataset, covering both regular and worst-case scenarios, yielding exceptionally positive results.

# Part I

# File fragmentation and timestamps

Highlights:

- Design and implementation of a privacy-friendly tool for file fragmentation analysis ([artefact-Dol19]), followed by the collection of a large-scale dataset on file fragmentation.
- Detailed reporting on file fragmentation, including the introduction of new metrics to address the phenomenon of 'out-of-order' fragmentation, an important aspect often overlooked in existing file recovery methods.
- An novel approach to file timestamp analysis through timeline creation, featuring artifacts that demonstrate the feasibility ([artefact-Bou21]) and visualization ([artefact-vdAke21]) of this method.

Chapter 2

# Collecting real-world data on file fragmentation

*This chapter presents an adapted version of the paper* File Fragmentation in the Wild: a Privacy-Friendly Approach*, by Vincent van der Meer, Hugo Jonker, Guy Dols, Harm van Beek, Jeroen van den Bos, and Marko van Eekelen, which was published in the 11th IEEE International Workshop on Information Forensics and Security [WIFS19].*

**Abstract** Digital forensic tooling should be based on reference data. Such reference data can be gathered by measuring a baseline, e.g. from volunteers. However, the privacy provisions in digital forensics tools are typically tailored for criminal investigations. This is not sufficient to ensure privacy obligations towards volunteer participants. Thus, privacy adaptations are necessary before such tooling can be used to establish or rejuvenate a baseline.

We illustrate the feasibility of this approach by rejuvenating a baseline for file carving, via a case study of file fragmentation. We derive a set of privacy requirements to prevent deanonymisation of individuals. Atypical properties of files can nevertheless still lead to plausible deanonymisation of the file. With regards to fragmentation, we find *out-of-order fragmentation*, where a later block is stored on disk before an earlier block of the same file, occurs in nearly half of all fragmented files. This is the first study to report on prevalence of this type of fragmentation. Its high rate of occurrence has implications for the practice of file carving.

## 2.1 Introduction

Forensics is the science of finding and evaluating evidence in the context of a criminal investigation. Digital forensics is the application of this science to the digital domain. Both need to be grounded in reality. Moreover, technological developments are swift, implying that any baseline for digital forensics must regularly be recalibrated.

Consider the case of file carving, a technique to reconstruct files from their contents as blocks stored on disk, instead of from metadata. This technique is used in digital forensics to e.g. recover deleted files. However, the amount of blocks on a modern drive is enormous, which leads to a gargantuan search space for file carvers. To reduce the search space, file carvers make several assumptions, one of which is how blocks belonging to the same file are distributed over the disk: *file fragmentation.* The current de facto baseline for file fragmentation is the seminal study by Garfinkel [Gar07] from 2007. Technology has marched on since then: that study concerned mostly FAT12, FAT16 and FAT32 file systems, which have since been replaced by NTFS; disks have become (much) larger, operating systems have been updated, and laptops continue to replace desktops – and act far more as *personal* computers than traditional desktops. Finally, modern laptops often use a fast SSD drive for read-write intensive operation, and a slower HDD drive for data storage. Such a task-based division of drives did not exist at the time of Garfinkel's study.

Given these considerations, it is important to restudy file fragmentation, specifically to examine current, real-world occurrences of fragmentation, to improve the reliability and development of existing and future file carvers. Thus, a new study of file fragmentation would ideally be based on a large group of real-world, in-use systems.

However, there is a privacy problem. Studying file fragmentation requires determining the location of all blocks, which requires access to file metadata. But this metadata also contains information not pertinent to studying file fragmentation, some of which may be privacy-sensitive (e.g., filenames containing personal details). Such metadata must absolutely not be collected in measuring fragmentation. This entails that standard forensics tools cannot be used as-is, since these tools do not offer the required privacy provisions. Fragmentation tools outside the forensic domain typically focus on solving fragmentation, not on analysing it. Thus, there are currently no tools which can analyse fragmentation and simultaneously ensure the privacy of volunteer participants.

This chapter focuses on RQ1, which is stated as: *How can data on file* *fragmentation be collected in a privacy-friendly manner?*

*Contributions.* The main contributions of this chapter are:

- We establish a set of requirements to safeguard privacy when collecting file information.
- We design a filtering wrapper for Fiwalk, a commonly used forensics tool, based on these requirements.
- We measure file fragmentation in the wild by a case study of 220 in-use, privately-acquired laptops and find:
  - the privacy requirements successfully prevent identification of individual users, but atypical properties of a file could lead to its plausible identification.
  - over 46% of fragmented files are fragmented out-of-order, a type of fragmentation whose occurrence was not previously reported on.

## 2.2 Background

### 2.2.1 Terminology

We make use of the following NTFS terminology:

- **Master File Table:** the Master File Table (MFT) contains the metadata of the files in a NTFS volume. For each file, the MFT holds the metadata (like timestamps, filename, size, permissions) and the allocation of blocks for the files data. The location of the MFT itself is stored in the boot sector [Car05]. To prevent fragmentation of the MFT itself, space is reserved for future growth.
- **Resident files:** resident files are files that do not have allocated blocks. Their data is fully stored in their record in the MFT. Typically, a resident file has a maximum size of about 700 to 800 bytes [Car05]. Note that resident files by definition cannot be fragmented, since no blocks are allocated to them.
- **Compressed files:** Files may be compressed by NTFS itself, as opposed to application-level compression. This compression is transparent to any application using NTFS.

- **Sparse files:** Files where only blocks containing non-zero data are stored. The file size of sparse files is thus typically larger than allocated on disk. (Used e.g. for virtual machine files.)
- **Hard links:** An MFT entry may contain more than one path + filename. These names appear to the user as individual files,
- **Symbolic links:** a symbolic link (also called soft link) is an entry in the MFT that points to a path + filename. To resolve a symbolic link, the path and filename must exist. While symbolic links resemble files, they do not contain any data and thus cannot be fragmented.
- **Volume:** A storage device is a physical unit for storing data. It is partitioned into one or more volumes, which in Windows are addressable via drive letters.

In addition, we use HDD to denote *hard disk drive*, i.e., a storage medium based on magnetic storage with moving read and write heads and spinning discs; and SSD to denote *solid-state drive*, i.e., a storage medium based storing data based on integrated circuits (typically flash memory), without physically moving parts.

### 2.2.2 Data storage and deletion on SSDs

SSD devices operate differently than HDDs. For example, to extend the longevity of the disk, they typically use wear leveling: a technique to avoid writing overly much in one area of the disk. However, wear leveling happens in the firmware and is invisible to the NTFS file system. That is: it does not affect the operation of NTFS, and the NTFS file system is not aware of this taking place.

SSD devices also handle deletion differently than HDDs. Regular HDDs handle deletion by marking the deleted blocks of the disc as available. That is, regular HDDs leave the data on the disc until it is overwritten. In contrast, an SSD drive cannot write to an already occupied part. Thus, each block must be empty before it can be written to. The earliest SSDs used a form of garbage collection to empty deleted blocks. This matured into the creation of the TRIM command, which wipes the specified blocks. Once blocks have been wiped, their data is physically removed from the disc and thus the data no longer recoverable. This raises the question of whether recovery of deleted files is possible at all on SSDs.

Nisbet et al. [NLR13] show that once the TRIM command has been sent to the drive, erasing usually takes places within minutes. They also show that, within the time frame of deleting a file by the user and the execution of the TRIM command, significant amounts of data can still be recovered, with small files being fully recovered, and for large files being partially recovered. After the execution of the TRIM command however, only up to 0.6% of the data was recoverable. This places concrete boundaries on the forensic effectiveness for file carving.

In case the data on the SSD was subjected to a successfully executed TRIM command, the data thus is not realistically recoverable. However, there is no one-to-one correspondence between file deletion and successful execution of a TRIM command. In particular, there are various reasons why a SSD either is not TRIM-enabled, or that a TRIM command is not succesfully executed [web-Foc20].

### 2.2.3 Fragmentation

The NTFS file system stores files into blocks, where each block occupies a fixed size on disk. Blocks are identified by their block number. A file is thus assigned a list of block numbers. A file is *not* fragmented if the assigned block numbers are listed in order, and these block numbers are consecutive. When this is not the case, the file is fragmented. This may be because the blocks occur out of order, because the block numbers are not consecutive, or both. This gives rise to four storage patterns, as depicted in Figure 2.1. Of these storage patterns, in-order contiguously stored files are not fragmented. The other patterns describe fragmented files.

| | *Contiguous* | *Non-contiguous* |
|---|---|---|
| *In-order* | $\boxed{A}\boxed{B}$ | $\boxed{A}\ldots\boxed{B}$ |
| *Out-of-Order* | $\boxed{B}\boxed{A}$ | $\boxed{B}\ldots\boxed{A}$ |

Figure 2.1: Examples of the four storage patterns for a bi-fragmented file

Two types of fragmentation can occur on a file system:

1. *fragmentation of free space* is caused due to the deletion and shrinking of files. While these operations typically do not fragment the file itself, they do create unallocated space that is likely not adjacent to the (other) already existing unallocated space.

2. *file fragmentation* occurs when the file system does not write a file contiguously. File fragmentation can happen when new files are created or existing files are extended. Note that file system implementations may choose to do so even when it is not strictly necessary (i.e., when there is sufficient contiguous free space available).

We refer to the various parts of a fragmented file as fragments. More specifically, a file consists of a number of blocks, which are grouped into one or more *fragments*. A fragment is contiguous and in-order, and cannot be extended with more blocks of the same file while remaining in-order and contiguous.

Note that since Windows Vista, Windows by default schedules a periodic defragmentation task.

Lastly, remark that while SSD firmware performs wear leveling (which distributes file blocks evenly over the physical storage), this is invisible to the file system.

### 2.2.4 Degree of fragmentation

The degree of fragmentation can be defined in various ways, depending on what is considered the total number of files. In literature, it is not always clear which definition is used. It is the ratio of the number of fragmented files divided by a total. Different choices can be made for the total, which gives rise to four definitions.

**Definition 1** (degree of fragmentation)**.** The degree of fragmentation is the number of fragmented files divided by the total number of files. The total number of files defined as:

I all MFT entries, OR

II all MFT entries with data, OR

III all MFT entries with blocks assigned, OR

IV all MFT entries with $\geq 2$ blocks assigned.

Note that definition I covers all MFT entries, including symbolic links; definition II excludes symbolic links; and definition III furthermore excludes resident files. Nevertheless, definition III still includes files of one block – which inherently cannot fragment. Definition IV is the only definition which excludes all non-fragmentable MFT entries from consideration. It is thus the most strict, while definition I is the most broad definition (gives the smallest degree of fragmentation).

We consider definition IV most relevant for reporting on measurements of file fragmentation. Definition III is useful when the number of blocks of a file is unknown (e.g., in file carving). Most studies unfortunately do not clarify which definition they use.

## 2.3 Related work

### 2.3.1 File fragmentation

There have been three large-scale studies reporting on file fragmentation. We summarise their findings in Table 2.1.

| *source* | | *year* | *% frag* | *used frag. definition* |
|---|---|---|---|---|
| [Gar07] | Garfinkel | 2007 | | |
| | – *all file systems* | | 6 | ? |
| | – *NTFS file systems* | | 12.2 | ? |
| [MB12] | Meyer & Bolosky | 2012 | 4 | ? |
| [WIFS19] | Van der Meer et al. | 2019 | 2.2 | all MFT entries (I) |
| | | | 4.4 | fragmentable files (IV) |

Table 2.1: Comparison of fragmentation rates found in literature

The seminal large-scale study into file fragmentation is due to Garfinkel [Gar07]. He gathered data from over 300 used hard disks. The dataset includes 219 FAT file systems, 51 NTFS file systems and 5 UFS file systems. He found an average percentage of file fragmentation of 6%. Most findings are reported over the entire dataset. The paper does provide sufficient information to derive the fragmentation rate over all 51 NTFS file systems, namely 12.2%.

Garfinkel reports several findings. He found different file types have different fragmentation rates, that most fragmented files are split into two parts (bifragmented), and he reports on the gap size between the two fragments of bifragmented files. It is not clear which definition of degree of fragmentation Garfinkel uses in his paper.

In a study on file system content of 597 Windows computers, Meyer and Bolosky [MB12] reported finding a level of file fragmentation of 4%. In addition, the most highly fragmented files within their dataset were log files. Note that it is not clear which definition Meyer and Bolosky used to calculate the percentage of fragmented files.

While the previous works focused on desktops and laptops, several studies have investigated file fragmentation on smartphones. Ji et al. [JCS+16] report on EXT4 fragmentation behaviour on four Android smartphones. They observe that files, especially database files, may suffer from severe fragmentation. In a study [CBJ+17] on the effects of file fragmentation, the authors observed that fragmentation quickly emerged in the aging process and impacts user-perceived latencies. In a study using five smartphones, Ji et al. [JCH$^+$18] find that, under daily use, fragmentation quickly begins to occur. They find that for such devices, fragmentation is strongly correlated with disk space utilization. Moreover, the specific way how SQLite files are used (frequent deletions, synchronous writes) exacerbates fragmentation as well.

Finally, Darnowski and Chojnackhi [DC18] derive a model of NTFS block allocation algorithms that predicts how a new file will be stored. They propose modelling the NTFS allocation strategy as a finite state machine. They define a sequential model for writing files, which provides predictions on block allocation. These predictions include predicting when fragmentation occurs and even cover out-of-order fragmentation. They confirm the accuracy of their model via synthetic experiments.

### 2.3.2 File carvers

Another area of related work regards file carvers. Garfinkel [Gar07] constructed a file carver targeted specifically at bi-fragmented files. Following up on this, Roussev and Garfinkel [RG09] argued for a specialized carving approach: carvers tailored to specific file content types. Such carvers would be able to improve upon generic file carving techniques.

A different approach was taken by Cohen [Coh07]. He supplied a mathematical analysis of file carving. His analysis accounts for a model of fragmentation, to reduce the number of mappings from blocks to files. Notably, the approach he proposed allows to model different kind of fragmentation patterns, including contiguous out-of-order fragmentation.

Nevertheless, file fragmentation poses a problem for file carving. This was discussed in several file carving surveys, such as Pal and Memon's survey of the evolution of file carving [PM09]. They found that the first generation of carvers, based on file structure (using identification of file-header and -footer) was unable to handle fragmentation. Such carvers would typically 'recover' files that had unrelated content in their midst. This led to the development of new approaches to file carving, which aimed to overcome the difficulties of carving caused by file fragmentation. Similar findings resulted from the multimedia file carving survey of Pahade et al. [PSS15]. They concluded that multimedia files are often fragmented and compressed, and find that then-current carving tools are often unable to handle different types of fragmentation patterns.

### 2.3.3 Privacy-preserving forensics

The other area of related work revolves around *privacy-preserving digital forensics*. The notion of preserving an individuals privacy while trying to uncover relevant information about a suspected crime seems a contradiction. Nevertheless, several approaches have been suggested to enable some form of privacy during a forensics investigation. Most approaches focus on revealing only part of the (seized) data. One early use case where this emerges is in unavoidable attribution, which is sometimes proposed for privacy-enhancing technologies. For example, Olivier discussed [Oli05] how to enhance a method for obscuring web requests to assure attribution in case of investigation.

A different approach, more applicable in digital forensics, is to limit the data revealed to the investigator only to that which is necessary. Efforts in this vein seek to balance the privacy of the investigated subject (who may be innocent) with the needs of a forensic investigation. For example, Croft and Olivier [CO10] proposed an incremental model of cryptographic keys where knowing plaintext (facts) pertaining to less privacy-sensitive issues will reveal keys for higher privacy-sensitive information. The approach by Lawe et al. [LCY+11] similarly sought to limit the information disclosed to an investigator. In contrast, they chose to use cryptography to enforce regulation. They proposed to use a keyword encryption scheme to encrypt the data. Requests for decryption keys are then evaluated by a designated intermediate, who is responsible for privacy. Finally, Verma et al. [VGG18] proposed a privacy-preserving framework for analysing collected data.

While these approaches are interesting, they focus on limiting access to data in the analysis phase of a forensics investigation. In contrast, this study is focused on acquiring data to improve forensic tooling. Since we do not gather data in the context of a (criminal) investigation, privacy must be ensured *prior to* data analysis. A concept more in line with this is the concept of verifiable limited disclosure, as proposed by Tun et al. [TPB+16]. The authors recognised that more crimes are happening on social media, and proposed a way for witnesses to share their view of the social media with investigators, without providing full access. They proposed an approach where cloud providers encrypt and certify partial time lines for forensics investigation. While our setting does not use cloud providers, the idea of collecting only limited data is one that we will apply.

## 2.4 A privacy-friendly approach to file system data collection

Ideally, all digital forensics is grounded in real-world data. However, digital forensic tools are privacy-invasive, which makes using these on volunteer participants (whose privacy must be ensured) problematic. Therefore, the development of privacy-preserving digital measurement techniques is foundational to advancing the field.

In order to gain an accurate view on modern fragmentation patterns, file fragmentation data from current, in-use devices is collected. To provide accurate data for this purpose, data should be collected from the general population. Acquiring fragmentation patterns requires a low-level view on a disk, specifically, knowing which blocks correspond to what file. Such a low-level view requires broad access to the disk. Since the data is collected from volunteer participants, privacy is a strong requirement. Measures must be taken to preserve participant privacy whilst collecting data. This will ensure privacy during analysis and further processing.

### 2.4.1 Ethical aspects of data collection

The study design was reviewed and approved by our institutional ethics review board. Participation was on a voluntary basis. All participants were informed about the data that would be collected, and the risks associated with the data collection to their equipment. Before participation, volunteers were informed that since data collection was anonymous, their data could not be removed from the collection.

### 2.4.2 Privacy requirements and operational constraints

To guarantee participant anonymity, we require that no elements of the data collection can be traced back to an individual participant. We consider an attacker that has access to the full dataset, but not to any participant's device. Thus, any file system data that could potentially identify a participant must not be recorded. For each column in the MFT, we considered whether a user could, in the regular course of using a system, cause personal identifiable information to be present. These are the `Data` and `FileName` columns. Users can enter information in these columns via the file contents itself, the file name and extension, and directory names.

In addition, some entries in the MFT are special (meta-files). These may also contain user-settable information. We considered all special entries in the MFT. Of these, only the `$Volume` entry contains user-settable information (the volume name of a file system).

This leads to the following privacy requirements:

- **no-participant-info:** information about participants must not be recorded.
- **no-file-contents:** file contents must be excluded.
- **no-filenames:** filenames must be excluded.
- **no-file-paths:** file paths must be excluded.
- **no-volume-names:** volume names must be excluded.

However, the motivation for collecting data is to result in data that is useful for the file carving community. In particular, it is important to learn whether specific file types are fragmented differently than others. Moreover, barriers to participation should be reduced as much as possible. This leads to the following operational constraints:

- **only-safe-extensions:** only file extensions known to not contain privacy sensitive data are included.
- **performance:** data collection on a system should be finished within a reasonable amount of time.
- **no-change:** the evaluation must not change the contents of the observed file system.

### 2.4.3 Implementation

Acquiring data on file fragmentation can be done using three approaches:

- using user-grade disk analysis tools,
- creating a new tools,
- using forensic-grade disk analysis tools.

The objective of user-grade fragmentation-related tooling is to defragment rather than to report on current status of fragmentation. As such, this type of tooling is rather limited in what it reports about fragmentation – an insufficient level of detail for fragmentation analysis. In addition, these tools cause changes on the studied object (the defragmentation itself), which is neither needed nor desirable. A straightforward approach to creating our own tooling would be to copy the MFT. However, the contents of resident files are stored within each MFT-entry. To avoid copying this content would require substantial effort in parsing the MFT format. On the other hand, forensic tools already provide such functionality. They are ideally suited to provide the measurements we sought. However, privacy is typically not a consideration when designing forensic tools. Thus, such tools cannot be used out-of-the-box.

Fiwalk [Gar09] is a forensic tool for gathering details from a file system. It can be configured to omit file contents and not compute hashes of file contents. Not only does this dramatically improve scanning speed (*performance* constraint), it also happens to satisfy the *no-file-contents* requirement. To ensure the *no-change* constraint, we chose to collect data via a bootable USB stick instead of running on the host OS. We ensured privacy in the following ways:

- **no-participant-info:** No information about participants was recorded.
- **no-file-contents:** Fiwalk provides an option (for speed optimisation) to exclude file contents. This also excludes the contents of resident files. In addition, we used the Fiwalk option to not compute hashes of file contents (by default, hashes are computed).
- **no-filenames, no-file-paths, no-volume-names:** Fiwalk output contains filenames, paths and volume names. During the data collection phase, after Fiwalk completes gathering the metadata from a volume, we immediately process the data to remove these.
- **only-safe-extensions:** Fiwalk output contains the full extension. We post-process the data to only keep known extensions and extensions of three (UTF-8) characters or less. For the list of known extensions, we used a list of known extensions from Wikipedia [web-Wik19].

This resulted in a script that executes Fiwalk and filters its output. This artefact is available from [artefact-Dol19].

### 2.4.4 Data collection

Data was collected from the personal machines of volunteer student participants, between October 2018 and January 2019. The machines were individually bought, managed, and maintained by their respective owners. The student population is divided into classes. By visiting each class once, we ensured no double participation, since students can only be enrolled in one class. Data was collected by a custom-made privacy-friendly data gathering tool [artefact-Dol19] based on Fiwalk, by [Gar09]. The output of this is standardised DFXML [web-For20] structured data. This was converted into an SQLite database for analysis.

On six storage devices, one or more volumes were encrypted and thus not accessible for data collection. The 4 encountered EXT4 volumes were also excluded from consideration.

### Out-of-scope NTFS data

The focus of this study was on collecting data on file fragmentation in a privacy-friendly fashion to benefit file carver development. Therefore, data related to NTFS features such as junctions, Encrypting File System (EFS)-files or alternate data streams, was not collected.

## 2.5 Results

| | *MFT entries* |
|---|---|
| All | 84,390,537 |
| With data | 82,960,039 |
| With blocks | 70,320,268 |
| With $\geq$ 2 blocks | 42,671,054 |
| | |
| Fragmented | 1,871,109 |
| Out-of-Order fragmented | 868,917 |
| % OoO of fragmented | 46.4 % |
| *% fragmented* | |
| of all (definition I) | 2.2 % |
| of those with data (definition II) | 2.3 % |
| of those with blocks (definition III) | 2.7 % |
| of those with $\geq$ 2 blocks (definition IV) | 4.4 % |

Table 2.2: Fragmentation results

In total, we collected data from the laptops of 220 volunteers, which are stored in the 'file fragmentation dataset' [dataset-vdMee19]. 217 of these were running

Windows 10, the other three were running Windows 7. Combined these laptops contained 334 drives. A common configuration was to find a laptop with both HDD and SSD (111 laptops), while three systems contained two SSDs. In 70 of the 106 systems with one drive, this was an SSD. All together, these drives were split into 729 volumes containing a total of ~84 million MFT entries, of which ~70 million with allocated blocks, see Table 2.2.

Of the NTFS volumes, 707 volumes had a block size of 4096 bytes. Other NTFS block sizes were rare: 14 volumes had a block size of 512 bytes; 7 had a block size of 1024 bytes and 1 volume had a block size of 2048 bytes.

### 2.5.1 Overall fragmentation results

Over the years, file fragmentation seems to be declining. In 2007, Garfinkel [Gar07] reported that 6% of files 'with data' was fragmented. Meyer and Bolosky [MB12] reported in 2012 that 4% of 'files' was fragmented. It is not clear if they consider this with respect to all MFT entries, all files 'with data' or some other ratio. For our study, we consider the most relevant fragmentation rate for file carving to be only files that could be fragmented, i.e., files of 2 or more blocks. We supply various fragmentation rates in Table 2.2 for comparison purposes. We distinguish between ratios for all MFT entries, for those with data (excluding symbolic links), for those with blocks (also excluding resident files) and for those with at least 2 blocks. Note that we cannot distinguish between empty and non-empty resident files – a consequence of the privacy-aware approach to data collection. The category 'files with data' therefore includes all resident files.

## 2.6 Analysis of fragmentation results

The presented fragmentation results provide a relevant view of the current state of file fragmentation on modern hardware. This provides valuable input into whether advanced file carving techniques remain useful in data recovery and if so, in what area their development should focus in order to increase performance.

### 2.6.1 Amount of fragmented data increases

When combined with earlier studies, there appears to be a steady decline in fragmentation over the years. In twelve years, fragmentation rate has decreased from 6% (Garfinkel in 2007, on all MFT entries with data) to 2.3%, a significant reduction. Whether this means that a smaller amount of data is fragmented is an entirely different matter.

We found that fragmentation of current HDD drives is very low at approximately 0.7%. However, in 2007 a typical $100 hard drive had a 500 GB capacity [web-McC19], while in Spring 2019, a hard drive in that price range had a capacity of 4 TB. So even though fragmentation is reduced by a factor of eight, the size of a typical hard drive has increased by that same factor.

SSDs were not considered in earlier studies, since they were not common in computer systems at the time. In the 2007 study, the average size of a disk was below 3 GB, so a 6% fragmentation rate would correspond to about 184 MB. The rate of fragmentation measured in this study would correspond to the same amount of data if current SSDs were only 8 GB in size. Given that the smallest SSD in our study is 16 GB, it is safe to assume that the amount of fragmented data has actually increased since 2007.

### 2.6.2 Relevance to file carving

Standard file carving can typically recover non-fragmented files. In our study, 97.7% of MFT entries with data were not fragmented and thus recoverable via a standard file carving approach. Nevertheless, there are cases where it is desirable to recover specific files, e.g., as they may contain key evidence. Standard techniques cannot do this. In particular, out-of-order fragmentation is not considered by current standard techniques. Yet, this type of fragmentation constitutes a significant portion of all fragmented files. For example, of all files that are fragmented into two parts, 25.75% is fragmented out-of-order. This means that a file carver focused on recovery of in-order two-part fragmented files (such as proposed by Garfinkel [Gar07]) will miss about a quarter of these files.

Carving for out-of-order fragmented files faces steep combinatorial complexity, but the amount of files fragmented in this fashion is sufficiently large (46.42% of all fragmented files) that in certain cases, recovery of such files will be worthwhile. In particular, there is a significant number of out-of-order fragmented files with two parts (14.46% of all fragmented files). This percentage is even greater than the percentage of all in-order fragmented files of three or more parts (11.89% of all fragmented files).

Cohen [Coh07] developed an advanced file carving approach to also recover contiguous out-of-order fragmented files (see Figure 2.1). Interestingly, this type of fragmentation is nearly non-existent. In our dataset of 42.3 million files with 2 or more blocks, only 8 files are fragmented contiguous out-of-order. All of those 8 files are fragmented into two parts. We conclude that while techniques for recovering out-of-order fragmented files are relevant, recovery of *contiguous* out-of-order fragmented files is currently not relevant.

## 2.7 Discussion

### 2.7.1 Privacy

The goal of the privacy-filtering was to not collect any personally identifiable (PII) data. To this end, we removed filenames, path names, and unknown file extensions from the collected data. Nevertheless, the collected data does contain a lot of information about an individual drive's master file table (MFT). Thanks to the filtering, none of this information leads to an individual. Thus, the stated privacy goals have been successfully achieved.

However, in reviewing the data, we found that we could make plausible conjectures to the identification of specific *files*. In particular, an online search showed that certain combinations of exact, large file size and extension (`exe`, `mp4`) seemed to be unique. Note that plausible identification of files does not lead to identification of individuals: at best, it serves to distinguish a disk from others, not to identify its owner. Nevertheless, such plausible identification of files was unexpected and we consider this undesirable.

One theoretical mitigation would be to remove all file sizes from the dataset. In this case, the file size can still be approximated from the block size and the (known) number of blocks, but the exact number of bytes is no longer available. In a small experiment, we tested whether a particular (very large) `exe` file would be harder to find when the exact size was not available. We found that the test file was still readily distinguishable amongst the search results. Thus, while removing the file size would theoretically introduce more uncertainty, in practice the effect is negligible.

Therefore, we deem that removing the file size is insufficient to mitigate this problem in practice. Moreover, additionally removing the block size likely does not bring relief: in our dataset, 97% of disks has the same block size. Thus, while participant privacy is successfully safeguarded, possible inferences regarding files must be further investigated before the dataset can be made publicly available.

### 2.7.2 Generalisability of file fragmentation findings

The finding that the average fragmentation rate has dropped since previous studies, is supported by the introduction of default weekly defragmentation in Windows systems since Garfinkel's study.

The second fragmentation finding is the rate of out-of-order (OoO) fragmentation. This type of fragmentation was previously not reported on. To validate this finding, we examined individual OoO rates for file systems with more than 10,000 files in our dataset. Of these, three quarters had an OoO-rate of over 40%. In addition, we measured file fragmentation on two fresh Windows 10 installations on virtual machines. Both installations were updated, one online, one with an offline update package. As expected, the fragmentation rates were low (0.2% tot 0.4%). We found that the percentage of fragmented files that was OoO-fragmented was 44.4% and 45.0%, respectively. These results show that our findings on out-of-order fragmentation are not user-dependent. Moreover, they are remarkably close to our finding of an average rate of out-of-order fragmented files of 46%.

## 2.8 Conclusions

In this chapter, we proposed a privacy-friendly approach to performing measurements using forensic tooling. We establish a set of requirements to prevent deanonymisation of participants. The requirements are derived by considering where personal identifiable information (PII) could be stored in the MFT. We developed a wrapper around Fiwalk to discard any such data. Our data collection attained the desired level of privacy: our data collection does not contain any PII. However, atypical properties of individual files may still allow the file to be deanonymised. While we believe this only applies in certain edge cases, we nevertheless consider this undesirable and will consider measures to address this in the dataset.

The presented data also provides new input for designing file carvers. As such, this data provides a basis for a discussion on which advanced file carving techniques to apply in specific situations. The major novel data points are that the percentage of fragmented files has reduced, the amount of fragmented data has nevertheless increased, and that there is a category of fragmentation that was not yet considered in practice: out-of-order fragmented files. This provides actionable input into designing advanced file carving techniques.

# Chapter 3

# A contemporary investigation of NTFS file fragmentation

*This chapter presents an adapted version of the paper* A contemporary investigation of NTFS file fragmentation*, by Vincent van der Meer, Hugo Jonker, and Jeroen van den Bos, which was published in the special issue "DFRWS APAC 2021 – Proceedings of the First Annual DFRWS APAC Conference" of the journal Forensic Science International: Digital Investigation [DFRWS-APAC21].*

**Abstract** There is a significant amount of research in digital forensics into analyzing file fragments or reconstructing fragmented data. At the same time, there are no recent measurements of fragmentation on current, in-use computer systems. To close this gap, we have analyzed file fragmentation from a corpus of 220 privately owned Windows laptops.

We provide a detailed report of our findings. This includes contemporary fragmentation rates for a wide variety of image, video, office, database, and archiverelated extensions. Our data substantiates the earlier finding that fragments for a significant portion of fragmented files are stored out-of-order. We define metrics to measure the degree of "out-of-orderedness" and find that the average degree of out-of-orderedness is non-negligible. Finally, we find that there is a significant group of fragmented files for which reconstruction is insufficiently addressed by current tooling.

## 3.1 Introduction

File fragmentation impacts (amongst others) file system performance and file recovery. Indeed, many studies in these areas rely on assumptions with respect to fragmentation. In the domain of digital forensics, this includes studies into file fragment classification (e.g., [RNP$^+$17]), generic file carvers (e.g., [YT10; Gar07]) as well as file type specific file carvers (e.g., [DKM19; YXL$^+$17]), and fragment dating (e.g., [BJ19]).

For all such studies, contemporary data on file fragmentation is a necessary prerequisite to determine carving strategies. The most recent large-scale study of file fragmentation is from 2007 by [Gar07], with data gathered from 1998 to 2006. This corpus is now outdated: it concerns mostly FAT-type file systems, while Windows (since XP) by default uses the NTFS file system. Moreover, this corpus concerns deprecated versions of Windows whose combined market share is below 1.75% [web-Net20]

As explained in the previous chapter, to remedy this, we gathered data of file fragmentation on NTFS file systems from 220 laptops. These machines are individually acquired, owned and maintained, and are in regular use by their owners. As these machines were owned by volunteer participants, privacy was paramount. Therefore, we designed a privacy-friendly approach to data gathering [WIFS19]. In the previous chapter, we also presented initial fragmentation findings. Key amongst those was that out-of-order fragmentation occurs fairly frequently – a type of fragmentation that seems to mostly have been overlooked in literature.

This chapter focuses on RQ2, which is stated as: *How are files on real-world, in-use Windows system fragmented?*

**Contributions.** In this chapter, we present in-depth, contemporary data on NTFS file fragmentation. The main contributions are:

- Our corpus provides a contemporary (Oct'18 – Jan'19) view on file fragmentation, based on the Dataset [dataset-vdMee19]
- The number of files in the corpus is significantly larger (2–10 times) than previous works (>1 mln `.jpg`; 14,000 `.doc`; 87,000 `.docx`; . . . ).
- We provide novel metrics on the convolutedness of fragmentation: *degree of internal fragmentation* and *degree of out-of-orderedness.*
- We report on a number of fragmentation characteristics: fragmentation vs. file size, fragmentation vs. used volume space, fragmentation per exten-

sion, gap size for files fragmented in two parts, distribution of number of fragments, correlation between fragmentation and disk size, fragmentation and disk type (primary / secondary).

- We find (amongst others) that the average degree of out-of-orderedness of fragmented files is non-negligible. This has implications for the field of digital forensics.

## 3.2 Results

In this section, we present our results. Note that many of the distributions on which we report are skewed. To provide some insight into the skewedness, we present both average and median values for such distributions.

The results will be presented using the different definitions on fragmentation (primarily def. I and def. IV), where we use the most relevant definition of the degree of fragmentation per context. However, these metrics do not convey how complex the fragmentation of a file is. Two aspects determine the complexity of a file's fragmentation: the number of fragments (relative to the file size) and the order between the fragments. To provide insight into the complexity of fragmentation, we introduce two corresponding metrics: the *percentage of internal fragmentation* to quantify the number of fragments in relation to the file size, and the *percentage of out-of-order'ness* (OoO'ness for short), which quantifies the extent to which the fragments occur out-of-order. Both definitions make use of the number of fragmentation points, which is the number of times a process reading the file sequentially would need to jump over one or more blocks to continue reading the file.

**Definition 2** (% of internal fragmentation)**.** The percentage of internal fragmentation of a file $f$ of at least 2 blocks is the ratio of the number of fragmentation points vs. the number of blocks minus one, i.e.:

$$\text{intfrag}(f) = \frac{\text{fragpoints}(f)}{\text{blocks}(f) - 1} \cdot 100,$$

where blocks($f$) denotes the total number of blocks of file $f$, and fragpoints($f$) is the number of times where, when reading a block of file $f$, the next block of $f$ is not the next block on disk.

For example, a file $f_1$ whose blocks are stored contiguous and in order has 0 fragmentation points and therefore intfrag($f_1$) = 0%. Another example, consider a file $f_2$ of $N$ blocks, where the blocks occur in order, but every block of $f_2$ is followed by a block of another file. In this case, there is a fragmentation point after every block except the last block of the file. Thus, fragpoints$(f_2) = N - 1$, which gives intfrag$(f_2) = \frac{N-1}{N-1} \cdot 100 = 100\%$.

**Definition 3** (% of OoO'ness)**.** The percentage of out-of-order'ness of a fragmented file $f$ is the ratio of the number of times the next fragment occurs prior to the current vs. the total number of fragmentation points, i.e.:

$$\text{OoOness}(f) = \frac{\text{backfragpoints}(f)}{\text{fragpoints}(f)} \cdot 100,$$

with fragpoints$(f)$ defined as before, and where backfragpoints$(f)$ denotes the number of times the next block of file $f$ is stored earlier on disk than the current block.

For example, consider a file $f_3$, of $N$ blocks, which is contiguous, but written backwards. I.e., the second block is the block before the first block; the third block is the block before the second, etc. In this case, every fragmentation point is backwards, hence OoOness$(f_3) = 100\%$. In contrast, OoOness$(f_2) = 0\%$, as file $f_2$ was stored in-order, so backfragpoints$(f_2) = 0$.

Remark that extreme values of OoO'ness correspond to relatively simple cases: an OoO'ness of 100% is a file where the next block is always stored earlier on disk (e.g., $f_3$), and an OoO'ness of 0% concerns a file where the next block is always stored further (e.g., $f_1$). In contrast, an OoO'ness of 50% means half the fragmentation points are backwards – i.e., when reaching the end of a fragment, there is no preference for either forward or backward direction to find the next block. Thus, an average OoO'ness of 50% is a worst-case (with respect to out-of-orderedness) situation for a file carver.

### 3.2.1 Fragmentation per MFT entry type

In Table 3.1, the main fragmentation characteristics of our dataset are presented, split per MFT entry type. For completeness and comparison purposes, we include our previously [WIFS19] reported totals (right column), extended with new measures of average internal fragmentation and average OoO'ness. Remark that both resident files and symbolic links can inherently not fragment. In our dataset, we find that hard-linked files are up to 7 times less likely to be fragmented than the average. Sparse files and NTFS compressed files were already known to be prone to fragmenting; to the best of our knowledge, we are the first to quantify the extent of this. In the dataset, we find that (under definition I) around 10% of both sparse and NTFS compressed files are fragmented. Under a stricter definition of fragmentation, one that only considers files that may potentially fragment (under definition IV), the ratios increase to one in five (NTFS compressed) and close to one in three (sparse), respectively. Finally, note that when NTFS-compressed files are fragmented, the average degree of internal fragmentation is lower than average (8.6% vs. 19.9%).

| | *NTFS-compressed* | *sparse files* | *hardlinks* | *resident files* | *symbolic links* | *MFT entries [WIFS19]* |
|---|---|---|---|---|---|---|
| all | 598,119 | 242,844 | 8,778,592 | 12,639,771 | 1,380,728 | 84,390,537 |
| with data | 597,255 | 97,322 | 8,659,294 | 12,616,364 | – | 82,960,039 |
| with blocks | 597,255 | 97,322 | 8,079,067 | – | – | 70,320,268 |
| with $\geq$ 2 blocks | 367,284 | 75,645 | 5,365,324 | – | – | 42,671,054 |
| fragmented files | 72,351 | 24,079 | 34,720 | – | – | 1,871,109 |
| out-of-order frag. files | 40,660 | 12,156 | 14,259 | – | – | 868,917 |
| *% fragmented* | | | | | | |
| of all | 12.1 % | 9.9 % | 0.4 % | – | – | 2.2 % |
| of those with data | 12.1 % | 24.7 % | 0.4 % | – | – | 2.3 % |
| of those with blocks | 12.1 % | 24.7 % | 0.4 % | – | – | 2.7 % |
| of those with $\geq$ 2 blocks | 19.7 % | 31.8 % | 0.6 % | – | – | 4.4 % |
| *of fragmented files:* | | | | | | |
| out-of-order fragmented | 56.2 % | 50.5 % | 41.1 % | – | – | 46.4 % |
| avg. internal fragmentation | 8.6 % | 12.8 % | 24.3 % | – | – | 19.9 % |
| avg. OoO'ness | 32.5 % | 29.2 % | 24.1 % | – | – | 29.9 % |

Table 3.1: Fragmentation per MFT entry type.

### 3.2.2 Fragmentation per file extension

Table 3.2 provides various data on the fragmentation per extension. In this table, we list the number of files with at least 2 blocks (i.e., the number of files relevant for definition IV), as well as the percentage of files that are fragmented. Specifically, we include both def. I for comparison purposes, and def. IV as most representative definition of fragmentation. Furthermore, like Garfinkel [Gar07], we provide the percentage of fragmented files that are fragmented into 2, 3, and 4 or more parts completely in-order, and similar for files that are fragmented at least partially out-of-order. For the fragmented files, we also provide the average internal fragmentation (definition 2), the average OoO'ness (definition 3), as well as the average number of fragments.

**Images.** Fragmented images are often fragmented out-of-order. For fragmented `bmp`, `png`, and `raw` files, the percentage of fragmented files that are fragmented out-of-order are 44.1%, 38.0% and 37.5%, respectively. For all other image formats, fragmented files are more likely to be fragmented out-of-order than in-order.

**Videos.** Yang et al. [YXL+17] claim `avi` files are more likely to be fragmented than other files. Our dataset does not corroborate this. We find that the average fragmentation rate for `avi` files (1.8%) is lower than the general average (4.4%). However, when `avi` files are fragmented, the number of fragments is often large (average of 40.8 fragments).

The `.mts` format is a video format typically used in camcorders. In our dataset, 2 systems account for 1,555 of the 1,591 `mts` files.

**Office documents.** Interestingly, Outlook `pst` files are often fragmented (35.8%). The number of fragments is low, leading to a negligible rate of internal fragmentation. The main complexity in recovering fragmented `pst` files is due out-of-orderedness. Another interesting document-related finding is that `pdf` files have a higher fragmentation rate than the word-processing extensions `rtf` (Wordpad), `odt` (OpenOffice), `doc` and `docx` (MS Word); an unexpected result considering `pdf` files are typically static, i.e., not intended for editing.

**Databases.** Ji et al. [JCH+18] studied fragmentation on Android systems and found that database files are prone to fragmentation, due to concurrent and frequent growth. Our dataset shows that this is true on NTFS systems as well: all database extensions are fragmented above average.

| | | | | *% of fragmented files with . . . fragments:* | | | | | | | | *of fragmented files:* | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *# files with* | *% fragmented* | | *in-order* | | | | *out-of-order* | | | | *avg. %* | *avg. %* | *avg. #* |
| *ext* | *≥2 blocks* | *def. I* | *def. IV* | all | *2* | *3* | *≥4* | all | *2* | *3* | *≥4* | *intfrag* | *OoO'ness* | *fragments* |
| *Image* | | | | | | | | | | | | | | |
| bmp | 70,425 | 1.6 | 2.5 | 55.8 | 40.7 | 9.7 | 5.4 | 44.2 | 14.6 | 10.7 | 18.8 | 10.3 | 29.2 | 3.2 |
| gif | 276,241 | 0.8 | 1.8 | 53.6 | 40.4 | 7.9 | 5.3 | 46.4 | 10.3 | 9.0 | 27.0 | 28.4 | 26.4 | 3.6 |
| jpeg | 13,774 | 8.5 | 8.7 | 39.7 | 24.8 | 8.8 | 6.0 | 60.3 | 10.6 | 9.1 | 40.7 | 13.6 | 33.6 | 3.8 |
| jpg | 1,043,198 | 2.7 | 3.1 | 42.6 | 32.6 | 6.2 | 3.8 | 57.4 | 13.8 | 10.5 | 33.1 | 12.4 | 33.5 | 4.4 |
| png | 2,389,752 | 0.9 | 3.1 | 62.0 | 48.9 | 9.5 | 3.5 | 38.0 | 14.4 | 10.2 | 13.4 | 32.9 | 25.6 | 2.8 |
| psd | 7,022 | 4.5 | 4.5 | 40.4 | 31.0 | 7.8 | 1.6 | 59.6 | 16.6 | 17.2 | 25.7 | 6.8 | 37.2 | 9.3 |
| psp | 422 | 4.6 | 6.2 | 42.3 | 15.4 | 15.4 | 11.5 | 57.7 | 3.8 | 7.7 | 46.2 | 5.7 | 24.8 | 8.7 |
| raw | 5,246 | 1.1 | 1.2 | 62.5 | 57.8 | 4.7 | 0.0 | 37.5 | 12.5 | 18.8 | 6.3 | 3.5 | 24.9 | 17.1 |
| tif | 6,309 | 9.3 | 9.7 | 37.7 | 13.3 | 18.7 | 5.7 | 62.3 | 3.6 | 19.7 | 39.0 | 4.8 | 31.3 | 4.1 |
| *Video* | | | | | | | | | | | | | | |
| avi | 9,800 | 1.8 | 1.8 | 14.1 | 9.6 | 1.7 | 2.8 | 85.9 | 0.0 | 1.1 | 84.7 | 1.1 | 29.9 | 40.8 |
| flv | 332 | 26.8 | 26.8 | 11.2 | 6.7 | 3.4 | 1.1 | 88.8 | 1.1 | 4.5 | 83.1 | 1.5 | 38.5 | 29.8 |
| mkv | 2,404 | 2.7 | 3.1 | 46.7 | 44.0 | 2.7 | 0.0 | 53.3 | 12.0 | 8.0 | 33.3 | 0.1 | 32.4 | 6.8 |
| mov | 4,459 | 4.3 | 4.4 | 32.0 | 30.9 | 0.5 | 0.5 | 68.0 | 13.9 | 13.9 | 40.2 | 0.5 | 39.0 | 20.2 |
| mp4 | 38,007 | 6.4 | 6.5 | 39.5 | 31.3 | 5.7 | 2.5 | 60.5 | 14.4 | 11.2 | 35.0 | 1.0 | 36.7 | 28.8 |
| mpg | 3,269 | 0.4 | 0.4 | 15.4 | 0.0 | 15.4 | 0.0 | 84.6 | 7.7 | 0.0 | 76.9 | 1.9 | 42.0 | 21.8 |
| mts | 1,591 | 0.2 | 0.2 | 33.3 | 33.3 | 0.0 | 0.0 | 66.7 | 0.0 | 33.3 | 33.3 | 0.0 | 52.4 | 4.3 |
| wmv | 27,328 | 0.7 | 0.7 | 36.7 | 33.2 | 3.6 | 0.0 | 63.3 | 35.7 | 10.7 | 16.8 | 1.8 | 50.1 | 5.8 |
| *Office* | | | | | | | | | | | | | | |
| doc | 14,831 | 5.1 | 5.5 | 40.8 | 21.4 | 9.7 | 9.7 | 59.2 | 8.5 | 13.4 | 37.3 | 15.8 | 31.4 | 5.1 |
| docx | 87,077 | 6.0 | 6.2 | 49.4 | 35.1 | 8.6 | 5.6 | 50.6 | 13.5 | 9.7 | 27.4 | 16.5 | 30.2 | 4.6 |
| msg | 7,120 | 0.7 | 6.2 | 75.7 | 75.7 | 0 0 | 0.0 | 24.3 | 23.6 | 0.0 | 0.7 | 38.2 | 24.1 | 2.1 |
| odt | 2,147 | 4.8 | 4.9 | 57.1 | 44.8 | 6.7 | 5.7 | 42.9 | 23.8 | 7.6 | 11.4 | 35.9 | 33.7 | 2.8 |
| pdf | 92,117 | 7.9 | 8.1 | 30.4 | 14.6 | 6.1 | 9.7 | 69.6 | 6.7 | 8.9 | 53.9 | 7.3 | 33.6 | 9.3 |
| ppt | 3,406 | 7.9 | 8.0 | 10.3 | 7.0 | 0.0 | 3.3 | 89.7 | 5.1 | 1.5 | 83.1 | 3.0 | 37.4 | 10.9 |
| pptx | 17,846 | 11.6 | 11.7 | 17.5 | 8.7 | 2.5 | 6.3 | 82.5 | 5.5 | 4.2 | 72.8 | 3.4 | 36.3 | 19.2 |
| prf | 1,113 | 0.9 | 4.6 | 66.7 | 66.7 | 0.0 | 0.0 | 33.3 | 23.5 | 0.0 | 9.8 | 31.6 | 26.8 | 2.4 |
| pst | 120 | 33.1 | 35.8 | 58.1 | 55.8 | 2.3 | 0.0 | 41.9 | 9.3 | 23.3 | 9.3 | 0.0 | 24.7 | 2.8 |
| rtf | 80,977 | 0.9 | 1.0 | 49.0 | 39.4 | 3.5 | 6.1 | 51.0 | 29.9 | 9.7 | 11.4 | 6.7 | 40.7 | 3.5 |
| xls | 8,550 | 2.0 | 2.3 | 36.1 | 22.2 | 5.7 | 8.2 | 63.9 | 5.2 | 13.9 | 44.8 | 13.9 | 33.8 | 5.2 |
| xlsx | 17,721 | 4.1 | 4.1 | 64.7 | 48.6 | 12.3 | 3.8 | 35.3 | 16.6 | 8.6 | 10.1 | 27.3 | 30.4 | 3.3 |
| *Database* | | | | | | | | | | | | | | |
| accdb | 1,450 | 12.0 | 12.0 | 14.9 | 8.6 | 3.4 | 2.9 | 85.1 | 2.9 | 13.8 | 68.4 | 4.7 | 40.6 | 30.0 |
| db | 33,320 | 12.0 | 17.4 | 39.6 | 28.2 | 7.8 | 3.5 | 60.4 | 8.8 | 9.2 | 42.5 | 19.5 | 32.2 | 24.5 |
| mdb | 11,052 | 3.8 | 6.1 | 33.0 | 21.1 | 7.9 | 4.0 | 67.0 | 14.0 | 13.5 | 39.5 | 9.8 | 39.2 | 5.1 |
| sqlite | 7,959 | 26.2 | 27.8 | 52.0 | 44.3 | 5.6 | 2.2 | 48.0 | 20.5 | 7.1 | 20.4 | 9.0 | 33.2 | 6.9 |
| *Archive* | | | | | | | | | | | | | | |
| 7z | 3,568 | 12.2 | 18.1 | 68.1 | 58.5 | 7.7 | 1.9 | 31.9 | 6.7 | 9.9 | 15.3 | 49.7 | 19.0 | 31.8 |
| gz | 48,900 | 1.8 | 3.7 | 60.7 | 33.4 | 21.0 | 6.2 | 39.3 | 5.9 | 12.5 | 20.9 | 56.2 | 20.5 | 6.4 |
| rar | 3,589 | 7.3 | 7.5 | 21.1 | 13.7 | 4.8 | 2.6 | 78.9 | 5.2 | 7.4 | 66.3 | 3.5 | 34.5 | 48.1 |
| zip | 53,919 | 7.9 | 11.2 | 38.3 | 22.9 | 7.6 | 7.9 | 61.7 | 8.5 | 7.7 | 45.5 | 15.9 | 30.4 | 22.4 |

Table 3.2: Fragmentation per extension (categorised)

### 3.2.3 Fragmentation in relation to file size

| *File size*[†] | *# fragmented files* | *% frag* |
|---|---:|---:|
| min[⋆]– 10 kB | 11,531,201 | 1.8 |
| 10– 50 kB | 15,669,438 | 3.9 |
| 50 – 100 kB | 4,468,221 | 4.9 |
| 100 – 500 kB | 6,490,196 | 7.4 |
| 0.5 – 1 MB | 1,573,008 | 6.8 |
| 1 – 5 MB | 2,096,812 | 8.2 |
| 5 – 10 MB | 397,872 | 7.8 |
| 10 – 50 MB | 341,782 | 9.5 |
| 50 – 100 MB | 45,148 | 14.0 |
| 100 – 500 MB | 44,534 | 21.4 |
| $>$ 500 MB | 12,842 | 46.1 |

[⋆] min: 2 assigned blocks, irrespective of file size and block size.
[†] kB = 1,000 bytes, MB = 1,000,000 bytes.

Table 3.3: Fragmented files per file size

Tables 3.3 and 3.4 show fragmentation and fragment properties split out in file size intervals. The ranges include start point, and exclude the end point. With regards to the smallest file that may be fragmented: this is dependent on the number of allocated blocks. Note that allocated blocks do not need to be filled. Indeed, we found 10 fragmented files, whose file size was 1 byte.

Table 3.3 shows that smaller files occur more often than larger files. Note that 74% of all files of at least two blocks are smaller than 100 kB. Furthermore, we make the following observations:

- Of all fragmented files with a file size between 1 and 100 MB, over 75% is fragmented out-of-order.
- As file size increases, the number of fragments typically increases (though this correlation is not perfect).
- For files >50 kB, the average OoO'ness is slightly over a third, more or less irrespective of the file size. This means that at each fragment boundary, there is, on average, a probability of about $\frac{1}{3}$ that the next fragment is located before the current fragment, and a probability of about $\frac{2}{3}$ of the next fragment being ahead.
- We found that some files are extremely fragmented, such as one file split into 20,000 fragments. This skews the average, but the median value of the

| | | | | # fragments | |
|---|---|---|---|---|---|
| *File size*[†] | *% OoO* | *% intfrag* | *% OoO'ness* | *avg* | *median* |
| min[⋆]– 10 kB | 18.7 | 77.0 | 18.2 | 2.0 | 2 |
| 10– 50 kB | 29.7 | 26.0 | 24.9 | 2.3 | 2 |
| 50 – 100 kB | 47.1 | 10.7 | 32.2 | 2.8 | 2 |
| 100 – 500 kB | 57.5 | 5.5 | 34.3 | 3.6 | 3 |
| 0.5 – 1 MB | 70.2 | 2.9 | 37.3 | 5.5 | 4 |
| 1 – 5 MB | 76.2 | 2.2 | 38.1 | 10.0 | 5 |
| 5 – 10 MB | 80.4 | 1.5 | 37.4 | 23.1 | 7 |
| 10 – 50 MB | 82.6 | 1.3 | 37.2 | 49.2 | 12 |
| 50 – 100 MB | 76.7 | 1.3 | 33.7 | 126.3 | 14 |
| 100 – 500 MB | 66.2 | 0.6 | 35.9 | 156.8 | 3 |
| > 500 MB | 74.1 | 0.1 | 36.3 | 93.1 | 4 |

[⋆] min: 2 assigned blocks, irrespective of file size and block size.
[†] kB = 1,000 bytes, MB = 1,000,000 bytes.

Table 3.4: Fragmentation characteristics of fragmented files versus file size

range is less affected and provides a more nuanced view on the number of fragments.

### 3.2.4 Distribution of the number of fragments

Table 3.5 shows how many files are split into $N$ parts, in percentages of the total number of fragmented files (rounded to two decimals, hence the numbers do not sum precisely to 100%).

Note that the majority of fragmented files are fragmented into two parts (56.76%), most of which are fragmented in-order. Furthermore, remark that for all files split into 3 or more fragments, out-of-order fragmentation occurs (much) more frequently than in-order fragmentation.

In Table 3.6, we extend upon Table 3.5 with file size and gap size information.

In-order bi-fragmented files are common amongst fragmented files, they constitute 41.84% of all fragmented files. Theoretically, as files are fragmented into more parts, it is increasingly less likely that all fragments occur in order. Our dataset corroborates this.

Files split into 11 fragments or more are over 14× more often fragmented out-of-order than in-order. Files split up in 100 parts or more make up for 0.6% of all fragmented files in the dataset.

Finally, note that the average OoO'ness is hardly correlated with the number

| *# Fragments* | *Total* | *In-order* | *Out-of-Order* |
|---:|---:|---:|---:|
| 2 | 56.76 % | 41.84 % | 14.92 % |
| 3 | 18.21 % | 7.92 % | 10.29 % |
| 4 | 8.58 % | 2.15 % | 6.43 % |
| 5 | 5.02 % | 0.77 % | 4.25 % |
| 6 | 2.96 % | 0.29 % | 2.67 % |
| 7 | 1.40 % | 0.13 % | 1.27 % |
| 8 | 0.97 % | 0.07 % | 0.90 % |
| 9 | 0.69 % | 0.04 % | 0.65 % |
| 10 | 0.53 % | 0.04 % | 0.49 % |
| $\geq$ 11 | 4.89 % | 0.34 % | 4.55 % |

Table 3.5: Distribution of number of fragments (excluding non-fragmented files)

of fragments. For any file fragmented into three or more fragments, average OoO'ness is yet again roughly a third.

| | | | | *Sum of all gap sizes (in blocks) (carving-distance)* | | | |
|---:|---:|---:|---:|---:|---:|---:|---:|
| *#fragments* | *#files* | *% OoO* | *average OoO'ness* | *min* | *average* | *median* | *max* |
| 2 | 1,062,539 | 26.3 | 26.3% | 1 | 7,038,401 | 711,673 | 517,861,056 |
| 3 | 340,422 | 56.5 | 32.8% | 2 | 15,594,100 | 5,406,252 | 990,207,960 |
| 4 | 160,472 | 74.9 | 34.9% | 3 | 25,645,754 | 11,833,174 | 811,066,168 |
| 5 | 93,835 | 84.6 | 35.5% | 4 | 34,104,018 | 16,748,220 | 969,975,568 |
| 6–10 | 122,388 | 91.5 | 36.7% | 5 | 53,979,693 | 28,003,492 | 2,002,994,256 |
| 11–20 | 45,031 | 93.5 | 36.8% | 11 | 90,567,873 | 47,230,761 | 3,037,661,708 |
| 21–100 | 35,890 | 92.4 | 36.8% | 42 | 227,973,821 | 96,196,576 | 9,852,412,280 |
| 101–1000 | 9,721 | 93.6 | 34.6% | 399 | 1,194,774,735 | 345,806,721 | 69,129,433,312 |
| 1001+ | 811 | 96.4 | 30.3% | 17,636 | 5,760,340,498 | 948,806,352 | 270,488,355,485 |

Table 3.6: Distribution of number of fragments per file (expanded)

### 3.2.5 Gapsize distribution of bi-fragmented files

For in-order fragmented files, the gap between two consecutive fragments is unambiguously defined as the distance from the last block ("tail") of the first, to the first block ("head") of the second. For out-of-order files, there is not one, unique, unambiguous definition of the distance between two consecutive fragments. Note that since Garfinkel's study does not consider out-of-order fragmented files, a direct comparison is not possible.

Figure 3.1 depicts three possible metrics. All three metrics have their applications. Note that when looking at in-order fragmented files, these three metrics are equivalent. It is only when the next fragment appears before the current fragment that differences arise.

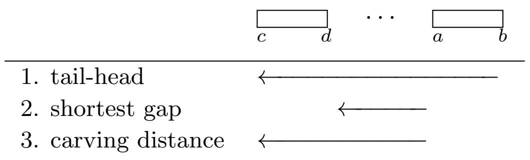

Figure 3.1: Possible metrics for gapsize of OoO fragmented files

The first, tail-head distance, covers the total length to be covered, but includes the length of both fragments themselves. For file carving, this is not that useful: once the first fragment is found, this will be skipped when searching for further fragments. This metric therefore will not directly support finding the start of a new fragment. The second metric, shortest gap distance, measures the shortest distance between the two fragments, which only makes sense if both fragments are known. The third metric, carving distance, measures the distance an out-of-order file carver would have to make. This includes the fragment length of the unknown fragment, but skips the already-found fragment.

Figure 3.2 depicts the number of in-order bi-fragmented files with a distance of 1 to 300 blocks. The part shown in the figure covers 10.0% of all distances between the fragments of in-order bi-fragmented files. The large trends depicted in the figure hold over the entire range; in particular, we found that distances in general decline, with a generic exception for gapsize distances that are a power of two (see also Table 3.9).

We evaluated all three distances for out-of-order bi-fragmented files. We found that there are only small deviations between them. Interestingly, the peaks at distances of powers of two as seen for in-order files occurred much more strongly for carving distance than for the other two distance metrics. Hence, from here on out we will use this metric for the gapsize of out-of-order files.

Figure 3.3 depicts the carving distances for out-of-order files. As was the case for in-order files, the main trends depicted in the figure continue across the entire range. The gapsizes depicted in the figure cover 5.0% of all out-of-order bi-fragmented files.

Lastly, concerning the aforementioned preference for gap lengths of powers of two: note that these gap lengths are not necessarily aligned with specific locations on disk. More specifically, the length of the first fragment determines the gap start. This preference for gap lengths of powers of two thus seems to be an artefact of how NTFS assigns blocks. Consequently, the fact that carving distance aligns well with these observations suggests that carving distance aligns with how NTFS

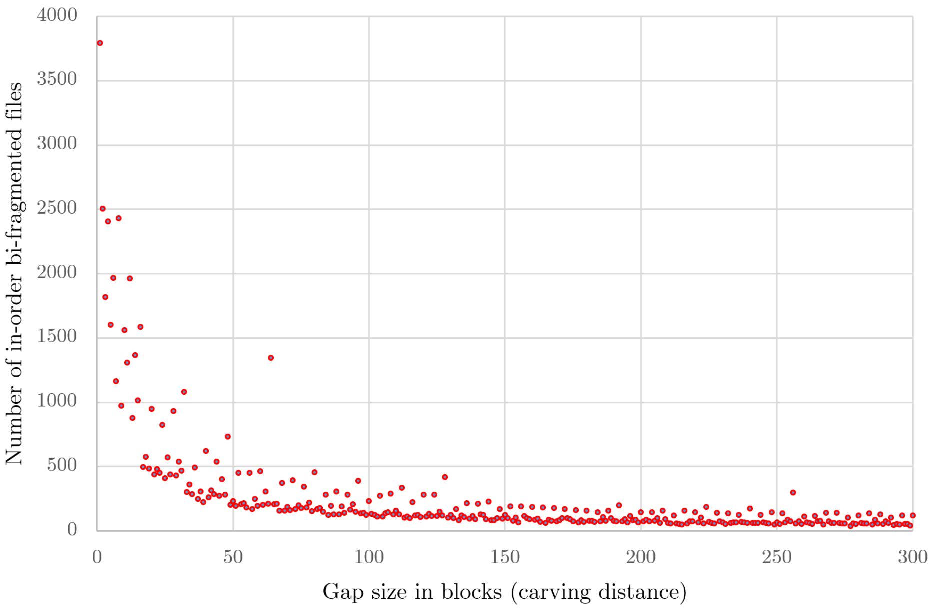

Figure 3.2: Gap-size distribution of in-order 2-fragmented files

allocates blocks.

### 3.2.6 Percentage of used volume space and file fragmentation

As a volume becomes more filled with data, the remaining unallocated space becomes progressively more scarce and more likely to be fragmented. This may impact for the degree of fragmentation. For example, Ji et al. [JCH$^{+}$18] concluded from their study of Android devices that the degree of fragmentation is highly correlated with the percentage of used volume space.

We examined this in our dataset. First of all, we excluded volumes with very few files ($\leq 15$), as we do not consider such volumes to be in active daily use (but act e.g., as recovery partition). Moreover, they contain so few files, that even a single fragmentation on such a volume will strongly skew the fragmentation rate, and thus, strongly affect the correlation. For example, in our dataset there are 44 volumes that each contain 3 files, one of which is fragmented (i.e., a fragmentation percentage of 33%).

Given these constraints, we find a moderate positive relation between data

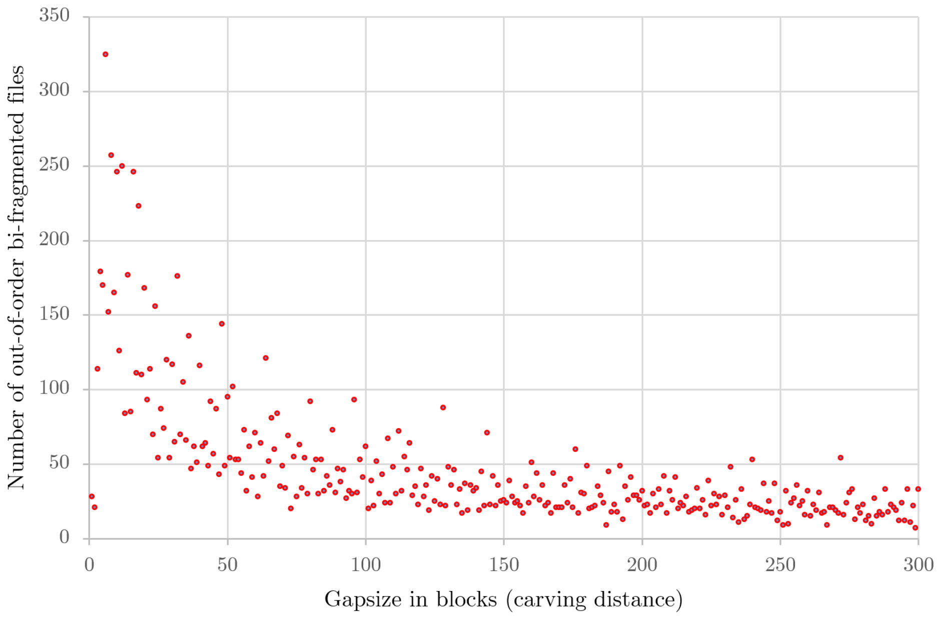

Figure 3.3: Gap-size distribution of out-of-order 2-fragmented files

fragmentation and the percentage of used volume space, using Pearson's correlation coefficient as a measure. For SDDs we find that the correlation is 0.462, and for HDDs the correlation is 0.464. Though the correlation coefficients are nearly identical, the underlying data distribution is rather different, as shown in Figure 3.4.

### 3.2.7 Fragmentation per storage device

For non-dual-boot systems, we distinguished between primary (boot disk) and secondary storage devices within our dataset based on file count and extension occurrence. This is possible as a Windows install has roughly 80,000 files, with many system-related extensions such as `.dll` and `.com`. For every non-dual-boot system in our dataset, these heuristics provided a clear division between primary and secondary storage device.

By default, Windows has a scheduled defragment-task, with different schedules for SSDs (monthly) and HDDs (weekly). The defragmentation strategy can differ per storage device [web-MS20].

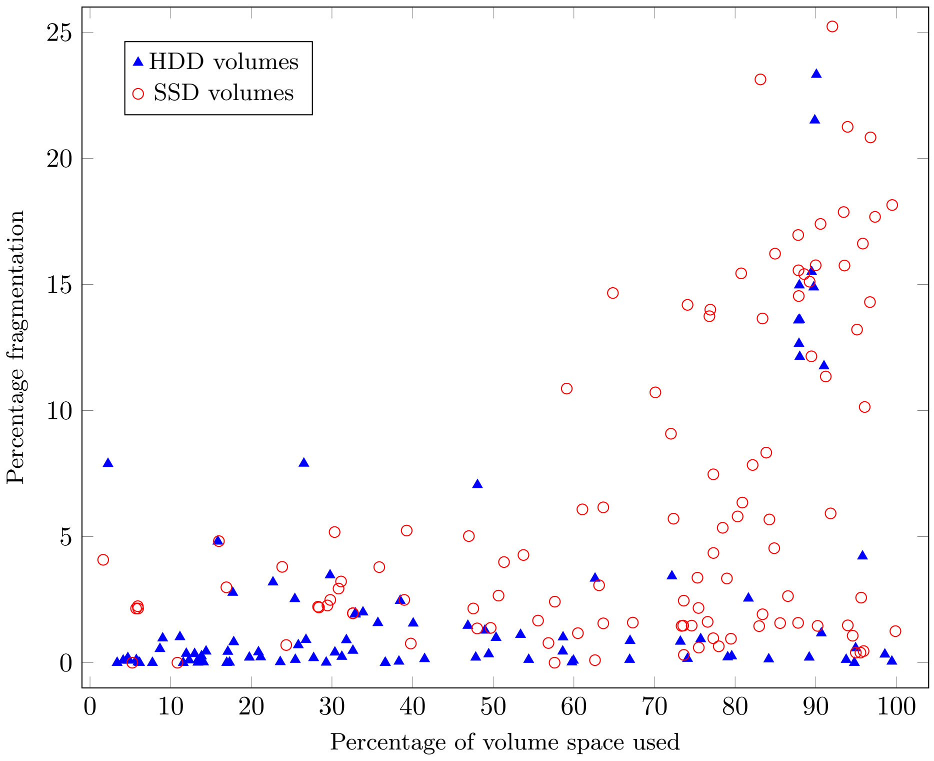

Figure 3.4: Fragmentation vs. used volume space

Table 3.7 shows that single disk SSD-systems are more fragmented than single disk HDD-systems, on average 2.4 times more. The most common system configuration is a SSD/HDD combination. In this configuration, the primary SSDs are way more fragmented than secondary HDDs, on average 5.2 times more. Note that in this dataset there was no system with a dual HDD configuration.

With respect to the popularity of SSDs versus HDDs: in our dataset, we find that 84% of the laptops use an SSD, and 67% uses an HDD.

### 3.2.8 Other extremes and curiosa

- In our dataset, there are 2,914 file extensions for which no file happened to be fragmented. The top 10 most occurring of these is listed in Table 3.8.
- Among the extremely fragmented files (files with thousands fragments or

| *Storage Device* | *#* | *average frag* | *median frag* |
|---|---|---|---|
| Single disk (SSD) | 67 | 5.6 % | 2.0 % |
| Single disk (HDD) | 36 | 2.3 % | 1.2 % |
| Primary disk (SSD) | 113 | 7.3 % | 4.7 % |
| Secundary disk (HDD) | 110 | 1.4 % | 0.2 % |
| Secundary disk (SSD) | 3 | 4.1 % | 3.9 % |

Table 3.7: Fragmentation per storage device

more), the most frequent occurring extensions are `exe`, `log`, `xml`, `dat`, and `dll`.

- The most fragmented file is a 2 GB `.bin` file that is split up in 20,464 fragments.

- Of all the 1,871,109 fragmented files, only 8 are fragmented contiguous out-of-order. All these 8 files are bi-fragmented.

- Some files occupy vastly more blocks than their file size requires. One file in our corpus had a 1 byte file size, yet had 369 blocks allocated on disk. Moreover, this 1 byte file was fragmented (out-of-order) into 5 fragments.

- Table 3.9 presents the frequency of gap sizes (measured in blocks) of powers of 2 for bi-fragmented files in our dataset. For comparison, we also show the incidence for adjacent gapsizes. The table points out that gap sizes corresponding with powers of 2 act as local maxima.

### 3.2.9 Auxiliary data per file extension

In Table 3.10, we provide auxiliary data on file sizes. The right-hand side of this table focuses on NTFS-compressed and sparse files. Recovery of such files is complex, irrespective of whether they are fragmented or not. Therefore, results concerning these file types in Table 3.10 are reported on all files with blocks (definition III), and not only files that could fragment (definition IV).

| | | | *File size in bytes* | | |
|---|---|---|---|---|---|
| *Ext* | *# files (def. IV)* | *# systems* | *Avg.* | *Median* | *St. Dev.* |
| `ctt` | 51,315 | 28 | 7,596 | 4,904 | 5,545 |
| `ovl` | 47,998 | 14 | 313,040 | 33,599 | 3,716,227 |
| `p7x` | 36,030 | 99 | 10,693 | 10,653 | 733 |
| `tt` | 24,032 | 47 | 27,017 | 26,786 | 3,466 |
| `anm` | 23,807 | 12 | 42,248 | 22,522 | 125,422 |
| `vcd` | 21,288 | 8 | 1,959 | 1,547 | 4,509 |
| `prx` | 20,885 | 197 | 11,889 | 4,286 | 241,195 |
| `slp` | 20,461 | 7 | 926,198 | 63,459 | 3,236,241 |
| `p7s` | 17,252 | 42 | 9,834 | 9,355 | 2,036 |
| `ovs` | 16,310 | 8 | 3,735,952 | 51,213 | 8,313,296 |

Table 3.8: Top 10 most frequently occurring extensions without fragmented files

| *Gap* | *# Files* | *Gap* | *# Files* | *Gap* | *# Files* |
|---|---|---|---|---|---|
| $2^0$ | 3793 | | | | |
| $2^1$ | 2507 | ... | ... | ... | ... |
| 3 | 1819 | 63 | 212 | 1023 | 28 |
| $2^2$ | 2408 | $2^6$ | 1346 | $2^{10}$ | 78 |
| 5 | 1603 | 65 | 210 | 1025 | 23 |
| ... | ... | ... | ... | ... | ... |
| 7 | 1165 | 127 | 121 | 2047 | 23 |
| $2^3$ | 2431 | $2^7$ | 421 | $2^{11}$ | 43 |
| 9 | 972 | 129 | 106 | 2049 | 11 |
| ... | ... | ... | ... | ... | ... |
| 15 | 1016 | 255 | 77 | 4095 | 10 |
| $2^4$ | 1587 | $2^8$ | 300 | $2^{12}$ | 20 |
| 17 | 498 | 257 | 57 | 4097 | 4 |
| ... | ... | ... | ... | ... | ... |
| 31 | 469 | 511 | 39 | 8191 | 3 |
| $2^5$ | 1083 | $2^9$ | 138 | $2^{13}$ | 7 |
| 33 | 302 | 513 | 46 | 8193 | 2 |

Table 3.9: In-order bi-fragmented gap sizes around powers of 2.

| | *# systems* | | *file size in bytes using def. IV:* | | | | *using def. III:* | | |
|---|---|---|---|---|---|---|---|---|---|
| *ext* | *def. III* | *def. IV* | *avg.* | *median* | *st. dev.* | *max.* | *# files* | *% NTFS-compressed* | *% sparse* |
| *Images* | | | | | | | | | |
| bmp | 214 | 213 | 380,653 | 36,176 | 4,603,975 | 1,150,221,432 | 105,371 | 0.1 | 0.0 |
| gif | 214 | 214 | 73,113 | 14,878 | 587,069 | 67,859,584 | 468,406 | 0.4 | 0.0 |
| jpeg | 181 | 177 | 430,900 | 132,696 | 921,712 | 19,905,785 | 14,137 | 5.1 | 0.1 |
| jpg | 215 | 215 | 469,789 | 43,499 | 1,383,941 | 202,187,275 | 1,157,750 | 1.7 | 0.1 |
| png | 215 | 215 | 77,909 | 13,385 | 819,778 | 443,815,127 | 6,551,794 | 0.6 | 0.0 |
| psd | 156 | 156 | 6,098,161 | 318,875 | 25,696,786 | 657,852,455 | 7,111 | 1.1 | 0.0 |
| psp | 68 | 67 | 2,913,965 | 164,864 | 13,461,247 | 79,354,648 | 444 | 5.0 | 0.0 |
| raw | 207 | 200 | 4,943,742 | 38,144 | 22,211,217 | 340,245,502 | 6,038 | 0.0 | 0.0 |
| tif | 188 | 178 | 2,174,293 | 178,288 | 9,322,608 | 536,980,180 | 6,512 | 0.8 | 0.0 |
| *Videos* | | | | | | | | | |
| avi | 199 | 199 | 20,822,230 | 730,952 | 105,051,447 | 1,886,142,464 | 9,805 | 0.1 | 0.4 |
| flv | 34 | 34 | 25,612,283 | 3,670,220 | 83,269,828 | 911,348,494 | 332 | 0.0 | 0.0 |
| mkv | 206 | 206 | 250,121,103 | 109,239,726 | 351,135,887 | 1,994,939,880 | 2,406 | 0.1 | 0.2 |
| mov | 95 | 95 | 51,779,439 | 15,788,157 | 119,573,812 | 1,925,087,760 | 4,478 | 1.6 | 0.0 |
| mp4 | 214 | 214 | 55,803,648 | 2,005,846 | 211,305,187 | 1,998,753,571 | 38,155 | 2.4 | 0.1 |
| mpg | 92 | 92 | 2,802,244 | 569,095 | 33,224,606 | 1,644,236,800 | 3,269 | 0.0 | 0.0 |
| mts | 8 | 7 | 168,426,360 | 113,362,944 | 191,510,208 | 1,893,931,008 | 1,747 | 0.0 | 0.0 |
| wmv | 206 | 206 | 4,187,920 | 398,973 | 47,347,382 | 1,892,176,290 | 27,382 | 0.0 | 0.0 |
| *Office* | | | | | | | | | |
| doc | 214 | 214 | 480,189 | 43,520 | 9,447,633 | 1,000,000,000 | 15,666 | 0.7 | 0.0 |
| docx | 214 | 214 | 371,838 | 29,359 | 2,032,033 | 117,328,214 | 87,124 | 2.8 | 0.0 |
| msg | 213 | 213 | 28,180 | 4,823 | 111,153 | 4,486,144 | 36,296 | 0.7 | 0.0 |
| odt | 140 | 140 | 152,304 | 16,500 | 818,144 | 24,663,942 | 2,147 | 1.1 | 0.0 |
| pdf | 215 | 215 | 2,619,014 | 462,167 | 12,769,129 | 695,725,963 | 93,265 | 1.1 | 0.0 |
| ppt | 210 | 210 | 1,462,596 | 802,816 | 2,396,460 | 35,269,926 | 3,406 | 0.6 | 0.0 |
| pptx | 211 | 211 | 4,711,035 | 1,089,065 | 16,722,112 | 871,334,541 | 17,851 | 1.3 | 0.1 |
| prf | 119 | 118 | 15,741 | 8,405 | 101,644 | 3,145,728 | 3,156 | 0.2 | 0.0 |
| pst | 31 | 27 | 152,551,442 | 173,720,576 | 213,757,822 | 1,896,784,896 | 125 | 2.4 | 12.8 |
| rtf | 214 | 214 | 183,797 | 82,239 | 1,051,434 | 77,456,537 | 90,604 | 0.3 | 0.1 |
| xls | 207 | 207 | 252,638 | 67,072 | 642,497 | 15,325,184 | 9,891 | 0.1 | 0.0 |
| xlsx | 214 | 214 | 205,915 | 17,573 | 4,159,236 | 307,409,090 | 17,729 | 1.1 | 0.0 |
| *Databases* | | | | | | | | | |
| accdb | 190 | 190 | 2,029,463 | 724,992 | 6,719,223 | 145,084,416 | 1,450 | 2.1 | 0.0 |
| db | 215 | 215 | 4,075,278 | 74,752 | 54,485,792 | 1,988,837,638 | 41,762 | 1.5 | 7.7 |
| mdb | 175 | 175 | 233,414 | 31,773 | 764,419 | 18,874,368 | 14,762 | 5.6 | 0.2 |
| sqlite | 212 | 212 | 782,992 | 65,536 | 7,712,433 | 454,340,608 | 8,245 | 1.8 | 9.6 |
| *Archives* | | | | | | | | | |
| 7z | 201 | 201 | 37,437,492 | 112,778 | 143,243,604 | 1,926,983,279 | 5,170 | 0.7 | 0.0 |
| gz | 213 | 213 | 243,467 | 10,277 | 7,615,496 | 816,336,896 | 84,665 | 1.7 | 0.0 |
| rar | 161 | 161 | 49,886,452 | 5,883,486 | 156,451,000 | 1,927,419,308 | 3,667 | 0.4 | 0.1 |
| zip | 217 | 217 | 18,503,962 | 168,076 | 102,922,058 | 1,988,366,193 | 67,884 | 0.3 | 0.0 |

Table 3.10: Meta information per extension (categorised)

## 3.3 Discussion

### 3.3.1 Overall fragmentation rates

The overall fragmentation rates (Sec. 3.2.1) have implications for file carvers. First, an upside for file recovery tooling: most files are not fragmented. This means that file recovery tools which ignore fragmentation (which are far easier to construct) will recover most files. Indeed, various studies assume that files are not fragmented, such as [GJ17; SZ12].

However, there is also a downside: out-of-order files constitute close to half of all fragmented files. This means that any tool that aims to recover fragmented files, *must* account for out-of-order fragmentation. This impacts existing studies. For example, neither the file carver due to Garfinkel [Gar07] nor the file carver for fragmented `jpg` files due to Abdullah et al. [AIM+13] account for out-of-order fragmentation.

### 3.3.2 Fragmentation per extension

The general trend of less fragmentation compared to previous studies extends also to specific files. In Table 3.11, we compare our findings to those reported in the 2007 study by Garfinkel. Note that `jpeg` and `jpg` file formats are equivalent, but they use a different extension. The same holds true for the `mpeg` and `mpg` file format. In Table 3.11, we compare Garfinkel's findings against our ratio as determined by def. IV (fragmentable files). We find a lower fragmentation rate across all extensions. Note that since this observation is true for def. IV, it is also true for the other three (less strict definitions) of file fragmentation.

### 3.3.3 Implications for file carving

Although file fragmentation is a topic that attracts some interest in the digital forensics research community, most popular file carvers used in practice focus almost exclusively on recovering unfragmented files. This is an understandable choice given the considerable time it takes to carve large disks and other media even in the simplest scenarios.

This study makes it possible for developers and users of file carvers to make informed choices about the type of recovery they implement and use. It allows an assessment of the added benefits of actually using bifragment gapcarving and whether to extend such an algorithm to include out-of-order fragments or extend it to something else, such as reconstructing files containing multiple gaps.

| | *# files reported* | | *% fragmented* | |
|---|---|---|---|---|
| *file type* | 2007, [Gar07] | 2020, def. IV | 2007 | 2020 |
| *Image* | | | | |
| `bmp` | 26,018 | 70,425 | 8 | 2.5 |
| `gif` | 357,713 | 276,241 | 8 | 1.8 |
| `jpeg` | 108,539 | 13,775 | 16 | 8.7 |
| `jpg` | – | 1,043,198 | – | 3.1 |
| `png` | 9,995 | 2,389,752 | 5 | 3.1 |
| *Office* | | | | |
| `doc` | 7,673 | 14,831 | 17 | 5.5 |
| `ppt` | 1,120 | 3,406 | 8 | 8.0 |
| `pst` | 70 | 120 | 58 | 35.8 |
| `xls` | 2,159 | 8,550 | 11 | 2.3 |
| *Video* | | | | |
| `avi` | 998 | 9,800 | 20 | 1.8 |
| `mpeg` | 168 | 9 | 17 | 11.1 |
| `mpg` | – | 3,269 | – | 0.4 |
| *Database* | | | | |
| `mdb` | 402 | 11,052 | 27 | 6.1 |

Table 3.11: Comparison of fragmentation rates between 2007 and this study

An important contribution is the explicit measurement of the incidence of out-of-order fragmented files (Tables 3.1, 3.2, 3.4 and 3.6), especially given that this is a large (percentage-wise) subset of all fragmented files. Additionally, the reporting on encountered actual gap sizes (Sec. 3.2.5) allows for practical estimations of the performance impact on deploying such an extended file carver. Given the amount of data fragmented out-of-order, as reported in this chapter, the impact of a file carver able to reconstruct such files can now be properly ascertained.

### 3.3.4 Carving of NTFS-compressed and sparse files

NTFS allows special storage modes that do not store the actual file contents as-is on disk: NTFS-compression and sparse files. For both types, the blocks as stored on disk are not sufficient to reconstitute a file. Note that either mode may be used irrespective of a file's contents or file type. Thus, these NTFS storage modes could pose a challenge for file carvers.

Yoo et al. [YPL$^+$12] state that most file carvers are unable to handle NTFS-compressed data (irrespective of fragmentation). They consider files of at least one block. In our dataset, only 0.8% of all files with allocated blocks (597,255 / 70,320,268) is NTFS-compressed. Yoo et al. propose a file carver to recover NTFS-compressed files. Their carver does not account for fragmented NTFS-compressed files, which (in our dataset) constitutes 12.1% of all NTFS-compressed files with blocks. Interestingly, their carver is targeted at NTFS-compressed `avi`, `wav` and `mp3` files. In our dataset, the percentage of these files that are NTFS-compressed is 0.1%, 0.0% and 0.1%, respectively (Table 3.10).

With respect to sparse files, we find only three extensions (of those investigated) have a significant portion of them as sparse: `pst` (12.8%), `sqlite` (9.6%), and `db` (7.7%). All of these are significantly more fragmented than the average: 35.8%, 27.8% and 17.4%, respectively. The percentage of sparse files for the other studied extensions remains below 0.5%.

## 3.4 Conclusions

We performed a contemporary study into file fragmentation. Our dataset [dataset-vdMee19] is comprised of disk information from 220 personally acquired, owned, and managed machines. The data was collected in a period of 4 months (Oct'18 – Jan'19).

Previous reports lacked a clear definition on which files were considered. We remedied this by distinguishing four possible definitions of fragmentation rates, from including all MFT entries to only including MFT entries that could possibly fragment. We focused our reporting on the latter definition: files that could possibly fragment. We found an average fragmentation rate of 4.4%, which presents a significant decrease compared to Garfinkel's 2007 study. This decrease is also evident on the level of individual file types. AIn particular, 25% of files fragmented into two parts is fragmented out-of-order, and this rate quickly increases with the number of fragments.

We reported on a number of fragmentation characteristics, including the convolutedness of fragmented files and the gapsize. To assess the convolutedness of fragmented files, we proposed two novel metrics: *degree of internal fragmentation* and *degree of out-of-orderedness*. Fragments are separated by a gap. We noted that there are three possible definitions of gapsize in case the next fragment precedes the current. Although the differences between these definitions are not very large, the *carving distance* still stood out: of the three, its measurements most strongly showed the "powers-of-two" gapsize property that forward-measured gapsizes so strongly exhibit. Lastly, 25% of files fragmented into two parts is fragmented out-of-order, and this rate quickly increases with the number of fragments.

# Chapter 4

# Reconstructing timelines: from NTFS timestamps to file histories

*This chapter is based on the paper* Reconstructing timelines: from NTFS timestamps to file histories*, by Jelle Bouma, Hugo Jonker, Vincent van der Meer, and Eddy van den Aker, which was accepted at the 16th International Workshop on Digital Forensics (WSDF 2023), and published in the proceedings of the 18th International Conference on Availability, Reliability and Security [WSDF23]. It was awarded the Best Research Paper Award of the workshop WSDF @ ARES 2023.*

**Abstract** File history facilitates the creation of a timeline of attributed events, which is crucial in digital forensics. Timestamps play an important role for determining what happened to a file. Previous studies into leveraging timestamps to determine file history focused on identification of the last operation applied to a file. In contrast, in this chapter, we determine all possible file histories given a file's current NTFS timestamps. That is, we infer all possible sequences of file system operations which culminate in the file's current NTFS timestamps. This results in a tree of timelines, with root node the current file state. Our method accounts for various forms of timestamp forgery. We provide an implementation of this method that depicts possible histories graphically.

## 4.1 Introduction

A major goal of digital forensics is to construct a timeline of events. Specifically, placing digital evidence (incriminating as well as exculpatory) correctly on possible timelines is a key goal of digital forensics. For example, suppose a hacked computer is seeding a torrent file of illegally downloaded content. It is rather relevant whether the seeding started before or after the computer was hacked. In some cases, this can be determined via the timestamps of relevant files. File timestamps log, amongst others, the effect of file operations: when a file last was written to, was last read, or was last accessed.

Operations on a file have a deterministic effect upon the file's timestamps. That is: executing an operation on a file will cause the file's timestamps to be updated in a specific, predetermined way, based on the time the operation was executed and the initial timestamps of the file. Moreover, different operations can affect timestamps differently. Therefore, assuming (see Sec. 4.8) no tampering and a monotonically increasing clock, a given set of timestamps can only result from a limited set of file operations. In addition, each of these possible operations imposes certain requirements on the values of the timestamps prior to its execution. As such, it is possible to determine the set of possible operations that were last applied to a file, i.e., the operations that could have led to that set of timestamps. By applying this recursively, we can reconstruct all histories of a file possible for a given set of operations. Moreover, some timestamp forgery approaches have detectable, distinctive effects and can, thus, be included in this history. We consider our method sufficiently mature to be used by practitioners.

This chapter focuses on RQ3, which is stated as: *To what extent can file history be recovered from file timestamps?*

**Contributions.** The main contribution of this research is a method to determine all possible histories of a file, given a set of operations. This method is based on:

1. determining, for each operation, its effect on timestamps,
2. inverting these effects and determining under which constraints this inverse may be applied,
3. reasoning back from the file's current timestamps by matching these inverse effects.

Secondly, we develop a proof-of-concept toolchain implementing this method for the NTFS file system. This allows us to visualise the histories (sequences of operations) possible given the file's current timestamps. Our toolchain, which includes timestamp effects of an initial list of file operations, can support practitioners in executing the proposed method.

**Availability.** We provide two proof-of-concept tools that together implement the presented method. Both tools are publicly available on GitHub. The tool TimestampAnalyzer [artefact-Bou21] implements the history-inference method; the tool TimestampVisualizer [artefact-vdAke21] parses TimestampAnalyzer's output and presents it graphically.

## 4.2 Background

**New Technology File System (NTFS).** The NTFS file system has been the default on Windows systems since Windows 2000. It stores information about files in a Master File Table (MFT). An MFT-entry contains metadata as well as disk allocation data for that file. Entries have multiple attributes for storing metadata; timestamps are stored as part of an entry's `$STANDARD_INFORMATION` (SI) and `$FILE_NAME` attributes (FN) [web-MS09].

**Timestamps in NTFS.** A timestamp denotes when a certain event has taken place. The NTFS file system stores eight timestamps per file in the file's entry in the MFT. Operations on files cause changes to anywhere between zero and eight timestamps. We therefore treat these eight timestamps as separate data points. Four timestamps are stored in a file's SI attribute, and another four are stored in the file's FN attribute. Timestamps are stored in units of 100 nanoseconds ($10^{-7}$ seconds) since 1601-01-01 00:00:00 UTC. As such, timestamp values are not affected by local time zone or daylight saving time.

Timestamps are also stored in a file called $Logfile, which is stored in the root of any NTFS volume. Cho [Cho13] has not observed any differences between the values of the timestamps stored in the $Logfile from those in the MFT.

## 4.3 Related work

The importance of timestamps in digital forensics has long been established in literature. For example, Buchholoz and Spafford [BS04] address importance of file system metadata (including timestamps), and discuss considerations and limitations that arise when trying to answer when and where a file came from, and who was responsible for the actions that generated the observed metadata.

**Effect of file operations on timestamps.** Timestamps change when file operations are executed. This can be leveraged for forensic purposes. In 2007, Chow et al. [CKL$^+$07] were one of the first studies to describe the effects of file operations on timestamps. They did this for both files and folders, using 3 SI timestamps from NTFS. The authors introduce timestamp rules to describe the effect of a file operation, warning to not apply these rules without considering timestamp-forgery. In their 2009 study, Bang et al. [BYK$^+$09] expanded upon this work by adding the FN timestamps for NTFS and the FAT file system, and their expected values for a set of file operations and FAT timestamps. With these timestamp rules, the authors demonstrate how to identify manipulation of timestamps. In 2011, Bang et al. [BYL11] expanded upon their previous study by analyzing file operations and their effects on timestamps for Windows from 2000 up to Windows 7. In addition, the authors observe different timestamp effects when modifying a file via Notepad vs. via MS Word. This suggests application-specific timestamp behaviour. By using a superincreasing sequence function, Cho [Cho16b] presents a method to identify file operations that were performed on a file. This method however does assume that the previous timestamps are known, in order to identify the correct file operation. Ho et al. [HKW18] also investigated file operations and their affect on timestamps, and expand that to cloud access behavioral patterns. They present observation rules for 7 file operations applied to a public cloud environment (OneDrive) and a private cloud environment (Hyper-V Management). In a 2021 study, Oh et al. [OLH21] created a tool called NTFS Data Tracker to identify and group all information related to a single file. This tool uses data from both the $MFT and $LogFile, and uses a Simulation of MFT Transaction technique. With it, they demonstrated they can reconstruct a file's history based on the available information in the $LogFile.

**Timestamp manipulation detection.** Timestamps can be manipulated. Various works have investigated detection of manipulated timestamps. Ding and Zou [DZ11] proposed a cross-reference time-based approach to detect timestamp-manipulation. With the Windows Registry as their cross-reference source for a given set timestamps and derived timestamp rules, they demonstrate how certain timestamp manipulations can be detected. Cho [Cho13] showed in 2013 how $LogFile can be used as a source for timestamp validation. Based upon the timestamp effects for seven file operations, he derives new rules that the timestamps must adhere to. When the timestamp rules are violated, the $LogFile can give conclusive evidence when the forgery occurred. In a subsequent study, Cho [Cho14] described timestamp changing patterns based on file operations. He presented ten distinguishable patterns which can be conclusively attributed to specific file operations. To the best of our knowledge, this is the first work that aims to identify the last operation that was performed on a file. Jang et al. [JAH$^{+}$16] presented a methodology to identify timestamp manipulation. Much like earlier work, they derive effects of operations on timestamps and build upon this. This resulted in equations of relations between $MFT and $LogFile, violation of which indicates timestamp manipulation. Palmbach and Breitinger [PB20] consider alternative sources of information for detecting timestamp manipulation. These include prefetch files, $USN journal (an NTFS log file), link files, and Windows event logs. None of the tested artifacts individually constitutes a reliable source of information, as each of them can be manipulated. However, the more sources of information included in a forensic analysis, the harder it is to manipulate all artifacts consistently.

**Other impacts on timestamps.** Other studies have considered timestamp effect beyond those of regular operations and timestamp manipulations. Schatz et al. [SMC06] report on system clock behaviour, one of the factors that influence timestamp reliability. They characterize the behaviour of drifting clocks and describe ways to correlate timestamps to events by using other, more reliable source(s). Willassen [Wil08] expanded upon their work, presenting a methodology for determining whether or not the system clock was altered using causality of timestamps. Galhuber and Luh [GL21] showed that, in addition to regular file operation and their effects on timestamp, many applications have specific timestamp effects not matching regular file operations. This gives rise to a potentially enormous set of operations with unique timestamp effects. While it may seem impossible to detect timestamp forgery in the face of this multitude of options, it turns out that most timestamp tools have telltale limitations (e.g., setting timestamps to full seconds), which allows their effects on timestamps to be distinguished from other operations. Lastly, Nordvid and Axelsson [NA22] examined whether different operating systems treated file systems equally. For the exFAT file system, they experimented with Windows, MacOS and Linux, and conclude that not all file system drivers implement the specification equivalently. Small differences exists in how each operating system stores exFAT timestamps.

## 4.4 Methodology: reasoning backwards

As previously stated, the goal is to arrive at the set of possible histories of a file. More specifically, each element in this set is an ordered list of operations that may have been applied to the file, with the last operation in the list culminating in the file's current state. Note that some sequences of operations are not possible. For example, creation must be the first operation; it cannot be preceded by other operations. To arrive at such a set of lists, we first determine how to denote a file's current state and make a selection of file operations to consider.

Next, for each file operation under consideration, we determine what effect it has on file state. Note that an operation can impose very specific effects on file state. This implies that the state of a file whose metadata lacks this telltale signature cannot be the direct result of that specific operation. For example, suppose an operation initialises all timestamps to the same value. A file whose timestamps are all equal may then be the direct result of this operation; however, a file whose timestamps vary, cannot.

Next, we determine for each operation its effect on file state. Then, we determine what the previous state of a file was, if the current state is the result of the operation under consideration. That is, we determine the effect's inverse. As mentioned, some operations cannot possibly have resulted in a given state. Such operations must thus be excluded for this transition in a file's history.

Given the inverse state transition for each operation, and the requirements that state transition imposes on the resulting timestamp, we can then reason backwards to reconstruct the set of all possible lists of operations applied to the file, given a specific list of file operations and a specific state of a file.

## 4.5 Effect of operations on timestamps

In this section, we discuss the effect of operations on NTFS timestamps. To that end, we first establish notation of file state and a list of file operations to consider. File operations have different effects depending on "modifiers" – system settings under which (some) operations have a (slightly) different effect on timestamps. Last, we measure the impact of these modifiers separately.

**Selecting file operations.** There is no consensus in literature on a canonical list of file operations to include for timestamp research. We align with previous studies by including the basic file operations (*create*, *access*, *delete*), operations for altering file metadata (*rename* and *attribute change*), and basic file system operations affecting two MFT entries (variants of *move* and *copy*). This leads to the following list:

- **Create:** Create new MFT entry

- **Access:** Access file contents
- **Update:** Modify file contents
- **Delete:** Mark MFT entry for deletion
- **Rename:** Change file name in MFT
- **Attribute change:** Change attribute(s) in MFT
- **Copy:** Create new MFT entry from source MFT entry, copy on-disk data
- **Overwriting copy:** Copy, destination is an existing MFT entry
- **Move within volume:** Updates the file path within the MFT entry
- **Move from another volume:** Create new MFT entry on destination volume, marks MFT entry for deletion on source volume
- **Overwriting move from other volume:** Destination is an existing MFT entry

### 4.5.1 Modifiers on file operations

File operation modifiers describe (system) environment changes that affect the workings of existing file operations. For example, 'last access updating' is a Windows system setting that can be active or not. Among others, the file operations 'copy' and 'update' are slightly different depending on this setting. In this research, we account for the following four file operation modifiers.

**File tunneling.** To prevent data loss during saving (e.g., in case of write errors), OS designers invented 'safe save'. 'Safe save' does not save changes to the original file, but to a new file instead. Once writing this file is finished successfully, the new file then replaces the original file by removing the original and renaming the new file. Normally, this file would appear to the user as a newly created file. To prevent that, Windows introduced file tunneling. Due to file tunneling, a create or rename operation is treated differently if the OS determines that file tunneling applies. More specifically, if a file is deleted or renamed, some of its metadata (incl. certain timestamps, long file name) is cached for a short time (default: 15 sec). If within this time frame a new file is created with the original name/path, or an existing file is renamed to the original name/path, that other file acquires the cached metadata. Note that, while file tunneling is intended to offer functionality for 'safe save', its implementation does not check for intent. That is, file tunneling occurs when its preconditions are met, irrespective of whether a 'safe save' operation was intended.

**Last access updating.** The last access timestamp value indicates the most recent event when the file was accessed. This feature is controlled by a registry setting, and was by default disabled for Windows Vista, 7, 8 and early builds of Windows 10. Since Windows 10 build 1607 (August 2016), last access updating is by default enabled for volumes smaller than 128 GB. The actual update of the last access timestamp itself is typically first cached in memory, not immediately written to disk. Writing the new timestamp to disk can be postponed by up to an hour [web-MS22].

**Transfer from FAT/exFAT.** Occasionally, files are copied from a non-NTFS file system onto an NTFS file system. As the originating file system may differ in which timestamps are tracked, and to what resolution, the NTFS copy's timestamps may bear recognizable traces of the file's origin. In this chapter, we investigate the effects of transferring files from FAT and exFAT file systems, two file systems widely used for portable storage media (memory cards, USB keys, portable hard disks).

The modified effects from file-operations originating from FAT or exFAT primarily come from limitations that these file systems have compared to NTFS.

FAT only tracks three timestamps per file ($SI_1$, $SI_2$ and $SI_4$). The $SI_1$ timestamp is stored in 10 milliseconds (or 0.01 seconds), and the $SI_2$ timestamp is stored with a resolution of two seconds [web-MS22]. The $SI_4$ timestamp is different: it is stored with an accuracy of one day for FAT. However, none of the operations under consideration transfer the $SI_4$ timestamp from FAT to NTFS systems. FAT also does not store any timezone information, whereas NTFS stores timestamps in UTC, making them 'resistant' to time zone changes or daylight savings time. exFAT (Extended File Allocation Table) solves some of the limitations FAT has. From a timestamp perspective, the same set of timestamps is available, i.e. $SI_1$, $SI_2$ and $SI_4$. The accuracy of both the $SI_1$ and $SI_2$ timestamps is now 10 milliseconds. $SI_4$ has a resolution of 2 seconds, but again, no operation under consideration transfers it to NTFS.

**Directories.** We follow Bang et al.'s approach [BYK$^+$09] to measure timestamp effect of operations on directories separately from that on regular files. This makes sense, as directories are MFT entries like regular files, except that some operations (overwriting copy, overwriting move from another volume) cannot be applied to them.

### 4.5.2 Measuring the effect of operations

For each combination of the considered file operations and file operation modifiers, we determine the effect on NTFS timestamps. As mentioned previously, the official documentation and existing literature is insufficient to determine these effects. Therefore, we perform experiments to measure the impact of each operation on timestamps. The process of performing the experiments consists of four steps:

1. Set up the environment with the volumes and files to be tested.
2. Read all the timestamps of the testfiles.
3. Perform the file operation on all the testfiles.
4. Read all the timestamps again and determine all the differences.

We perform tests for all Windows versions from Windows XP to Windows 11, for the following setup:

- Windows settings with last access updating enabled or disabled
- Files with filesizes ranging from 1 kb to 10 GB.
- Files with different extensions (i.e. .jpg, .exe, .txt)

**Experimental setup.** Since NTFS file timestamps are stored in the Master File Table, the effects of a file-operation can be observed in the MFT. We used RawCopy [web-Sch22a] to make an image of the test-volume before and after the file operations are performed, and used Mft2Csv [web-Sch22b] to extract the timestamps in a human-readable form from the MFT.

### 4.5.3 Experiment results

**Notation.** In this work, we consider file state as determined by the timestamps of an NTFS file. We denote the timestamps as $SI = (SI_1, SI_2, SI_3, SI_4)$ for the SI timestamps and the FN timestamps as $FN = (FN_1, FN_2, FN_3, FN_4)$. These are abstractions of the created, modified, record changed, and accessed timestamps. When considering the state transition resulting from a specific file operation, timestamps before the operation are denoted as $SI$ and $FN$; timestamps

| Operation | $SI', FN'$ | | | |
|---|---|---|---|---|
| Create | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Access | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Update | $(SI_1,$ | $op_{end},$ | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Delete | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Rename | $(SI_1,$ | $SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| Attribute change | $(SI_1,$ | $SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Copy | $(op_{start},$ | $src.SI_2,$ | $op_{end},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting copy | $(SI_1,$ | $src.SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Move within volume | $(SI_1,$ | $SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| Move from other volume | $(src.SI_1,$ | $src.SI_2,$ | $op_{end},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting move from other volume | $(src.SI_1,$ | $src.SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |

Table 4.1: Timestamp effect of file operations (no modifiers)

post-operation are denoted as $SI'$ and $FN'$. Finally, we denote the time when an operation starts as $op_{start}$ and the time when it ends as $op_{end}$. Treating timestamps symbolically abstracts away from irrelevant details such as the speed of the storage device being used, and instead highlights the effect of the specific operation under investigation.

We denote the effects of file operations under a modifier by showing the same table (with measurements for the non-modifier case) in gray, with any changes in regular black. Operations whose behaviour remains unchanged under a modifier are omitted from the table.

Table 4.1 shows the effects of the file operations on timestamps. These values were established without any modifiers enabled.

For example, for *copy* we see that the target file's timestamps are affected: all *FN* values are set to the starttime of the operation ($op_{start}$), the $SI_2$ timestamp is set to the source file's $SI_2$ timestamp ($src.SI_2$), $SI_1$ and $SI_4$ are set to operation starttime ($op_{start}$) and finally $SI_3$ is set to operation endtime ($op_{end}$). We found no difference in the effect of file-operations for the different versions of Windows, file size or file extension.

**File tunneling.** Table 4.2 lists the operations whose behaviour is affected by file tunneling, and their behaviour under this operation. In short, in some cases, $SI_1$ and $FN_1$ take their value from the original (now removed) file's $SI_1$ and $FN_1$ timestamps, respectively.

| Operation | $SI', FN'$ | | | |
|---|---|---|---|---|
| Create | $(del.SI_1,$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| | $(del.FN_1,$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Rename | $(del.SI_1,$ | $SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| Copy | $(del.SI_1,$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| | $(del.FN_1,$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Move within volume | $(del.SI_1,$ | $SI_2,$ | $op_{start},$ | $SI_4)$ |
| | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |

*del*: the file with the same exact path and name as this file, that was deleted prior to this operation (within the File Tunneling time window).

Table 4.2: Modifiers: File Tunneling ("safe save")

**Last access updating.** Effect on timestamps of file operations when last access updating is active is presented in Table 4.3. If the system crashes before last access updating is effectuated, the timestamps will not be updated beyond what was presented in Section 4.5.3. From the table, it is clear that enabling last access updating only affects timestamp $SI_4$.

**Transfer from FAT / exFAT.** The results for file operations regarding incoming files from FAT and exFAT are presented in Tables 4.4 and 4.5 respectively.

**Directories.** We measured the effect of the considered file operations upon directories. The results of this are presented in Table 4.6; the only change from Table 4.1 is for the *update* operation, w.r.t. $SI_4$.

| Operation | $SI', FN'$ | | | |
|---|---|---|---|---|
| Access | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $op_{start})$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Update | $(SI_1,$ | $op_{end},$ | $op_{start},$ | $op_{end})$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Copy | $(op_{start},$ | $src.SI_2,$ | $op_{end},$ | $op_{end})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting copy | $(SI_1,$ | $src.SI_2,$ | $op_{start},$ | $op_{start})$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Move from another volume | $(src.SI_1,$ | $src.SI_2,$ | $op_{end},$ | $op_{end})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting move from other volume | $(src.SI_1,$ | $src.SI_2,$ | $op_{start},$ | $op_{start})$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |

Table 4.3: Modifiers: Last access updating

### 4.5.4 Effects of timestamp forgery

Lastly, we illustrate the potential of our file histories method to detect timestamp forgery. To that end, we determine the effects of three timestamp forgery approaches: two basic Windows system calls and a popular timestamp forgery tool. The effects of each of these are presented in Table 4.7.

**Approach 1: SetFileTime.** Timestamps can be altered by the Windows system call *SetFileTime*. This system call allows to set the $SI_1$, $SI_2$, and $SI_4$ timestamps in whole seconds. Timestamp changing tools using this call are thus limited to changing only these three timestamps with conspicuous values. While in theory it is possible for these three timestamps to all three be exact whole seconds for most operations, in practice, this is a strong indication of tampering.

**Approach 2: NtSetInformationFile.** Another way of manipulating timestamps is through the undocumented *NtSetInformationFile* Windows system call. This system call can set the $SI_1, SI_2$ and $SI_4$ timestamps to any user specified values with full precision.

**Approach 3: Timestomp.** Lastly, we consider Timestomp, a timestamp manipulation tool. Timestomp uses the *NtSetInformationFile* system call [Cho16a]. Nevertheless, like other timestamp manipulation tools, it limits accuracy of altered timestamps to full seconds. These stand out from NTFS's default resolution of 100

| Operation | $SI'$, $FN'$ | | | |
|---|---|---|---|---|
| Copy | $(op_{start},$ | `valB`, | $op_{end},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting copy | $(SI_1,$ | `valB`, | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Move from FAT volume | (`valA`, | `valB`, | $op_{start},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting move from FAT volume | (`valA`, | `valB`, | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |

`valA`: $src.SI_1$ rounded up to centiseconds (0.01 second) plus time zone difference ($tzd$);
`valB`: $src.SI_2$ rounded up to even seconds plus $tzd$.

Table 4.4: Modifiers: Transfer from FAT

| Operation | $SI'$, $FN'$ | | | |
|---|---|---|---|---|
| Copy | $(op_{start},$ | `valB`, | $op_{end},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting copy | $(SI_1,$ | `valB`, | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Move from exFAT volume | (`valA`, | `valB`, | $op_{end},$ | $op_{start})$ |
| | $(op_{start},$ | $op_{start},$ | $op_{start},$ | $op_{start})$ |
| Overwriting move from exFAT volume | (`valA`, | `valB`, | $op_{start},$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |

`valA`: $src.SI_1$ rounded up to centiseconds (0.01 second);
`valB`: $src.SI_2$ rounded up to centiseconds

Table 4.5: Modifiers: Transfer from exFAT

nanoseconds. This is a strong indication of tampering, though most operations could theoretically result in such values.

## 4.6 Deducing possible file histories from NTFS timestamps

With the effects of file operations on the $SI$ and $FN$ timestamps, we have laid the groundwork for reconstructing possible timelines. The concept will be introduced with a simplified running example. First, we can describe the state of a file along with its eight timestamps. When a file undergoes multiple file operations, its state and its timestamps change accordingly. For example, in Figure 4.1 a file

| Operation | $SI'$, $FN'$ | | | |
|---|---|---|---|---|
| Update | $(SI_1,$<br>$(FN_1,$ | $op_{end},$<br>$FN_2,$ | $op_{start},$<br>$FN_3,$ | $op_{end})$<br>$FN_4)$ |
| Overwriting copy | operation not applicable to directories | | | |
| Overwriting move from another volume | operation not applicable to directories | | | |

Table 4.6: Directories

| Method | $SI'$, $FN$ | | | |
|---|---|---|---|---|
| `SetFileTime()` | (`valA`,<br>$(FN_1,$ | `valB`,<br>$FN_2,$ | $op_{start},$<br>$FN_3,$ | `valD`)<br>$FN_4)$ |
| `NtSetInformationFile()` | $(any_1,$<br>$(FN_1,$ | $any_2,$<br>$FN_2,$ | $any_3,$<br>$FN_3,$ | $any_4)$<br>$FN_4)$ |
| Timestomp | (`valA`,<br>$(FN_1,$ | `valB`,<br>$FN_2,$ | `valC`,<br>$FN_3,$ | `valD`)<br>$FN_4)$ |

| | |
|---|---|
| `valA`, `valB`, `valC`, `valD`: | any value, rounded to whole seconds. |
| $any_i$: | any value, up to full precision. |

Table 4.7: Effect of API / tool timestamp manipulation

is first created, then updated, and, lastly, renamed. After each state transition, its timestamps are updated (changes in bold italic red) according the timestamp rules described in the previous section.

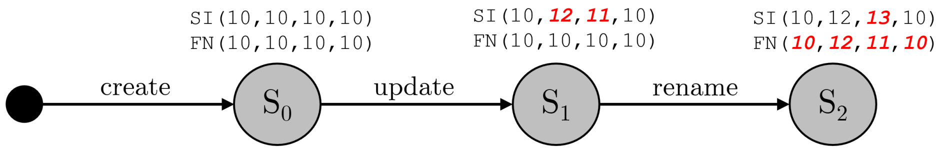

Figure 4.1: Example: forwards evolution of timestamps under file operations

We see that the *update* operation changes $SI_2$ and $SI_3$, leaving all other timestamps unchanged. Similarly, *rename* overwrites all $FN$ timestamps with copies of the original's $SI$ counterparts and changes $SI_3$, leaving the other three timestamps unchanged.

With the forwards evolution of timestamp established, we can now apply this reasoning backwards to determine previous allowed states of the file. Assume that you find a file in a certain state with `SI(10, 12, 13, 10)` and `FN(10, 12, 11, 10)`. In Figure 4.1, this state was the result of the rename operation. If we now apply the inverse of the effect of the rename operation on the timestamp, we arrive at a state where we do not know any value of the *FN* timestamps, as they were overwritten. We do know what the *SI* timestamp should be in that previous state – namely, precisely the values of the *FN* timestamp of the current state. This is depicted graphically in Figure 4.2.

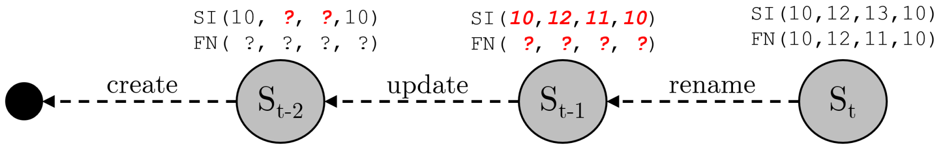

Figure 4.2: Example: backwards reasoning and information loss

The inverse effects of file operations on timestamps are described in Table 4.8. Some timestamps are overwritten by the operation; for these, no information of its previous state is available (denoted as '?' in the table). This table allows us to consider not one, but all operations at each turn. For each file state under consideration, we can try to apply the inverse of each file operations. Not all inverses will be possible, for example, for state $S_t$ in Figure 4.2, only rename and copy (source) are possible. No other file operations could result in that state's timestamps. In general, most file operations can only result in certain values for timestamps. For example, the timestamps following a create operation are all equal. Therefore, any file state whose timestamps are not all equal, cannot be the result of a create operation. In other words, this imposes a constraint on (relations between) values for timestamps which can follow from a create operation.

Note that in Table 4.8, unlike in Table 4.1, we must distinguish between source and target for overwriting copies/moves. This is because for overwriting copy and overwriting move, two files must have existed previously: source and target. For all other operations, there is only one "ancestor" file.

For a state to result from a specific file operation, certain conditions have to be met, depending on the operation. These conditions are specified in Table 4.9. In addition to operation-specific constraints, there are constraints related to operation start/end time. We refer to these collectively as 'OPTIMING'. The first type of OPTIMING constraint is that operations cannot end before they start. Thus, all timestamps set to $op_{start}$ must have values smaller than or equal to any timestamp set to $op_{end}$. Second, under a monotonically increasing system clock, operation start/end time must always be later than timestamps copied from the previous state. The last OPTIMING constraint is that if more than one timestamp is set to $op_{start}$, all such timestamps must be equal. This holds

| Operation | $SI^{t-1}$, $FN^{t-1}$ | | | |
|---|---|---|---|---|
| Create | N/A | | | |
| | N/A | | | |
| Access | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Update | $(SI_1,$ | ?, | ?, | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Delete | $(SI_1,$ | $SI_2,$ | $SI_3,$ | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Rename | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| | (?, | ?, | ?, | ?) |
| Attribute change | $(SI_1,$ | $SI_2,$ | ?, | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| Copy (source) | (?, | $SI_2,$ | ?, | ?) |
| | (?, | ?, | ?, | ?) |
| Overwriting copy | | | | |
| – target | $(SI_1,$ | ?, | ?, | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| – source | (?, | $SI_2,$ | ?, | ?) |
| | (?, | ?, | ?, | ?) |
| Move within volume | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| | (?, | ?, | ?, | ?) |
| Move from another volume (source) | $(SI_1,$ | $SI_2,$ | ?, | ?) |
| | (?, | ?, | ?, | ?) |
| Overwriting move from another NTFS volume | | | | |
| – target | (?, | ?, | ?, | $SI_4)$ |
| | $(FN_1,$ | $FN_2,$ | $FN_3,$ | $FN_4)$ |
| – source | $(SI_1,$ | $SI_2,$ | ?, | ?) |
| | (?, | ?, | ?, | ?) |

Table 4.8: Previous state for file operations for
$SI^t = (SI_1, SI_2, SI_3, SI_4)$, $FN^t = (FN_1, FN_2, FN_3, FN_4)$

similarly for $op_{end}$.

For brevity, we abstract away from stating all such constraints explicitly and denote these as 'OPTIMING' in Table 4.9. The $op_{start}$ and $op_{end}$ columns indicate which timestamps are set to operation starttime and which for operation endtime, respectively. Lastly, note that, for both overwriting operations, the constraints apply equally to both source and target, and there is no further information on which to base any further constraints for either.

Using these constraints, we can extend the example of Figure 4.2. A more extended, but still not complete, example of backwards reasoning is shown in Figure 4.3. We see branching, when a state could have been the result from more than one operation. We also see that information loss occurs, e.g., when timestamps are overwritten as by the rename operation. Lastly, we see that

| Operation | Constraint | $op_{start}$ | $op_{end}$ |
|---|---|---|---|
| Create | $SI_i = FN_j$, for $i, j \in \{1, \ldots, 4\}$ | $SI_{1\ldots4}$, $FN_{1\ldots4}$ | |
| Access | True | | |
| Update | OPTIMING | $SI_3$ | $SI_2$ |
| Delete | True | | |
| Rename | OPTIMING $\wedge$ $FN_i = SI_i, i \in \{1, 2, 4\}$ | $SI_3$ | |
| Attribute change | OPTIMING | $SI_3$ | |
| Copy | OPTIMING | $SI_{1,4}$, $FN_{1\ldots4}$ | $SI_3$ |
| Overwriting copy | OPTIMING | $SI_3$ | |
| Move within volume | OPTIMING $\wedge$ $FN_i = SI_i, i \in \{1, 2, 4\}$ | | $SI_3$ |
| Move from another volume | OPTIMING | $SI_4$, $FN_{1\ldots4}$ | $SI_3$ |
| Overwriting move from other NTFS volume | OPTIMING | $SI_3$ | |

Table 4.9: Constraints operations impose on timestamps

applying the backwards reasoning process recursively, we can end up in a state where no information on timestamp values is known anymore ($S'_{t-2}$ in the figure). There might still be more file history preceding this state, but nothing is known about this.

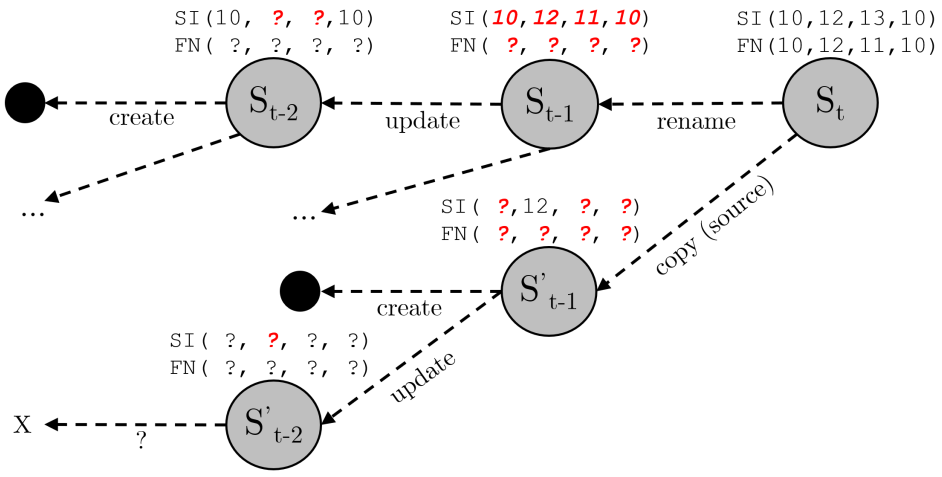

Figure 4.3: Example: backwards reasoning with branching and different branch-lengths

## 4.7 Proof of concept implementation

We implemented and realized the presented methodology of reasoning backwards in two proof-of-concept tools: a timestamp analyzer and a visualisation tool.

### 4.7.1 Timestamp analyzer

This tool is configured with a list of file operations, where, for each operation, its effect on timestamps must be specified. We supply the list of operations and effects as discussed in the previous section. Operations that do not affect any timestamp are removed from consideration, as these can occur infinitely often at all points in a timeline. For our previously established list, this concerns first, the *access* operation when last-access-updating is not active, and second, the *delete* operation. Below, we describe three aspects (one concept and two algorithms) of the implementation.

**Tracking information loss.** To track loss of information in progressive steps of reasoning backwards, we use the concept of markings. A timestamp is marked when its value was changed due to the preceding file operation. This implies a loss of information: the value of a marked timestamp prior to the file operation that caused its marking is unknown and may not be used.

File operations can also *un*mark a timestamp. Unmarking is the reverse of marking: it denotes a gain of information, which occurs when a timestamp's new value originates from a known source. For example: the *rename* operation sets $SI'_3$ to $op_{start}$(which is thus marked), but also sets $FN'_1$ to $SI_1$, $FN'_2$ to $SI_2$, etc. Thus, we can still derive each of $SI_{1...4}$ from $FN'_{1...4}$.

**Match-operation algorithm.** The matching algorithm is responsible for identifying file operations compatible with the current state. As input, it is given the timestamps, markings, and a candidate file operation. If no constraint is violated, then the given state could have resulted from this operation and the algorithm returns `True`. In addition to the criteria presented in Table 4.9, the algorithm also takes constraints following from modifiers (Tables 4.3–4.6). This includes time-resolution constraints following from modifiers 'transfer from FAT/exFAT' (Tables 4.4, 4.5).

**Timelines construction algorithm.** The Timelines construction algorithm recursively builds all possible timelines of a given file state. Each timeline consists of a sequence of one or more file operations matching the state of the file at that point in the sequence (timestamps, markings). A sequence terminates when it encounters a *create* operation, or when no more operations can be matched to the file state. The latter can occur either due to complete loss of information, or when no file operation can match the available information.

**Detecting timestamp forgery.** Our proof-of-concept implementation considers timestamp forgery for timestamps. Our tooling adds a forgery branch only when:

1. not all information has been lost yet,
2. no regular operation can have caused the timestamps under consideration,
3. the timestamps match the requirements induced by Table 4.7.

**Implementation limitations.** The proof-of-concept has only been tested on a limited number of MFTs and has not been optimised; quality attributes such as scalability and performance were not part of our testing process.

### 4.7.2 Visualisation tool implementation

This tool creates a visualisation of all possible timelines based on the output of the analysis tool. To illustrate the use of this visualisation tool, we show in Figure 4.4 a visualisation of the analysis of a file that underwent comparable file operations as our running example. The only difference is that these are real timestamps, instead of simple numbers.

The visualisation is oriented like a timeline, with time increasing to the right. States further left are older, with the current (known) state depicted rightmost. The time at which an operation has taken place is noted in the black titlebar. When multiple operations lead to an identical state, they are grouped together for readability purposes. A file state that does not contain sufficient information to match any file operation is preceded by a question mark instead of another file state. This denotes that there is insufficient information to go back further in time.

## 4.8 Discussion

**System clock monotonicity.** Our method relies on the key assumption that the system clock is monotonically increasing. This cannot be correct in all

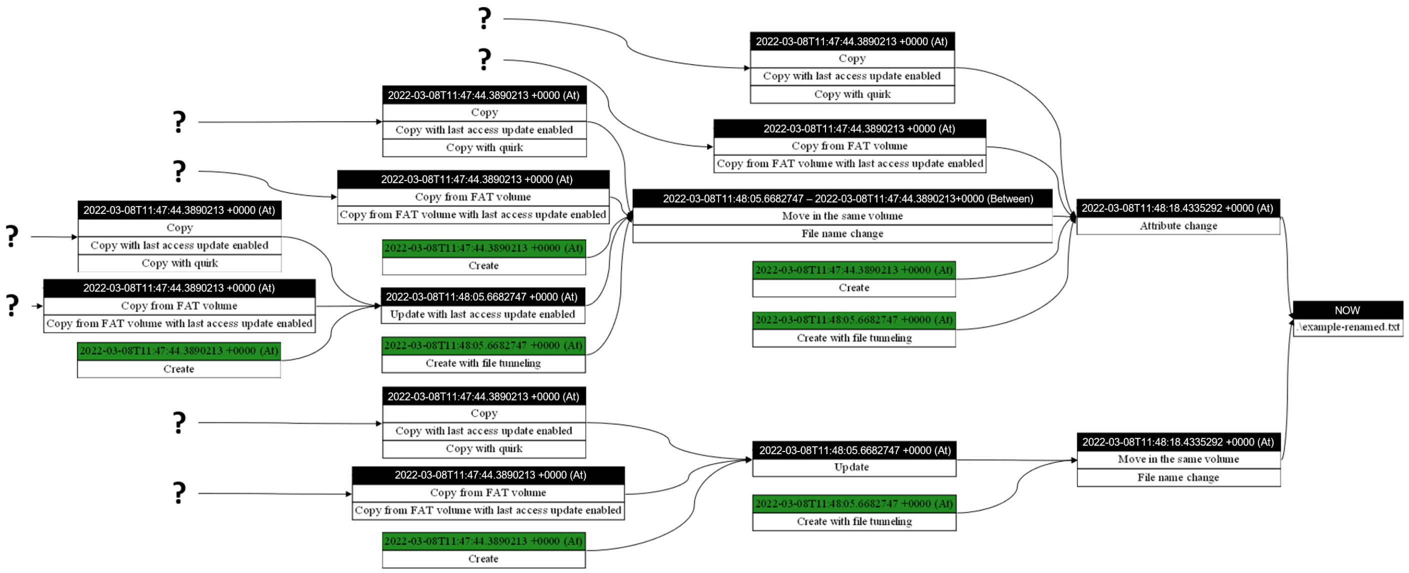
?
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy
Copy with last access update enabled
Copy with quirk
2022-03-08T11:47:44.3890213 +0000 (At)
Copy
Copy with last access update enabled
Copy with quirk
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy from FAT volume
Copy from FAT volume with last access update enabled
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy from FAT volume
Copy from FAT volume with last access update enabled
2022-03-08T11:48:05.6682747 – 2022-03-08T11:47:44.3890213+0000 (Between)
Move in the same volume
File name change
2022-03-08T11:48:18.4335292 +0000 (At)
Attribute change
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy
Copy with last access update enabled
Copy with quirk
2022-03-08T11:47:44.3890213 +0000 (At)
Create
2022-03-08T11:47:44.3890213 +0000 (At)
Create
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy from FAT volume
Copy from FAT volume with last access update enabled
2022-03-08T11:48:05.6682747 +0000 (At)
Update with last access update enabled
2022-03-08T11:48:05.6682747 +0000 (At)
Create with file tunneling
NOW
.\example-renamed.txt
2022-03-08T11:47:44.3890213 +0000 (At)
Create
2022-03-08T11:48:05.6682747 +0000 (At)
Create with file tunneling
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy
Copy with last access update enabled
Copy with quirk
?
2022-03-08T11:47:44.3890213 +0000 (At)
Copy from FAT volume
Copy from FAT volume with last access update enabled
2022-03-08T11:47:44.3890213 +0000 (At)
Create
2022-03-08T11:48:05.6682747 +0000 (At)
Update
2022-03-08T11:48:05.6682747 +0000 (At)
Create with file tunneling
2022-03-08T11:48:18.4335292 +0000 (At)
Move in the same volume
File name change

Figure 4.4: Full backwards reasoning running example

circumstances: switches from daylight savings time to regular time set the clock back. Similar problems may occur when system time is synchronised (e.g., via NTP) and the system clock drifts too far ahead, requiring the system clock to be wound back. Moreover, it is typically trivial to change the system clock.

**Information loss.** Our method for backwards reasoning inherently suffers from information loss. Therefore, each reconstructed timeline has a finite horizon of how far back our method can reconstruct possible file states. Possible file histories beyond a file's horizons cannot be reconstructed via this method.

**Forensic limitations.** Our approach supports reasoning about any file operation. However, our proof-of-concept analysis tool is only seeded with a list of basic file operations. Concretely, it lacks application-specific file operations. We recommend practitioners to follow our approach to determine the timestamp effects of file operations for any applications that they wish to incorporate into the reasoning framework. Expanding the list suffices to incorporate new file operations into the tooling.

## 4.9 Conclusion and future work

In this chapter we presented a novel method to reconstruct an overview of allowed histories of a file based on its timestamps. We discussed how to measure the timestamp effect of operations with and without modifiers (file tunneling and last access updating), and apply this to a set of basic file operations. The measurements reveal that operations can result in a file state where there are specific relations between some of the timestamps (Table 4.8). Our method leverages this to derive previous states: a certain file state cannot be the result of a given file operation, unless the state's timestamps all satisfy the relations required by the operation (Table 4.9). Thus, certain operations can be excluded as having caused the current state. This gives a set of possible previous file states. By recursively applying this process, our method can construct all allowed histories of a file, for the operations under consideration. This is fundamentally different from prior timestamp studies, which limit their approach to identifying only the last possible operation. In contrast, our method constructs a set of timelines of events. Lastly, we implemented our method in a proof-of-concept and showed viability of this approach to construct file histories.

The described method can be readily used by practitioners to reconstruct possible file histories. This will help them to substantiate or refute timeline-related hypotheses.

**Future work.** First, to improve practical use, the list of file operations can be expanded with application-specific file modifiers.

Second, the presented method of reconstructing possible file histories is based on the timestamp values stored in the MFT. Possible timeline reconstruction can be improved upon when sources of previous timestamp information can be included, such as log files, link files, or prefetch files [PB20]. With such additional points of information, information loss would be reduced and therefore the timeline horizon could be improved.

Last, the algorithms of the proof-of-concept tool are operation-agnostic. That is, they will accept any list of operations. Moreover, information loss ensures that our process for determining timelines always terminates for the operations from Table 4.1. It is possible that information loss of some other, not yet considered operations is insufficient to guarantee termination. That is, some operations could exist that jointly cause cyclical patterns in the reconstructed timeline. How to detect this and how to handle such occurrences are left for future work.

# Part II

# JPEG validation

Highlights:

- Design and demonstration of a novel algorithm capable of detecting fragmentation points in JPEG files, addressing a key challenge in digital forensics.
- Implementation ([artefact-vdBvdM23]) and thorough validation of the artefact.
- Extensive evaluation of the proposed methods using real-world datasets, demonstrating their effectiveness and applicability for validation and file recovery. This effectively solves a long standing problem in the field of Digital Forensics.

Chapter 5

# JPEG file fragmentation point detection

*This chapter presents an adapted version of the paper,* JPEG File Fragmentation Point Detection using Huffman Code and Quantization Array Validation*, by Vincent van der Meer and Jeroen van den Bos, which was accepted at the 14th International Workshop on Digital Forensics (WSDF 2021), and published in the proceedings of the 16th International Conference on Availability, Reliability and Security [WSDF21].*

**Abstract** File carving is a data recovery technique used in many investigations in digital forensics, with some limitations. Especially JPEG files are difficult to recover when fragmented, because they consist almost entirely of large blobs of highly compressed entropy-coded data, with no clearly discernible structure.

This chapter describes an approach that leverages two observations about many JPEG files in practice. First, the Huffman tables used to decode a large proportion of the entropy-coded data often do not use all possible code values at their longest code length, offering possibilities to detect errors when invalid codes are encountered. Second, after translating Huffman codes to symbols, the next step in decoding involves filling quantization arrays with exactly 64 values, offering another possibility to detect errors when an overflow is encountered.

This chapter presents algorithms to validate the entropy-coded data using these two observations and finds that the odds of finding fragmentation points are quite high, especially with regard to invalid Huffman codes. It will work with the example Huffman tables provided by the JPEG standard that are used by many digital cameras, but also with many optimized Huffman tables generated by specialized applications.

## 5.1 Introduction

Many investigations in digital forensics rely on data recovery. Apart from suspects hiding or removing information, accidents or acts of violence often physically damage devices making their contents no longer directly accessible. A key technique in data recovery is file carving, which is a process of recovering files that is not based on metadata (e.g., such as file system information) but on recognizing the internal structure of file types. Using this technique, many types of files can be recovered quickly and reliably from a raw copy of a device's contents.

A complication that often arises when file carving is file fragmentation: file systems commonly split files into multiple pieces for performance reasons. Basic file carvers, that look for the magic values in the headers and footers of many file types cannot recover fragmented files. To recover fragmented files, more advanced carving tools are required that reconstruct fragmented files. However, this requires a method to determine where the actual fragmentation has occurred (i.e., where the end of a fragment is located).

For this purpose a file validator is employed: a recognizer for a specific file format that, when given a potential file, responds whether it indeed qualifies as a valid instance of that file type and ideally also provides detailed information about where an error occurred if the answer is no. Whether it is complicated or even possible to construct a validator for a specific file type that can accurately determine the location of such a *fragmentation point* depends on the internal structure of that file type.

Arguably the most important file type in digital forensics is JPEG: by a significant margin, the JPEG file format is the most widely used digital storage format for photos [HLN$^+$18]. It is used used for images in any media, on digital cameras, on websites, social media and in any other type of application. In order to effectively carve JPEG files, a file validator is needed that is capable of accurately identifying the fragmentation point when reading from the start of a JPEG fragment. JPEG has a fairly simple file structure, but because almost its entire contents is not guarded by values such as length fields, checksums or other easily identifiable markers, constructing a file validator with this capability for it is non-trivial.

In this chapter we present a novel approach to validate the entropy-coded data of JPEG files based on detecting invalid Huffman codes and quantization array size overflows. Our contributions consist of both the two validation approaches as well as the described algorithms that incorporates both. The algorithms presented are capable of validating baseline as well as progressive JPEGs. In the following section we discuss the background of both file fragmentation, the file structure of JPEG and the encoding and decoding processes. Next, we discuss some observations that lead to our description of the validation algorithms, followed by the description of an example showing how the algorithm works in practice. Finally, we analyze the success rate, discuss limitations, considerations and related work.

This chapter focuses on RQ4, which is stated as: *To what extent can entropy-coded JPEG data be validated?*

## 5.2 Background

### 5.2.1 File fragmentation in practice

File fragmentation occurs in many situations: file systems use it on storage media, protocols in network streams and operating systems in how they allocate memory. Since file carving is performed most commonly on storage media such as hard drives and flash chips, we focus here on aspects specifically relevant to file systems, but the principles apply everywhere.

File systems allocate files on storage media. The smallest amount of data they allocate is called a block. For the NTFS file system commonly used on Windows systems, the default size of these blocks is 4096 bytes for volumes up to 16 TB [web-MS21]. So whenever a file is stored in a file system, it takes up at least one single block [1] but in many cases it will take up multiple blocks, up to thousands or millions for extremely large files. File systems use this technique in order to support very large storage devices with relatively modest metadata overhead.

A file is considered fragmented when its blocks are not stored both contiguously and in-order. Files thus need to have two blocks or more allocated to them to become susceptible to file fragmentation. A file fragment consist of one or more contiguous blocks that form a part of a file. This type of fragmentation occurs because of churn on the file system: when a system is in use, its operating system regularly creates, resizes and removes files. When storing and resizing a file on a volume with a lot of activity, the operating system often has no other choice but to cut it into fragments in order to store it in the available space.

Some research has been done on analyzing the fragmentation rate on storage media. In a dataset that consists of file systems gathered between 1998 and 2006, Garfinkel [Gar07] reports a 6% fragmentation rate. Meyer and Bolosky [MB12]

[1]Except for small files that can sometimes be stored in the file system's metadata.

report a 4% fragmentation rate in 2012. In Chapter 2, we reported a 2.2% fragmentation rate for all files, and a 4.4% fragmentation rate for files that can be fragmented. Despite a slowly decreasing fragmentation rate, the amount of fragmented data has increased in that time period due to exponential increases in average storage media size.

### 5.2.2 The JPEG file format

The JPEG image file format is the predominant file format for photos by a significant margin [HLN$^+$18]. Since its introduction in 1992, it quickly became the de facto standard in many domains. The JPEG standard is extensive and offers many extension points and implementation options, but in practice nearly all files encountered in practice use lossy compression using Huffman encoding and are serialized into one of two popular file formats that adhere to the JPEG Interchange Format (JIF). These two formats are the JPEG File Interchange Format (JFIF) and an extension of it that is popular among digital cameras called Exchangeable Image File Format (EXIF). The JPEG specification allows for 16 types of encoding of image data (SOF0–SOF15). In practice, two types of encoding are used: baseline JPEGs (SOF0, Fig. 5.1) and progressive JPEGs (SOF2, Fig. 5.2). Baseline JPEGs must be encodable/decodable in a single scan of the image data. Progressive JPEGs require more than one scan.

JPEG uses various transition and encoding techniques to transform pixel color values to highly compressed image data. A typical JPEG file is the result of the following transformations:

1. Color conversion from RGB to YCbCr. This transformation is lossless, as it only separates brightness from color data.

2. 2D Discrete Cosine Transform (DCT), which is also lossless.

3. Quantization of DCT-coefficients, using a quantization matrix. This transformation is lossy.

4. The quantization matrix is mapped using a zig-zag pattern, and encoded with either delta (the first coefficient) or zero run-length encoding for the remaining coefficients. Both encodings are lossless.

5. Huffmann encoding to further compress the remaining data (lossless).

A JPEG file consists of two types of data: metadata needed for decoding, and image data (also called entropy-coded data) that is actually decoded when viewing an image. The metadata specifies how the image should be decoded. JPEG files are self-contained since they contain the values of the quantization tables along with information needed to construct the Huffman tables.

**Markers.** The JPEG format uses markers for structure. Markers (also called magic values) are reserved two-byte values that may not be used for other purposes. These markers denote the the type of serialized data structure and length fields to simplify parsing. The marker values all start with byte value 0xFF followed by a byte that specifies its meaning. For example, there are markers for start/end of image, type of encoding, metadata, etc. (see Table 5.1). By definition, each JPEG file starts with a START OF IMAGE (SOI) marker and ends with an END OF IMAGE (EOI) marker. The START OF SCAN (SOS) marker indicates the start of an entropy-coded data segment, and the first such marker thus is the separator between header and content of a file. The header contains all information needed for decoding such as information on how many color channels an image has (part of START OF FRAME (SOF) marker), whether, and what type of chromatic subsampling is used (also part of the SOF marker), what values should be used for quantization (DQT marker), and compressed representations of the Huffman tables (DHT marker) that should be used.

| Symbol | Hex value | Meaning |
|---|---|---|
| SOI | `FFD8` | Start of Image |
| APP0 | `FFE0` | JFIF metadata |
| APP1 | `FFE1` | Exif metadata |
| DQT | `FFDB` | Define quantization table(s) |
| SOF0 | `FFC0` | Start of frame, baseline DCT |
| SOF2 | `FFC1` | Start of frame, progressive DCT |
| DHT | `FFC4` | Define Huffman table(s) |
| DRI | `FFDD` | Define restart interval |
| SOS | `FFDA` | Start of scan |
| RSTn | `FFDn` | Restart ($n \in \{0, 1, ..., 7\}$) |
| EOI | `FFD9` | End of image |

Table 5.1: Most important JPEG Markers

The SOI, DQT, SOF and DHT data structures constitute only a few kilobytes of a typical JPEG file of several megabytes as produced by any modern mobile phone or camera. The rest of the data is in one or more so-called *Scans* following the SOS structure and terminating right before the EOI. Within this entropy-coded data some escaped values (starting with 0xFF) are allowed, but these are rare in practice.

As a result, any JPEG file larger than a couple of kilobytes will consist almost entirely of entropy-coded data, which is highly compressed and does not contain any checksums, length fields or other hints to easily determine whether the contents has been corrupted or fragmented. A single exception to this are restart markers, which is an optional mechanism specified by the JPEG standard that allows numbered markers to be inserted into the entropy-coded data specifically to detect errors. Unfortunately, the mechanism is optional and it is rarely encountered in practice (in our experience, but also as reported by Tang et al. [TFC+16]).

**Decoding compressed image data.** At the lowest level, the JPEG format stores the values of each $8 \times 8$ block of pixels in a Minimal Coded Unit (MCU). An MCU contains either one (Y) or three (YCbCr) color channels. Each color channel is described by a two-dimensional $8 \times 8$ data structure called the quantization array (QA). The QA is stored as follows: the lengths of the quantization values are Huffman-encoded. In addition, the decoded Huffman values also encode how many (if any) zeroes must be placed in the quantization array before the actual value. The actual values are stored directly in the bitstream.

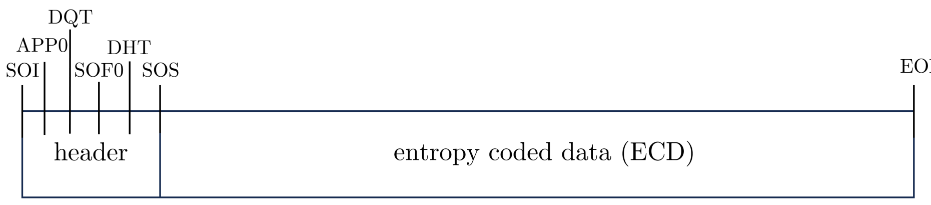

Figure 5.1: Example JPEG baseline file structure

**Progressive encoding.** Progressive JPEGs, as previously mentioned, mandate sequential scans since the image is constructed of multiple layers of image data, with each subsequent layer adding additional image detail. The occurrence of markers within the high-entropy-coded data section of a progressive JPEG differs compared to a baseline coded JPEG. Not only does each layer start with a SOS marker, each layer can be accompanied by new Huffman tables, indicated by the presence of a DHT marker.

**Support for file recovery.** In terms of file recovery, JPEG has some major shortcomings. While it does have identifiable markers, it lacks integrity checks

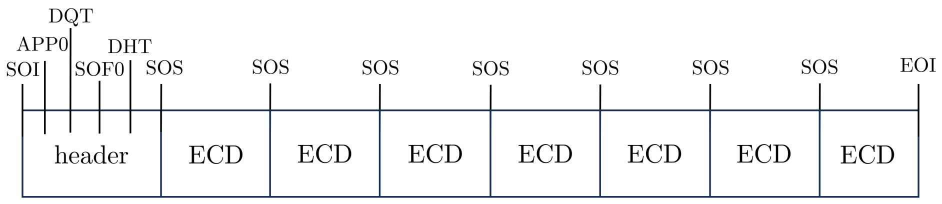

Figure 5.2: Example JPEG progressive file structure

(like CRC) or length fields that encompass the image data. Additionally, its design prioritizes data storage optimization, meaning that image data has high-entropy. By prioritizing compression over robustness, the JPEG file format's design severely complicates file recovery.

## 5.3 Fragmentation point detection for JPEG

**Fragmentation point detection.** In file recovery, sometimes a file needs to be reconstructed from fragments that consists of one or more consecutive blocks of data, found on a storage device. This is generally a difficult task, since the file system metadata with the allocated blocks for that file is missing.

In order to successfully carve fragmented JPEG files, a validator is needed that can accurately pinpoint where inside a provided candidate file the fragment ends. This will allow the file carver to remove or reshuffle the blocks around only that location in order to more quickly find a valid file. If validation concludes successfully, the file is reconstructed from the used blocks. If the validator returns an error, the reported location helps the carver determine in which block the fragmentation occurred. The validator reports the last known good location. The more precise the validator's information is, the smaller the search space that the file carver needs to consider.

If the fragmentation occurs near the start of a JPEG file, then this is fairly easy: the first data structures have clear markers, length fields and a well-defined structure. Unfortunately, this only concerns the first few blocks as the rest of a JPEG file is mostly entropy-coded, highly-compressed data. This chapter describes a method to easily and quickly validate many parts of this entropy-coded data by looking for invalid Huffman codes and possible size overflows when constructing quantization arrays.

### 5.3.1 Validation using Huffman table lookup errors

A large proportion of the entropy-coded data consists of Huffman codes, which when mapped to their symbol are either directly used or refer to a value in the data following the code. Because of this it is difficult to directly describe the percentage of data that consists of Huffman codes, since both Huffman codes and values have variable sizes. In practice, it seems reasonable to expect more than 50% of the entropy-coded data to consist of Huffman codes.

Instead of storing the Huffman tables directly, the DHT data structure describes the symbols and a list of code lengths to be generated by the decoder. Since this algorithm is fixed in order to guarantee interoperability, the resulting Huffman table is always the same and the DHT structure is just a form of compression.

These Huffman tables consist of a list of codes of variable length and their accompanying symbols. Whenever a code is encountered in the input, it is translated to its symbol for further interpretation. An example of such a table is shown in Table 5.2, which shows one of the default Huffman tables provided by the JPEG standard (and as such, used by many encoders).

| Code | Length |
|---|---|
| `00` | 2 |
| `010` | 3 |
| `011` | 3 |
| `100` | 3 |
| `101` | 3 |
| `110` | 3 |
| `1110` | 4 |
| `11110` | 5 |
| `111110` | 6 |
| `1111110` | 7 |
| `11111110` | 8 |
| `111111110` | 9 |

Table 5.2: One of the default Huffman tables

Huffman codes are generated in such a way that there is never any ambiguity during decoding: the shortest possible matching stream of bits that matches a code always maps to that code, since there is never a longer code with that value as prefix. An observation that holds on many Huffman tables (including all the default tables provided by the JPEG standard) is that often not all combinations of the longest code length are all used, such as the code `111111111` (of length 9) which does not appear in Table 5.2.

This is actually a side-effect of design choice in the algorithm to keep its complexity low. For example, in Table 5.2, it would have provided a marginally better compression rate to have two tokens of size 8, instead of one of size 8 and one of size 9. This would have kept the table identical, except that it would have eliminated the trailing `0` on the longest code and would have ensured that all codes of valid sizes could be translated. In many cases, it would simply not be useful to map each possible code at the highest length to a useful symbol, leaving the possibility for codes of valid length to exist that do not map to any symbol.

Due to this design, there is an opportunity to validate a stream that uses a Huffman table with this characteristic. In the above example of Table 5.2, whenever a Huffman code of length 9 is encountered, this value must be `111111110`, since `111111111` is of the same length but not defined in the Huffman table.

### 5.3.2 Validation using quantization array overflows

The process of decoding Huffman codes into symbols leads to values that must be stored in a quantization array. Each first code maps to a length value that describes a value that appears in the data after the Huffman code, which must be stored in the first location in the quantization array. The following Huffman codes map to symbols that each encode two 4-bit values in a single byte. These two values (along with possibly some additional bits in the input) represent a run-length encoded description of one or more values to be inserted into the quantization array.

This process leads to another possibility to detect errors in the entropy-coded data. If the description leads to an overflow in filling out the quantization array, an error has been detected in the data.

An example of such an error is that if 62 of the 64 values in the array have been filled, only a few possible values are allowed: two descriptions of two individual values of a single byte, a run-length encoded description with a size that equals 2 bytes or the terminator special value that denotes that the rest of the array consists of zeros. Any other value that appears is invalid since it would lead to an overflow of the specified quantization array size.

### 5.3.3 Entropy-coded data validation

When validating the entropy-coded data, some considerations need to be taken. First, JPEG allows escaped values in the entropy-coded data, that always start with `0xFF`, so whenever at the start of a full byte this value is encountered, it needs to be interpreted according to the rules defined. This can include restart markers, a literal byte value of `0xFF` or another marker (only EOI is allowed).

Any incorrect values encountered at this level will immediately lead to a validation error. Examples are a restart marker without the header of the JPEG specifying them or a restart marker with an incorrect increment. Additionally, an escaped value that does not need to be escaped or an EOI marker not at the end of a logical block of decoded data both also denote a validation error.

The smallest amounts of data that can be decoded together are called Minimum Coded Units (MCU). In our example we will assume a non-chromatic subsampled baseline JPEG. The principles apply to all other common variants, but are left out here to simplify the description.

An MCU consists of 3 color channels (Y, Cb, Cr), and each color channel has 64 coefficients stored in a 8*8 matrix. The first value of this matrix is a DC value, and decoded using a Huffman table specifically defined for DC values. The next 63 values are AC values, and decoded using a separate Huffman table specific to AC values. As such, an MCU can be seen as a 3*8*8 matrix (or as 3 8*8 matrices). DC (Direct Current) values represent the average color intensity of an image block, capturing the baseline level of brightness. AC (Alternating Current) values, on the other hand, detail the variations in color intensity and texture within the block, encoding the finer details and contrasts. Together, DC and AC values enable efficient image compression by separately managing basic intensity and intricate details.

### 5.3.4 Algorithm for baseline JPEGs

The pseudocode for baseline JPEG validation is detailed in Algorithm 1. This algorithm demonstrates how to leverage the two bit-level validation techniques to detect fragmentation points in baseline JPEGs.

The error checks are the bold-faced If-statements in Algorithm 1. If they evaluate to true, an error has been encountered in the entropy-coded data and the location of the byte where the associated data was read from is reported as the first location where validation failed. The block this location resides in can then be considered to not be part of a valid JPEG file and a fragmentation point has been identified: the end of the directly preceding block.

**Algorithm 1** JPEG Baseline validation algorithm

```
[For each color channel] (once for each matrix in the MCU), perform the following steps:
  1. Set QAcounter to 1, b to empty bitstring.
  2. while (b is not a valid Huffman code)
     add next bit of bitstream to b
     If no valid Huffman code found after maximum Huffman code length:
     validation error, report last known good location.
  3. Convert b to symbol via DC Huffman table.
  4. Interpret the symbol as integer and skip this amount of bits; (re)set QAcounter
     to 1.
  5. Until QAcounter is 64, perform the following steps:
     5.1. Read bits from bitstream until the shortest valid Huffman code is encountered
          If not found: validation error, report last known good location.
     5.2. Use AC Huffman table to convert found code to symbol.
     5.3. # Validate quantization array
          Interpret the symbol:
          5.3.1. If the symbol is 0x00: the quantization array is complete, set QAcounter
                 to 64, skip to next iteration.
          5.3.2. If the symbol is 0xF0: add 16 to QAcounter.
          5.3.3. For all other values, split symbol into upper and lower nibble.
                 • Add int(upper nibble) + 1 to QAcounter
                 • Skip int(lower nibble) number of bits in bitstream
     5.4. If counter > 64: validation error, report last known good location.
  6. # Validation succeeded
     return true
```

### 5.3.5 Validation of progressive JPEGs

Progressive JPEGs contain multiple Start-of-scan (SOS) markers. Moreover, all progressive JPEGs use spectral selection encoding. This encoding method necessitates multiple scans, each refining the image's detail level. Spectral selection affects how QA-overflow validation works. Lastly, progressive JPEGs may also make use of successive approximation encoding. Both encodings enable additional validation opportunities.

**Spectral selection.** Spectral selection only fills the quantization array for a specific range of values, starting from the first value. Each scan (indicated by a SOS-marker) adds a specific number of values to the quantization array. The number of values to be added is part of the SOS-marker. This alters the QA-overflow validation method, since each SOS-marker determines the upper bound for QA-indices to be filled for that scan.

**JPEG structure validation using SOS markers.** Each SOS marker must be present from the first bit following completion of the previous scan. In addition, each next SOS marker must cover QA-indices adjacent to the previously filled range. Violating either of these requirements constitutes a validation error.

**Successive approximation.** Successive approximation is a refinement of spectral selection. In successive approximation, quantization values are stored per 8 bits. In the first scan (denoted as 'DC-first' or 'AC-first' in the validation algorithm), a bit sequence of at least one bit (commonly 6 or 7 bits) is decoded. In the refinement scans, a single bit per quantization value is added. This means that the decoded Huffman value for a refinement scan must have a lower nibble with value '1', since the lower nibble of the decoded Huffman value determines the number of bits to be read. A lower nibble with a different value than '1' in a refinement scan thus constitutes a (coefficient length) validation error.

### 5.3.6 Algorithm for progressive JPEGs

The description here only shows the validation mechanisms related to Huffman table lookup errors, quantization array size overflow, and violations of the single-bit refinement used in successive approximation. Interestingly, JPEGs need not be optimally encoded. The specification itself leaves room for suboptimal (or inefficient) use of Huffman encoding as well as suboptimal run-length encoding. Such inefficiencies are still valid within the JPEG standard, and thus do not cause a validation error.

The validation algorithm has four separate decoding phases. Initial quantization values are set in the 'DC-first' and AC-first' phase, respectively. Progressive JPEGs that only use spectral selection only use these two phases, the other two phases are only used for JPEGs containing successive approximation encoded values. More specifically, the 'DC-refine' and 'AC-refine' phases add bits to an existing QA-value.

The resulting validation algorithm is split up in the Main Algorithm 2, and its four subalgorithms, namely Algorithms 3, 4, 5 and 6.

**Algorithm 2** JPEG progressive validation: main algorithm

```
Main algorithm
  1. For each SOS-marker:
      (a) Determine SOS-type & validate spectral selection range:
          DC-first, DC-refine, AC-first, or AC-refine
      (b) For all channels in this scan, for each MCU:
          perform relevant validation case
  2. # validation succeeded
     return true
```

**Algorithm 3** JPEG progressive validation: DC-first

```
  1. Set b to empty bitstring.
  2. while (b is not a valid Huffman code)
     add next bit of bitstream to b
     If no valid Huffman code found after maximum Huffman code length:
     validation error, report last known good location.
  3. Convert b to symbol via DC Huffman table.
  4. Interpret the symbol as integer and skip this amount of bits;
```

**Algorithm 4** JPEG progressive validation: AC-first

```
  1. Set QAcounter to start-of-spectral-selection, b to empty bitstring
  2. while (b is not a valid Huffman code)
     add next bit of bitstream to b
     If no valid Huffman code found after maximum Huffman code length:
     validation error, report last known good location.
  3. # Validate quantization array
     Interpret the symbol:
     3.1. If the symbol is 0x00: the quantization array is complete, skip to next
          iteration.
     3.2. If the symbol is 0xF0: add 16 to QAcounter.
     3.3. For all other values, split symbol into upper and lower nibble.
            • Add int(upper nibble) + 1 to QAcounter
            • Skip int(lower nibble) number of bits in bitstream
  4. If QAcounter > end-of-spectral-selection: validation error, report last known
     good location.
```

**Algorithm 5** JPEG progressive validation: DC-refine

```
Case: DC-refine
  1. skip one bit for each MCU.
```

**Algorithm 6** JPEG progressive validation: AC-refine

```
1. for QAcntr = SpectralSelection-start to SpectralSelection-end:
   (a) while (b is not a valid Huffman code)
       add next bit of bitstream to b
       If no valid Huffman code found after maximum Huffman code
       length: validation error, report last known good location.
   (b) Split the symbol into a symbol into upper and lower nibble.
   (c) # Check for coefficient length error
       If the lower nibble is not 1:
       validation error, report last known good location.
   (d) Read one bit, add its value to the least significant position at QAcntr.
   (e) Set ZSKIP = upper nibble (i.e., #zeroes to be skipped)
       If ZSKIP > zeroes available in spectral selection range:
       validation error, report last known good location.
   (f) Skip ZSKIP zeroes in the bitstream
```

### 5.3.7 Runtime-performance

Our algorithm, designed for integration with a file carver, is not I/O-intensive. Compared to the tasks performed by a standard JPEG decoder, our implementation involves a subset of these operations, primarily focusing on image validation rather than display. Based on this, we anticipate that the algorithm will not significantly impact the overall runtime performance of a file carver.

## 5.4 Example and analysis

In this section we provide an example that shows the validation of a single MCU from a JPEG file, as a practical illustration of the algorithm. Additionally, we analyze how quickly the algorithm would be expected to detect an error when encountering incorrect data. Finally, we discuss some of the considerations and limitations of applying this approach in practice.

### 5.4.1 Example: validating a single MCU

Table 5.5 contains a representation of 10 bytes that together make up a single MCU from a JPEG file generated by the GIMP application on Linux[2]. Since

[2]We omit a detailed description of the process and settings since any JPEG file that employs Huffman compression should suffice for this example.

GIMP generates optimized Huffman tables, the example MCU does not use the standard tables as defined by the JPEG standard. Table 5.3 (for DC values) and Table 5.4 (for AC values) show the generated optimized Huffman tables as decoded from the JPEG file that contains the example MCU. In this case, a total of 4 Huffman tables were generated, which is common.

Furthermore, the file is a baseline (as opposed to progressive) encoded JPEG that does not employ chroma subsampling. As discussed before, all principles apply across all common types of JPEG files, but this configuration was chosen because of its relative simplicity for this example.

| DC#0 | | DC#1 | |
|---|---|---|---|
| Code | Symbol | Code | Symbol |
| 0 | 00000101 | 00 | 00000000 |
| 10 | 00000100 | 01 | 00000001 |
| 110 | 00000110 | 10 | 00000100 |
| 1110 | 00000010 | 110 | 00000010 |
| 11110 | 00000011 | 1110 | 00000011 |

Table 5.3: Decoding example: DC Huffman tables

Table 5.5 has five columns. The first column (Offset) contains the relative offset of the value in the MCU. The second column (Values) contains the value of the byte(s) at that offset in binary encoding. Two bytes are shown when applicable, and this illustrates clearly how Huffman-compression crosses byte boundaries. The third column (Huffman table lookup) refers to a translation from a Huffman code to a symbol, if that row contains such an action. First, the applicable Huffman table is named (referencing the names in the headers of Table 5.3 and Table 5.4) and next to it the code that was matched. The translated value (the symbol) appears as a result of the lookup in the appropriate Huffman table. If no match is found the validation fails.

The fourth column shows the interpretation, in case something else needs to be done except translate a Huffman code to a symbol. In all cases this refers to interpreting the resulting symbol, along with skipping bits in the input, which would normally be read as a value to be stored in the quantization array. Since a validator is not interested in decoding a JPEG file but simply recognizing it for validation purposes, the actual values are ignored and instead a counter is incremented to note that a value would have been read.

| AC#0 | | AC#1 | |
|---|---|---|---|
| Code | Symbol | Code | Symbol |
| 00 | 00000001 | 00 | 00000000 |
| 01 | 00000010 | 01 | 00000001 |
| 100 | 00000011 | 10 | 00010001 |
| 1010 | 00000000 | 110 | 00000010 |
| 1011 | 00000100 | 11100 | 00000011 |
| 1100 | 00010001 | 11101 | 00010010 |
| 1101 | 00100001 | 111100 | 00100001 |
| 11100 | 00010010 | 111101 | 00110001 |
| 11101 | 00110001 | 1111100 | 00010011 |
| 111100 | 01000001 | 1111101 | 00100010 |
| 1111010 | 00100010 | 1111110 | 01000001 |
| 1111011 | 01100001 | 11111110 | 01010001 |
| 11111000 | 00000101 | | |
| 11111001 | 00010011 | | |
| 11111010 | 00010100 | | |
| 11111011 | 00010101 | | |
| 11111100 | 00100011 | | |
| 11111101 | 01010001 | | |
| 11111110 | 10100001 | | |
| 111111110 | 10110001 | | |

Table 5.4: Decoding example: AC Huffman tables

| Offset(s) | Values | Huffman table lookup | Interpretation | Counter |
|---|---|---|---|---|
| 0 | $\texttt{\underline{110}01111}$ | DC#0: 110 ⇒ 00000110 | One value: skip 6 bits, counter = 1 | #1=1 |
| 0, 1 | $\texttt{110\underline{01111\ 1}1011110}$ | | 6 bits skipped | |
| 1 | $\texttt{1\underline{1011}110}$ | AC#0: 1011 ⇒ 00000100 | No zeros, one value: skip 4 bits, counter+1 | #1=2 |
| 1, 2 | $\texttt{11011\underline{110\ 0}0110111}$ | | 4 skipped bits | |
| 2 | $\texttt{0\underline{01}10111}$ | AC#0: 01 ⇒ 00000010 | No zeros, one value: skip 2 bits, counter+1 | #1=3 |
| 2 | $\texttt{001\underline{10}111}$ | | 2 skipped bits | |
| 2, 3 | $\texttt{00110\underline{111\ 00}010010}$ | AC#0: 11100 ⇒ 00010010 | One zero, one value: skip 2 bits, counter+2 | #1=5 |
| 3 | $\texttt{00\underline{01}0010}$ | | 2 skipped bits | |
| 3 | $\texttt{0001\underline{00}10}$ | AC#0: 00 ⇒ 00000001 | No zeros, one value: skip 1 bit, counter+1 | #1=6 |
| 3 | $\texttt{000100\underline{1}0}$ | | 1 skipped bit | |
| 3, 4 | $\texttt{0001001\underline{0\ 0}0001010}$ | AC#0: 00 ⇒ 00000001 | No zeros, one value: skip 1 bit, counter+1 | #1=7 |
| 4 | $\texttt{0\underline{0}001010}$ | | 1 skipped bit | |
| 4 | $\texttt{00\underline{00}1010}$ | AC#0: 00 ⇒ 00000001 | No zeros, one value: skip 1 bit, counter+1 | #1=8 |
| 4 | $\texttt{0000\underline{1}010}$ | | 1 skipped bit | |
| 4 | $\texttt{00001\underline{01}0}$ | AC#0: 01 ⇒ 00000010 | No zeros, one value: skip 2 bits, counter+1 | #1=9 |
| 4, 5 | $\texttt{0000101\underline{0\ 1}1111001}$ | | 2 skipped bits | |
| 5 | $\texttt{1\underline{111100}1}$ | AC#0: 111100 ⇒ 01000001 | 4 zeros, one value: skip 1 bit, counter+5 | #1=14 |
| 5 | $\texttt{1111100\underline{1}}$ | | 1 skipped bit | |
| 6 | $\texttt{\underline{11101}011}$ | AC#0: 11101 ⇒ 00110001 | 3 zeros, one value: skip 1 bit, counter+4 | #1=18 |
| 6 | $\texttt{11101\underline{0}11}$ | | 1 skipped bit | |
| 6, 7 | $\texttt{111010\underline{11\ 00}110100}$ | AC#0: 1100 ⇒ 00010001 | 1 zero, one value: skip 1 bit, counter+2 | #1=20 |
| 7 | $\texttt{00\underline{1}10100}$ | | 1 skipped bit | |
| 7 | $\texttt{001\underline{1010}0}$ | AC#0: 1010 ⇒ 00000000 | Fill with zeros, counter=64, next array | #1=64 |
| 7, 8 | $\texttt{0011010\underline{0\ 1}1010000}$ | DC#1: 01 ⇒ 00000001 | skip one bit, counter = 1 | #2=1 |
| 8 | $\texttt{1\underline{1}010000}$ | | 1 skipped bit | |
| 8 | $\texttt{11\underline{01}0000}$ | AC#1: 01 ⇒ 00000001 | No zeros, one value: skip 1 bit | |
| 8 | $\texttt{1101\underline{0}000}$ | | | |
| 8 | $\texttt{11010\underline{00}0}$ | AC#1: 00 ⇒ 00000000 | Fill with zeros, counter=64, next array | #2=64 |
| 8, 9 | $\texttt{1101000\underline{0\ 0}0110010}$ | DC#1: 00 ⇒ 00000000 | Skip 0 bits, counter=1 | #3=1 |
| 9 | $\texttt{0\underline{01}10010}$ | AC#1: 01 ⇒ 00000001 | No zeros, one value: skip 1 bit | #3=2 |
| 9 | $\texttt{001\underline{1}0010}$ | | 1 skipped bit | |
| 9 | $\texttt{0011\underline{00}10}$ | AC#1: 00 ⇒ 00000000 | Fill with zeros, counter=64, next MCU | #3=64 |

Table 5.5: Decoding example: entropy-coded data representing a single MCU in a JPEG file.

### 5.4.2 Detecting errors

The example in Table 5.5 covers only 10 bytes but has many opportunities for validation errors to occur: 18 times a Huffman code is read from the input. Since none of the four generated Huffman tables use all possible values at the longest code size, every time a Huffman code is read it can potentially have a correct size but an incorrect value (e.g., for DC#0 in Table 5.3 this would be `11111`, since the only defined code at length 5 is `11110`). In addition, three quantization arrays are filled, which can all potentially lead to a size overflow if more than 64 values are defined for it in the input.

As a result, these 10 bytes constituting a single MCU have 21 validation opportunities. In order for this validation approach to be usable to detect a fragmentation point in the entropy-coded data of a JPEG file, only one single validation needs to fail in an entire block that typically has a size of 4096 bytes (or bigger).

### 5.4.3 Practical analysis

To gain an idea of the potential usefulness in practice, the following sections investigate the success rates of both methods of validating entropy-coded data. They are separately discussed since they have different characteristics, mostly because they operate on different levels: the Huffman codes are directly present in the data while the quantization array values are discovered after a step of decoding. For both methods, we assume a block size of 4096 bytes since that is the minimum encountered in practice on all modern systems.

#### Invalid Huffman codes

Each Huffman table lookup has a chance of failing, which in turn fails validation on the entire JPEG file, which indicates that the current block is not part of a valid JPEG file. The success rate of this approach depends on both the maximum length of each Huffman table that is used, as well as the amount of lookups that are performed within a block. In the example in this section, the chance for a failed lookup for each Huffman table is $(1/2)^5$ (DC#0), $(1/2)^4$ (DC#1), $(1/2)^9$ (AC#0) and $(1/2)^8$ (AC#1). These values represent the chance of encountering an invalid bit at the end of a string of bits of the longest size in the respective Huffman table.

On average, in this example 1.8 lookups are performed per byte. Extrapolated to 4096 bytes, that would result in more than 7,000 lookups per block. Given that only one lookup needs to be erroneous to establish that a fragmentation point has passed, the odds of this method being successful are extremely high even though each individual chance is low. However, considering JPEG files with longer maximum Huffman code sizes, the odds change, but since every JPEG potentially has a different set of tables, it is difficult to estimate a realistic worst case.

#### Quantization array size overflows

The success rate of a validation failure caused by a quantization array size overflow depends not on reading a single value in the input data, but relies on a longer sequence of AC-huffman table lookups that together cause the total size to overflow. A typical MCU consists of three quantization arrays. A small investigation of different JPEG images made with different cameras shows that there are between 490 and 640 arrays per block.

As with detecting invalid Huffman codes, only one quantization array size overflow is needed for validation to fail. In reference to the algorithm explained in Section 5.4.3, the loop at step 3 must be repeated without 3.2.1 occurring or the loop finishing at exactly 64. The chance of this happening will be dependent on both the used quantization tables while encoding (i.e., the lower the compression due to quantization, the higher the chance of success), as well as the Huffman code used to decode the special 0x00 symbol. The longer that specific Huffman code is, the higher the chance of detecting a quantization array size overflow.

Given the high interdependence between many factors it seems unhelpful to speculate about practical odds. These factors include the sizes of the relevant Huffman tables, the size of the code in these tables that maps to the special `0x00` symbol and during a decode process how far along the construction of a quantization array is. Instead, validating quantization array sizes can better be considered an almost free extra chance to discover errors that incurs no performance penalty to the validator and may sometimes yield a validation error.

### 5.4.4 Limitations and considerations

Even though the expected success rate of invalid Huffman code detection looks promising, this approach has some limitations that must be considered when applying it in practice.

Empty blocks (zero-blocks) will not necessarily be recognized as incorrect, due to one or more zeros likely mapping to reasonable symbols in each Huffman table. However, recognizing and removing (partially) empty blocks is typically a concern handled by the file carver that assembles candidate files (e.g., by excluding blocks with very low entropy as not being candidates for compressed data).

When the validator encounters other JPEG fragments that are encoded with the same Huffman tables, chances decrease of this validation technique being successful. However, the entropy-coded data is not automatically byte-aligned, so fragmentation can occur at any point in the entropy-coded data. Thus, fragmentation point detection is still possible in this case, but the odds decrease slightly in case the fragments end up being aligned.

Even if validation does succeed in an entire block that is not part of the original JPEG file, subsequent blocks can still fail to validate. The first block (or more) after the fragmentation point will then be missed by the file carver, but in this way a partial file may potentially still be recovered.

The described approach will work when the default Huffman tables are used, which are very popular especially with digital cameras. The Huffman tables as they are presented in the JPEG specification were not meant as a standard, but were presented as a good reference point. However, these tables became a de facto standard [web-Has21] because calculating custom Huffman tables for each picture comes with processing costs, with only limited gains with in compression efficiency. Since each default table in the JPEG spec has the same property as described in Section 5.4.3, both the default Huffman tables and many custom-generated Huffman tables will work with this approach.

## 5.5 Related work

Since the early 2000s, the recovery of deleted computer files has been a topic of extensive research, as highlighted by Pal and Memon [PM09]. Recovery strategies primarily fall into two main categories: file structure and content-based approaches, though there are other methods and combinations of these methods as well.

**JPEG file structure.** Research on recovering JPEG files often rely on the identification of JPEG markers. The work by Mohamad and Deris [MD09] focuses on the Define Huffman Table (DHT) marker in a JPEG files. By analyzing the length fields of the DHT marker, they determined how to validate DHT data, highlighting fragmentation points if the subsequent data fails this validation. The effectiveness of this method hinges on the frequency of fragmentation within the JPEG's DHT section.

In a study on thumbnail carving using image pattern matching, Abdullah et al. [AIM13] use the fact that each JPEG starts and ends with a Start of image

(SOI) and End of image marker (EOI). Thumbnails, smaller versions of the original picture, found within the JPEG header, also use these markers. Therefore, a JPEG might house multiple SOI/EOI pairs. When tested on the DFRWS 2006 and 2007 datasets, their file carver flagged 4 files incorrectly as thumbnails but surpassed a benchmark algorithm, recovering 31 compared to the latter's 28 files.

Validation categories can also be combined: Karresand and Shahmehri [KS08] use both the restart markers and their expected occurrence within a file in addition to using metrics in the expected maximum change in luminosity in order to try to reassemble JPEG files.

In their work, Fei and Adbullah [FA20] introduced a file carver tailored for the recovery of in-order fragmented JPEGs, using the file's structure. When tested on the DFRWS 2006 dataset, this carver successfully retrieved 8 of the 12 JPEG images.

**Content-based approach.** The recovery of JPEG files can also be approached through analyzing the visual representation of its data—a content-based strategy. Most studies interpreting potential JPEG image data often incorporate knowledge of the JPEG file structure to optimize their analyses. Memon and Pal [MP06] focus on file fragment classification and present techniques for image reconstruction. Using a greedy search algorithm, the sum of differences metric outperformed a pixel matching strategy for file recovery purposes. Li et al. [LSC$^+$11] delve into artifacts emerging from fragmented or corrupted data. Notably, they highlight the DC-values, which are delta encoded, as indicators of sudden color shifts. Furthermore, they demonstrate that the distribution of AC-values can signal errors in JPEG data. Uzun and Sencar [US15] propose a method to infer Huffman tables, subsampling ratios, and quantization values for dealing with a missing JPEG header. They analyzed statistics from photos uploaded to Flickr to determine common values for JPEG decoding metadata. Given that 99.5% of the Flickr dataset was encoded with default Huffman tables, their method offers a reliable way to discern the remaining decoder settings for JPEG fragments. Birmingham et al. [BFV17] leverage the embedded thumbnail within the JPEG header to predict the primary image's characteristics. By applying a probabilistic model centered on thumbnail affinity, they showcase this method's capacity to pinpoint invalid JPEG data.

**Recovery related to metadata.** Metadata may be derived from image data, and be used for image identification. In their research, Thai et al. [TCR$^+$17] proposed a method to estimate quantization steps for an image originally compressed as JPEG but later saved in a lossless format. Through their analysis, they demonstrated that a fingerprint technique could suggest potential true quantization steps, achieving over 99% identification accuracy on grayscale images from the Dresden database (Gloe and Böhme [GB10]).

Unlike traditional digital cameras, smartphones frequently undergo software updates and setting modifications. In a comprehensive study on Apple smartphones, Mullan et al. [MRF19] explored how evolving software might influence source identification. Using machine learning, they devised classifications from EXIF data and quantization matrices. Their findings reveal that while EXIF headers and JPEG quantization table values can effectively differentiate specific apps or OS versions, identifying images from smartphones proves more challenging than from standard digital cameras.

**Statistics oriented.** Another perspective on JPEG recovery involves statistical analysis. Pal et al. [PSM08] combine syntactical tests (keywords and file head matching) statistical tests in the form of entropy analysis. With a matching-metric (tuned with a training-set of images), chances are calculated for the likeliness of two fragments belonging together. Nescar and Memon [SM09] propose a method to match JPEG fragments to JPEG headers, based on pattern recognition in the encoded data. Kadir et al. [KAC15] employed statistical byte frequency analysis to distinguish groups of JPEG fragments, noting that each image exhibits distinct characteristics. Their study on 4 JPEG files indicated that byte frequency analysis unveiled multiple unique patterns. Taking a similar route, Tang et al. [TFC+16] introduced a novel similarity metric, the Coherence of Euclidean Distance (CED), to determine if two data blocks belong to the same JPEG. Their results showed the CED algorithm outshining the Adriot Photo Forensics (APF) in file recovery. For 3-piece JPEGS, CED recovered 96 out of 109 files, whereas APF managed 66. For 4-piece JPEGS, CED retrieved 61 out of 75, with APF securing only 32. Lastly, Azhan et al. [AIR+22] developed the Error Level Analysis technique to pinpoint the distinct signature of 8x8-pixel JPEG blocks. Their tests on 21 JPEG images demonstrated the uniqueness of each block.

**Camera sensor information.** Each camera's sensor introduces unique noise to an image. This sensor noise can determine if an image originated from a specific camera (Lukás et al. [LFG06]). Building on this concept, Durmus et al. [DMT+17] demonstrated how JPEG fragments can be both attributed to a particular camera and pinpointed to their location within an image, assuming the originating camera is known. To verify fragment correctness, the researchers employed Sum-of-Differences and Histogram Differences. In tests, their method achieved a true positive rate of 94.2%, correctly identifying 21,713 out of 23,040 fragments. In a subsequent study, Durmus et al. [DKM19] noted the limitations of their earlier work, especially its potential weaknesses under real-world conditions due to overlooked brightness and color artifacts. To address this, they introduced a compatibility metric for fragment matching and subsequent image stitching. They tested their approach on 2,000 images from a single camera, all converted to JPEG with identical quality settings. Results showed a 52.4% correct fragment identification rate for JPEGs at a quality factor of 90, and a 42.0% identification rate for those at 80.

**JPEG file carvers (other).** De Bock and de Smet [dBdS16] presented a novel file carving approach, implemented in the tool JPGcarve, which employs an external decoder library (libjpeg-turbo) as a validation mechanism for the JPEG data. While validating JPEG data, the decoder either processes it successfully or fails, indicating a fragmentation point. The file carver itself includes support for single- and multifragment file recovery, and search space reduction techniques. In tests across six datasets, JPEGcarve successfully recovered all multi-fragmented JPEGs, totaling 46 images.

Further advancing file carving methods, Ali and Mohamad [AM21] introduced RX_myKarve, combining the Extreme Learning Machine and JPEG structure validation. This dual approach classifies file fragments to distinguish between JPEG and non-JPEG fragments. Subsequent structure-based carving aids in JPEG reconstruction. The authors highlight its efficacy, noting the recovery of all 19 images from the DFRWS 2006 dataset and 18 from the DFRWS 2007 dataset.

**Generic file recovery.** Not all JPEG recovery techniques are exclusive to the JPEG file format. Generic file recovery approaches can sometimes be applicable to JPEG. For example, Ying and Thing [YT10] posed file fragment reconstruction as a graph-theoretic challenge. In a test involving 10 files, their method surpassed a brute-force technique, successfully restoring all files to their original state.

In a study on hash-based file carving, Garfinkel and McCarrin [GM15] introduce a modified whole file hashing approach. While the hash value of known files can identify intact files, this method struggles with fragmented, altered, or incomplete files. With their hash-based carving technique, centered on individual data blocks of a target file and leveraging a target hash database, the authors demonstrate the feasibility of this approach.

Leveraging Convolutional Neural Networks (CNN), Ghaleb et al. [GSF$^{+}$23] unveil a light-weight file fragment classification model. They report enhanced time efficiency and comparable accuracy to earlier CNNs, achieving 79% accuracy on the FFT-75 dataset. However, the team echoes prior findings, pointing out the challenge in classifying high-entropy file fragments due to their lack of distinguishable statistical patterns.

## 5.6 Conclusion

Even though file carving is a powerful data recovery technique used in many investigations in digital forensics, it would be even more useful if it could reliably recover fragmented files in especially the most relevant file format: JPEG. Unfortunately, JPEG files consist almost entirely of large blobs of highly compressed entropy-coded data, making it very difficult to construct a reliable validator to aid file carvers in recovering fragmented files.

In this chapter we describe an approach that leverages two observations about many JPEG files in practice. First, the Huffman tables used to decode a large proportion of the entropy-coded data often do not use all possible code values at their longest code length, offering possibilities to detect errors when invalid codes are encountered. Second, after translating Huffman codes to symbols, the next step in decoding involves filling quantization arrays with exactly 64 values, offering another possibility to detect errors when an overflow is encountered.

This chapter describes algorithms to validate the entropy-coded data in both baseline JPEGs and progressive JPEGs using these two observations and finds that the odds of finding fragmentation points in practice are quite high, especially with regard to invalid Huffman codes. It will work with the example Huffman tables provide by the JPEG standard that are used by many digital cameras, but also with many optimized Huffman tables generated by specialized applications.

The next step is to implement this approach to work with a file carver designed to recover fragmented files and report on practical findings.

# Chapter 6

# Problem solved: a reliable, deterministic method for JPEG fragmentation point detection

*This chapter presents an adapted version of the paper* Problem solved: a reliable, deterministic method for JPEG fragmentation point detection*, by Vincent van der Meer, Jeroen van den Bos, Hugo Jonker, and Laurent Dassen, which was accepted at the 11th annual Digital Forensics Research Conference Europe [DFRWS-EU24]. It was awarded the DFRWS EU 2024 Best Paper Award.*

**Abstract** Recovery of deleted JPEG files is severely hindered by fragmentation. Current state-of-the-art JPEG file recovery methods rely on content-based approaches. That is, they consider whether a sequence of bytes translates into a consistent picture based on its visual representation, treating fragmentation indirectly, with varying results. In contrast, in this chapter, we focus on identifying fragmentation points on bit-level, that is, identifying whether a candidate next block of bytes is a valid extension of the current JPEG. Concretely, we implement and exhaustively test the deterministic algorithms for finding fragmentation points in JPEGs, as presented in the previous chapter. Even in the worst case scenario, our implementation finds over 99.4% of fragmentation points within 4 kB – i.e., within the standard block size on NTFS and exFAT file systems. As such, we consider the problem of detecting JPEG fragmentation points solved.

## 6.1 Introduction

Photos can contain crucial evidence in forensics. File recovery (i.e., the reconstruction of deleted digital files) is an essential forensic capability. Since the JPEG file format is the most widely used digital storage format for photos [HLN+18], JPEG file recovery is an important and ongoing field of research. Recovery of JPEG files is severely hindered by file fragmentation. While not all JPEG files represent photos, those that do are sufficiently large (a few megabytes) that roughly ∼8% are likely to be fragmented (Chapter 3, Table 3.4) on NTFS file systems. Exacerbating this problem is the fact that JPEG files consist mostly of high-entropy data, without markers or checksums that can validate partial (nor full) integrity. This seems to imply that there is hardly any relation between the bytes of a JPEG file. Consequently, most existing literature on JPEG file recovery has thus focused on one of two approaches. The first focuses on recovering non-fragmented JPEG files based on header and footer matching, for example, the works by Karresand and Shahmehri [KS08] or Fei and Abdullah [FA20]. The second approach focuses on recovering any JPEG file using visual compatibility, such as the works by Li et al. [LSC+11] or Tang et al. [TFC+16].

However, in Chapter 5 we showed that there are internal consistency requirements for the high-entropy portion of a JPEG file. We proposed a theoretical algorithm that would leverage these consistency requirements to determine whether a block of bytes, as stored on-disk, is a valid continuation of an initial part of a JPEG file. While we showed feasibility, we did not implement our approach and therefore did not test efficacy.

This chapter focuses on RQ5, which is stated as: *How effective are the newly identified approaches to JPEG fragmentation point detection in real-world JPEGs?*

**Contributions.** In this chapter, we implement, and measure efficacy of the algorithms introduced in the previous chapter. We implemented a JPEG validator that reports the exact location where the continuation of a JPEG bitstream (specifically within the high-entropy-coded data sections) becomes invalid by

1. Huffman code lookup errors, and
2. quantization array overflow.

For the purpose of testing these validation methods, we gathered a sizeable, relevant dataset ([dataset-vdMee22]) of JPEG files, covering all variants of baseline and progressive JPEGs occurring in the real world. The performance of our validator in detecting fragmentation points is tested against these JPEGs. The results show that our validator achieves phenomenal performance: for the predominant encoding (baseline JPEGs), it achieves a success rate of 99.997% in identifying a fragmentation point.

**Availability.** An open-source implementation of the algorithm proposed in this chapter is available for download from our GitHub repository [artefact-vdBvdM23]. The JPEG datasets, as described in Section 6.3.4, are also included in this repository. We have compiled a comprehensive list of 230,157 JPEG image filenames used in our validation process (see Section 6.3.1). Available as a text file in the repository, this list aids in enhancing transparency and facilitates the reproduction and validation of our research.

## 6.2 Construction of a wide-coverage evaluation test set

To determine efficacy of the algorithms discussed in the previous chapter, we need a real world, diverse set of JPEGs. There are various possible sources for such a set, but not all are equally suitable. For instance, platforms like Instagram or Imgur, which are image-centric social media sites, could serve as potential sources for JPEGs However, these platforms tend to recompress and/or re-encode images in order to optimize storage and bandwidth. This leads to little divergence in relation to used encoders and encoder-settings. An alternative would be photo-sharing sites, such as Flickr. Such sites tend to preserve the original input to avoid altering the intended vision of the image. As such, the original input encoder settings are more likely to be preserved. However, such sites are typically used for high-resolution photos taken by high-end equipment. This also leads to a bias in relation to encoders and encoder-settings. The ideal source should offer a wide range of encoders and/or encoder settings for JPEGs.

One suitable source for this is Wikipedia. Wikipedia requires a very diverse set of images, from photos to maps to diagrams of electrical wiring, the solar system, to geometric proofs, to newspaper scans, etc. Succinctly put, any graphically representable concept can be found on Wikipedia in order to transfer knowledge about the concept. Moreover, Wikipedia is open to contributions from anywhere in the world. This leads to the expectation that a large variety of image creators for a large variety of content can be found there, and thus that a large variety of encoders would be used.

### 6.2.1 Collection & sanitisation

In order to collect JPEG files from Wikipedia, a crawler[1] was used with both `wikipedia.org` and `wikimedia.org` as starting points, with a crawl depth of two. Only files with extension `.jpg` or `.jpeg` (case insensitive) were collected, with a minimal file size of 4097 bytes. [2] We collected 230,157 JPEG files in January 2022. It is important to note that the media files on Wikipedia are subject to various licenses, many of which require attribution. Therefore, while our method of data collection can be replicated, sharing the entire set of collected images directly is not feasible due to these attribution requirements. To facilitate research transparency and reproducibility, the filenames of all JPEG files in this dataset have been compiled and are available in a text file within our GitHub repository [artefact-vdBvdM23].

Files that did not start or end with a valid JPEG marker (i.e., `0xFFD8` or `0xFFD9`) were removed. For bit-identical images, only one image was kept in the dataset.

### 6.2.2 JPEG dataset characteristics

Table 6.1 shows characteristics of the collected dataset. As can be seen, baseline JPEGs are by far the most frequently occurring type of JPEGs.

**Marker occurrence.** Markers in the entropy-coded data can be leveraged for validation. Progressive JPEG files contain additional SOS markers in the entropy-coded data. Both baseline and progressive JPEG files may contain Restart Markers (RST) in entropy-coded data. However, our data collection shows that the rate of occurrence of these markers is rare. Only 0.6% of the collected files are progressive, and only 1.0% of the files contain Restart Markers.

**Chromatic subsampling.** Chromatic subsampling is an optional JPEG feature that reduces the resolution of the chrominance color-channels (Cb, Cr). This reduces the amount of color information by a factor 2 (either horizontal or vertical subsampling) or 4 (both horizontal and vertical subsampling). Therefore, there are twice or even four times as many Huffman table lookups (during decoding) for luminance (Y) compared to a color channel (Cb, Cr). This would bias lookup errors

[1]WFDownloader

[2]Smaller files fit into one default-sized NTFS block, which implies they cannot be fragmented (in default settings). The default block size on exFAT file systems varies with volume size, but is at least 4096 bytes, so the same reasoning applies.

| Description | # files | % of total |
|---|---:|---:|
| JPEG files in dataset | 230,157 | 100.0% |
| Baseline JPEG (SOF0) | 228,793 | 99.4% |
| Progressive JPEG (SOF2) | 1,364 | 0.6% |
| – using spectral selection | 397 | 0.2% |
| – using successive approximation | 967 | 0.4% |
| | | |
| *JPEGs that include or use* | | |
| Grayscale | 26,359 | 11.5% |
| Restart markers | 2,309 | 1.0% |
| Chromatic subsampling | 53,128 | 23.1% |
| – horizontal subsampling | 13,335 | 5.8% |
| – vertical subsampling | 1,855 | 0.8% |
| – horizontal and vertical subsampling | 37,937 | 16.5% |

Table 6.1: JPEG dataset characteristics

to occur more frequently in Huffman tables for the chrominance channel. Nearly a quarter of all JPEGs in the dataset use a form of chromatic subsampling (see Table 6.1). The most common form is both horizontal and vertical subsampling.

### 6.2.3 Huffman code lengths

An important mechanism of validating JPEGs relies on Huffman table lookups (Sec. 5.3.1). If one or more of a JPEG's Huffman tables have short maximum code-lengths, a random bitstream continuation is more likely to trigger a lookup validation error. Therefore, it is relevant to know what Huffman table lengths occur in our dataset. The distribution of maximum Huffman table code lengths is shown in two figures, in Figure 6.1 for Luminance-DC and AC, and in Figure 6.2 for Chrominance-DC and AC.

## 6.3 Evaluation design

### 6.3.1 Validating the validator

We deliberately aimed to collect a dataset of JPEGs that would show a broad diversity in how the file format is used. Even if these violate the official JPEG specification, these files are in use in the real world and therefore we hold that, ideally, all these files should be correctly processed. We fed the validator process

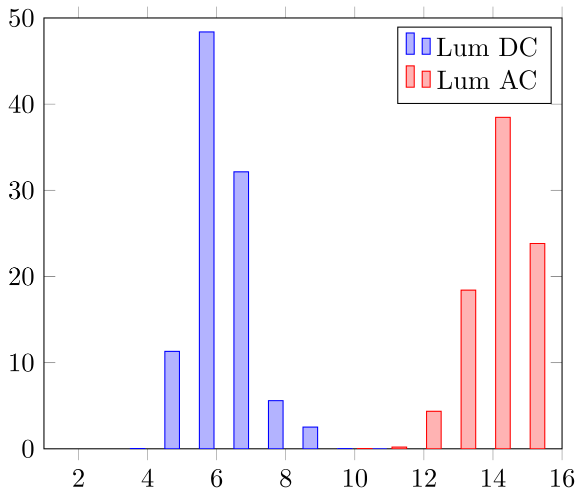

Figure 6.1: Maximum HT code lengths for luminance-DC and luminance-AC

each file in the dataset individually. Our validator (at the time of writing) is able to correctly process 230,149 of 230,157 files. The 8 remaining files exhibit rarely occurring deviations of the JPEG specification. If desired, support for these deviations can still be added in the future.

### 6.3.2 Choosing fragmentation points

To test the validator's efficacy, we will feed it a stream of bits from a JPEG file from the dataset, and then suddenly switch to a random bitstream.

There are three main criteria for determining at which point we want to break the input stream, which simulates a fragmentation point. First, we will only fragment JPEGs after the header, since our validation mechanism is only designed to validate high-entropy coded data sections. The second criteria is to align with the most frequently occurring block size. The default block size on both exFAT and (modern) NTFS file systems is at least 4 kB [web-MS21]. Lastly, for robustness, we opt to measure validation performance at two distinct fragmentation points. These criteria led us to test fragmentation after 16 kB and after 32 kB.

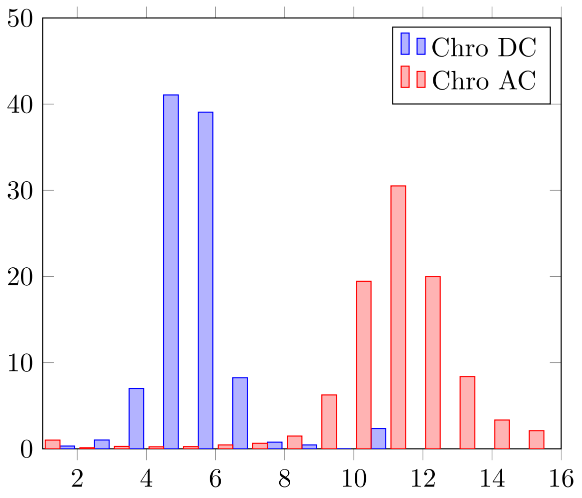

Figure 6.2: Maximum HT code lengths for chrominance DC and chrominance AC

### 6.3.3 Post-fragmentation point data

An important experiment design consideration is what data is presented after a fragmentation point. JPEG's entropy-coded data exhibits high entropy. Various types of file formats tend to use compression, including audio, video, and office file formats. Compressed data is, by definition, high-entropy data and would resemble random data. Therefore, we generate and inject random data after the fragmentation point.

### 6.3.4 Experiment goals

The main goal of our experiments is to determine efficacy of the validation algorithms described in Sec. 5.3.4 and 5.3.6. That is, how well do they perform in identifying fragmentation points? We are interested in:

1. how well they perform against the entire set,
2. the contribution of each validation mechanism to the validation result,
3. evaluating how well validation mechanisms perform in extreme scenarios.

**1. Benchmarking validation success.** We evaluate our implementation of the baseline validation algorithm, as introduced in the previous chapter, using the SOF0-set of baseline JPEGs. Additionally, we evaluate our extensions for progressive validation using the SOF2-set of progressive JPEGs. Since our dataset contains an order of magnitude more baseline than progressive JPEGs, the SOF0 set is 10× as large as the SOF2-set. We select subsets of 1,000 and 100 JPEGs, respectively, from the baseline and progressive sets, of at least 100 kB. Baseline JPEGs may optionally contain any of the following: restart markers, gray scale, or chromatic subsampled JPEGs. Progressive JPEGs may, in addition, also optionally use successive approximation.

**2. Benchmarking the contribution of each validation mechanism.** To this end, we keep track of which validation mechanism raised the error for all test-sets. We include the new coefficient length mechanism (Sec. 5.3.5), the various variants of Huffman lookup table and QA-overflow mechanisms, and existing JPEG file marker mechanisms. It's important to note that validation halts upon detecting the first failure, as any subsequent bits would be incorrect, rendering further validation pointless. These tests thus show which mechanisms trigger the soonest, not how well a validation mechanism performs. The latter would require a different kind of experiment, one where other validation mechanisms are excluded.

**3. Benchmarking extreme cases.** We consider 3 extreme cases for the validation mechanisms: HT-max, consisting of 100 JPEGs with the longest Huffman table codes from the data set; HT-min, consisting of 100 JPEGs with the shortest Huffman table codes; RST, consisting of 100 JPEGs containing restart markers.

The length of the Huffman table codes impacts the likelihood of a lookup error. Longer code length means more bit sequences are valid Huffman symbols. This implies that the longer the code length, the less likely a bit sequence from elsewhere triggers a lookup error. The longest code lengths thus constitute a worst-case scenario for this validation mechanism (HT-max set). Conversely, the shortest code lengths constitute a best-case scenario (HT-min set). Lastly, the RST-set is created to test RST-marker validation. Although restart markers are only present in 1.0% of all JPEG files in the dataset (Table 6.1), RST-markers are one of the few marker-based validation mechanism usable within the entropy-coded data sections (aside from SOS markers in progressive JPEGs).

### 6.3.5 Execution

Both the SOF0- and SOF2-sets underwent tests at two fragmentation offsets: 16 kB and 32 kB. This approach ensures any validation behavior unique to the 16 kB fragmentation point will be revealed. For the other datasets, we conducted the tests using a single fragmentation offset, specifically at 16 kB from the start of the data, as opposed to the dual 16 kB and 32 kB offsets used for the SOF0- and SOF2-sets. Each file is tested 100 times, with each of these 100 tests performed with different random data after the fragmentation point. The number of tests performed per fragmentation offset is therefore 100,000 for the SOF0 set and 10,000 each for the SOF2, HT-max, HT-min, and RST-sets.

## 6.4 Results

The evaluation results are shown in Tables 6.2–6.9.

### 6.4.1 Overall performance

Table 6.2 shows the performance of the algorithms in terms of correctly identifying the fragmentation point within a number of bytes. In Table 6.3, we present distribution of the location (in bytes) at which the validation algorithms detected the fragmentation points in the evaluation. Negative offsets indicate that the last known good location precedes the fragmentation point.

| | SOF0 frag. point at | | SOF2 frag. point at | |
|---|---|---|---|---|
| bytes | 16 kB | 32 kB | 16 kB | 32 kB |
| < 512 | 88.854% | 88.878% | 96.97% | 93.15% |
| < 1,024 | 97.566% | 97.607% | 98.72% | 97.28% |
| < 2,048 | 99.849% | 99.853% | 99.59% | 99.21% |
| < 4,096 | 99.997% | 99.999% | 99.78% | 99.92% |
| < 8,192 | 100.000% | 100.000% | 99.92% | 100.00% |
| < 16,384 | 100.000% | 100.000% | 100.00% | 100.00% |
| ≥ 16,384 | 100.000% | 100.000% | 100.00% | 100.00% |

Table 6.2: Fragmentation point detection within given number of bytes

| | SOF0 frag. point at | | SOF2 frag. point at | |
|---|---|---|---|---|
| | 16 kB | 32 kB | 16 kB | 32 kB |
| Minimum | -3 | -2 | -1 | -25 |
| 25th percentile | 52 | 51 | 8 | 12 |
| 50th percentile (median) | 131 | 129 | 25 | 36 |
| 75th percentile | 285 | 284 | 77 | 124 |
| 95th percentile | 729 | 729 | 340 | 684 |
| 99th percentile | 1,272 | 1,255 | 1,393 | 1,963 |
| 99.9th percentile | 2,193 | 2,193 | 5,712 | 3,773 |
| Maximum | 5,675 | 4,941 | 11,076 | 5,810 |

Table 6.3: Number of bytes from frag. point till validation error (distribution)

### 6.4.2 Contributions of individual validation mechanisms

Tables 6.4 and 6.5 show the contribution of each validation mechanism to fragmentation point detection. The results are split into two tables, since the validation mechanisms needed to be adapted, albeit slightly, for progressive JPEGs. In particular, Huffman encoding validation is performed in four distinct phases for progressive JPEGs (Sec. 5.3.6). Note that the *Coefficient length* validation mechanism did not trigger once in our experiments.

| SOF0 validation type | frag. point at | |
|---|---|---|
| | 16 kB | 32 kB |
| Huffman-DC | 25.944% | 25.801% |
| Huffman-AC | 0.893% | 0.870% |
| QA-overflow | 72.741% | 72.979% |
| Restart marker | 0.422% | 0.350% |
| Start of scan marker | 0.000% | 0.000% |
| End of file marker | 0.000% | 0.000% |

Table 6.4: Baseline (SOF0): Validation per validation type

In addition, Table 6.6 splits the results per validation type and per channel (being luminance (Y), blueness (Cb), and redness (Cr) from the YCbCr color space).

| SOF2 validation type | frag. point at | |
|---|---|---|
| | 16 kB | 32 kB |
| Huffman-DC First | 47.87% | 20.00% |
| Huffman-AC First | 1.47% | 1.07% |
| Huffman-DC Refine | 0.00% | 0.00% |
| Huffman-AC Refine | 0.00% | 0.43% |
| Coefficient length | 0.00% | 0.00% |
| QA-overflow | 49.55% | 76.13% |
| Restart marker | 0.00% | 0.00% |
| Start of scan marker | 1.11% | 2.37% |
| End of file marker | 0.00% | 0.00% |

Table 6.5: Progressive (SOF2): Validation per validation-type

### 6.4.3 Validation performance for extreme cases

Tables 6.7, 6.8, and 6.9, are the counterparts to Tables 6.2, 6.3, and 6.4, respectively. These tables present the performance of the validation algorithm for extreme cases (HT-max,HT-min, and RST-sets).

## 6.5 Analysis of the results

### 6.5.1 Overall performance

Tables 6.2 and 6.3 show within how many bytes fragmentation was detected. Table 6.2 present this data in number of bytes used in block sizes on file systems; Table 6.3 shows the distribution of the number of bytes from the fragmentation point till a validation error occurred.

First, we consider the position of the fragmentation point: 16 kB vs. 32 kB. In a rare few SOF2 cases, fragmentation is not detected within 8,192 bytes from the 32 kB fragmentation point (it always is for the 16 kB point). However, the distributions associated with the two fragmentation points are rather similar, and differences in validation performance are tiny. We therefore conclude that no special validation behaviour should be expected at the 16 kB point.

Secondly, we examine performance on the level of file system blocks. In Table 6.2, we include the full range of allowed block sizes for completeness sake. The default block size on NTFS volumes is 4 kB; on exFAT volumes, 4 kB is the minimum size.[3] The main takeaway from the table then is that, for both SOF0

[3]For exFAT volumes larger than 256 MB, the block size is at least 32 kB.

| Validation type | SOF0 frag. point at | |
|---|---|---|
| | 16 kB | 32 kB |
| Huffman-DC | 25.944% | 25.801% |
| – luminance (Y) | 14.143% | 13.956% |
| – blueness (Cb) | 6.070% | 6.108% |
| – redness (Cr) | 5.731% | 5.737% |
| | | |
| Huffman-AC | 0.893% | 0.870% |
| – luminance | 0.245% | 0.282% |
| – blueness | 0.478% | 0.451% |
| – redness | 0.170% | 0.137% |
| | | |
| QA-overflow | 72.741% | 72.979% |
| – luminance | 68.910% | 69.098% |
| – blueness | 1.948% | 1.941% |
| – redness | 1.883% | 1.940% |

Table 6.6: Baseline (SOF0): Validation types per color channel

and SOF2 JPEGs, our validator is all but certain to determine fragmentation occurred within the first file system block following the fragmentation point. Moreover, the validation mechanism is fully deterministic. These two factors combined enable file carvers to use more efficient search strategies, since there is next to no uncertainty in the results of the validation mechanism.

Lastly we consider the distribution of the amount of bytes after the fragmentation point before a validation error is triggered. In Table 6.3, we see that the last known good location can occur before the fragmentation point. This can happen when JPEG-markers or Huffman table values start close to the fragmentation point. Such cases may end up not continuing / terminating correctly after the fragmentation point, triggering the validation error. The table also shows that the distributions are rather skewed. The 95th and 99th percentiles are within 2 kB from the fragmentation point, and even the 99.9th percentile for SOF0 JPEGs (the most frequently occurring type of JPEGs in our dataset) is slightly less than half the maximum value found.

### 6.5.2 Contribution of individual validation mechanisms

In the previous chapter where we proposed the validation algorithm for SOF0 JPEGs, we hypothesized on the use of quantization array overflows that they

| bytes | HT-max | HT-min | RST |
|---:|---:|---:|---:|
| < 512 | 79.74% | 95.84% | 84.20% |
| < 1,024 | 95.21% | 99.72% | 96.87% |
| < 2,048 | 98.81% | 100.00% | 99.89% |
| < 4,096 | 99.41% | 100.00% | 100.00% |
| < 8,192 | 99.74% | 100.00% | 100.00% |
| < 16,384 | 99.90% | 100.00% | 100.00% |
| < 32,768 | 100.00% | 100.00% | 100.00% |
| ≥ 32,768 | 100.00% | 100.00% | 100.00% |

Table 6.7: Fragmentation point detection within given number of bytes for HT-max, HT-min, and RST-sets

| | HT-max | HT-min | RST |
|---|---:|---:|---:|
| Minimum | -1 | -1 | 0 |
| 25th percentile | 94 | 41 | 78 |
| 50th percentile (median) | 208 | 102 | 175 |
| 75th percentile | 418 | 206 | 359 |
| 95th percentile | 996 | 481 | 871 |
| 99th percentile | 2,468 | 758 | 1,312 |
| 99.9th percentile | 16,253 | 1,344 | 2,049 |
| Maximum | 32,766 | 1,621 | 2,705 |

Table 6.8: Number of bytes from frag. point till validation error (distribution)

"may sometimes yield a validation error". Indeed they do: In Tables 6.4 and 6.5, we see that the QA-overflow mechanism triggers first in ~73% of SOF0 cases and SOF2 cases with a 32 kB fragmentation point.

Interestingly, the SOF2 set at the 16 kB fragmentation point is an outlier. In this case, the Huffman-DC First validation mechanism shows unexpected strong performance. This may be linked to properties of progressive JPEGs. In particular, we recall that progressive JPEGs contain multiple scans of varying length, each of which allows the use of new Huffman tables. We suspect that the fact that Huffman tables may be replaced in progressive JPEGs is part of the reason, but finding a full explanation will require further experimentation.

The difference between the performance of the Huffman-DC and Huffman-AC mechanisms are explained by the fact that DC-tables have a maximum length of 11, whereas the AC-tables have a maximum length of 16. Amplifying this point, Figures 6.1 and 6.2 show that the DC-max lengths often are much shorter than

| Validation type | HT-max | HT-min | RST |
|---|---|---|---|
| Huffman-DC | 5.61% | 28.24% | 10.91% |
| Huffman-AC | 0.34% | 0.93% | 0.36% |
| QA-overflow | 94.03% | 70.83% | 58.92% |
| Restart marker | 0.00% | 0.00% | 29.81% |
| Start of scan marker | 0.00% | 0.00% | 0.00% |
| End of file marker | 0.02% | 0.00% | 0.00% |

Table 6.9: Validation types for HT-max-, HT-min-, and RST-set

their theoretical maximum length compared to their AC counterparts.

When evaluating the results per color-channel (Table 6.6), we see that validation errors are most frequently detected in the luminance channel. A factor that plays a role is chromatic subsampling. First, chromatic subsampling is relatively common (23.1% of SOF0+SOF2 JPEGs). Second, chromatic subsampling reduces the amount of chromatic data (compared to luminance data) by either a factor of 2 or a factor of 4. The number of Huffman table lookups for chrominance are thus reduced by the same factor.

When considering the QA-overflow mechanism, we see an even more dominant result for the luminance channel. This is not surprising: chrominance is often much more compressed than luminance in the quantization tables (see e.g., [tech-II92, Table K.1, K.2]). Chrominance quantization arrays therefore typically contain significantly less non-zero values than luminance arrays. They are therefore more likely to contain a “fill out with zeroes” command, which fills out the quantization array completely and correctly, thereby avoiding triggering a QA-overflow.

### 6.5.3 Validation performance for extreme cases

The HT-max set represents a worst-case scenario for Huffman-based validation mechanisms. Validation for this set is indeed significantly hindered, as shown in Tables 6.7, and 6.8. Table 6.9 shows clearly that the otherwise so reliable Huffman-DC validation mechanism is kneecapped by this set. In several test cases, the End of File marker mechanism was even triggered before any other validation mechanism. This implies that the entire (fragmented) bitstream was considered valid, except only for the absence of an End of File marker at the correct bit.

Conversely the HT-min set constitutes a best-case scenario for this mechanism. The distribution of number of bytes before a validation error was triggered is only slightly better than for the SOF0 set (Table 6.3 vs. Table 6.8). The performance of the Huffman-DC mechanism is only slightly better in this best-case scenario than for the SOF0 set (Table 6.4 vs. Table 6.9). Apparently, the common case is thus not far off from the best-case scenario.

Table 6.9 shows that ∼30% of all validation errors in the RST-set are due to the RST validation mechanism. Even in a dataset where all files contain restart markers, the QA-overflow mechanism remains the dominant validation mechanism.

## 6.6 Conclusion

We implemented the algorithms for fragmentation point detection in both baseline JPEGs and progressive JPEGs, as introduced in our previous chapter. To rigorously evaluate our validator, we assembled a comprehensive test set comprising over 230,000 JPEG files by scraping WikiMedia. This test set encompassed a diverse range of variations in JPEG files, including differences in cameras, encoders, and encoder settings. We tested our validator thoroughly, in different scenarios where each JPEG file was tested 100 times, with different random bitstreams following the fragmentation point. For each test scenario, we focused on specific JPEG properties and specific fragmentation points. Of all the validation mechanisms, the QA-overflow most often was the mechanism that triggered first.

Considering the combined performance of all validation mechanisms, in the worst case scenario, our implementation has over 99.4% probability of correctly invalidating an incorrect bitstream within 4096 bytes of the fragmentation point. For the most common case of baseline JPEGs, the validator achieves over 99.99%. These results are, frankly, astounding – especially given the diversity and quantity of the test set. Therefore, we consider the problem of finding JPEG fragmentation points solved in practice.

# Part III

# Closing remarks

# Chapter 7

# Conclusions and future work

## 7.1 Conclusions

In order to address the main research question, this section will first reiterate and discuss the five subsidiary research questions.

**RQ1:** How can data on file fragmentation be collected in a privacy-friendly manner?

RQ1 raised the critical issue of collecting data on file fragmentation while maintaining the privacy of individuals. In Chapter 2, this research question is explored by devising methodologies and tools committed to respecting and safeguarding personal privacy. The chapter outlines a strategy for acquiring file fragmentation data without revealing the identity of individuals involved. This was achieved through a high-level adoption of privacy- by-design principles, detailed in six stringent requirements and operational constraints, leading to the creation and development of a tailor-made tool facilitating this process. This tool ([artefact-Dol19]) enabled privacy-friendly data collection, but with a minor caveat: although no personally identifiable information was collected, it was sometimes possible to make an educated guess about the identity of specific files. Because this was an unforeseen aspect, and we could not be certain if there were other similar unforeseen issues, we decided not to make the dataset publicly available.

**RQ2:** How are files on real-world, in-use Windows systems fragmented?

RQ2 invited a deep dive into understanding the patterns of file fragmentation in real-world, actively used Windows systems. Following the establishment of privacy-preserving data collection methods in RQ1, Chapter 3 presents a comprehensive analysis of the collected file fragmentation dataset [dataset-vdMee19].

The degree of fragmentation we found was 2.2% for all MFT entries, and 4.4% for files that could be fragmented (i.e., files with two or more blocks allocated). Specific file types or file extensions are more likely to be fragmented than others. For example, for files with two blocks or more, the observed fragmentation rates are: NTFS-compressed files (19.7%), sparse files (31.8%), `.jpg` files (3.1%), `.docx` files (6.2%), and `.pst` files (35.8%). Additionally, of all the fragmented files, nearly half (46.4%) are fragmented out-of-order.

**RQ3:** To what extent can file history be recovered from file timestamps?

RQ3 explores into the capabilities of reconstructing a file's history using the timestamps available within the file system. Chapter 4 showcases techniques for uncovering the past states and versions of files, leveraging timestamp information to piece together the chronological sequence of file modifications. This research introduces a temporal dimension to the existing realm of timestamp analysis and offers a new perspective on the lifecycle of digital files, enhancing ways for digital investigators to test, verify or refute working hypotheses.

Our conclusion is that potential file histories can be deduced from a current set of timestamps. However, there is an inherent loss of information during the reconstruction of such a timeline. This loss affects the number of potential file operations that can be accurately matched with a given set of timestamps. Ultimately, each timeline either concludes with a file creation event or remains open-ended. Additional sources of timestamp information for a specific file can impact the potential timelines in two ways: by limiting the number of branches (i.e., reducing the number of potential timelines) and by extending the remaining timelines (i.e., allowing us to trace further back in time for a given timeline).

**RQ4:** To what extent can entropy-coded JPEG data be validated?

RQ4 investigates the realm of JPEG file validation with the goal of fragmentation point detection. This investigation, detailed in Chapter 5, encompasses a comprehensive examination of existing techniques, and identifies and demonstrate a novel, deterministic approach for fragmentation point detection in JPEG files. This approach detects invalid Huffman codes and quantization array size overflows within the entropy-coded data section, a section which notoriously lacks file markers or other validation mechanisms. The algorithm's correctness is demonstrated. However, the practical effectiveness of the algorithm remained to be evaluated, as it depended on the actual structure and composition of JPEG files encountered in real-world scenarios. This uncertainty led to the formulation of RQ5.

**RQ5:** How effective are the newly identified approaches to JPEG fragmentation point detection in real-world JPEGs?

Building upon the findings of RQ4, RQ5 examines in detail the practical efficacy of the newly identified approaches for JPEG fragmentation point detection. Chapter 6 presents an empirical evaluation of these methods, applying them to real-world JPEG files (as collected in [dataset-vdMee22]) to assess their accuracy and reliability in a variety of scenarios. The results are remarkably convincing: the constructed JPEG validator [artefact-vdBvdM23] successfully identifies fragmentation points within a regular block size with an accuracy of over 99.4% in worst-case scenarios and exceeds 99.9% in average scenarios. This achievement effectively solves a longstanding challenge in the field of digital forensics.

With the discussion of the subsidiary research questions complete, we now turn to the central aim of this research, encapsulated in the primary research question:

*How to identify and leverage unexplored potential in file recovery?*

To address this, the investigation was approached from three perspectives, as outlined in the introduction:

- **File system** – The use of file systems and the extent of file fragmentation were extensively studied through a large-scale data collection on real-world, in-use laptops, using a privacy-respecting approach. This research highlighted, among others, the relationship between file fragmentation and volume space usage, differences in file fragmentation across types of storage devices, and the prevalence of specific NTFS file types.
- **File metadata** – Metadata has demonstrated significant value in research. By collecting and analyzing file allocation data and extensions, a comprehensive analysis of file fragmentation types was conducted. Furthermore, the study of timestamps—alongside file operations and their impact on these timestamps—revealed the ability to reconstruct possible file histories of individual files.
- **File format** – Successful file recovery, particularly of fragmented files, hinges on accurate fragment validation. This research not only demonstrates the feasibility of bit-level validation for JPEG files but also presents a JPEG validator that exhibits exceptional performance. This finding is particularly noteworthy because JPEG for the most part lacks internal mechanisms that would facilitate such validation.

Beyond the scientific findings, our approach to the central research question also embodies a specific research strategy. Drawing upon the foundation of scientific literature, insights from experts, and integrating student projects into our exploratory process, we not only enriched our research with fresh perspectives but also expanded our exploration into additional research avenues.

## 7.2 Future work and final thoughts

In the evolving domain of digital forensics, continuous research and development is crucial to keep pace with rapidly advancing technology and emerging challenges. The work presented in this thesis lays a foundation for several promising areas of exploration and advancement. The following paragraphs outline key directions for future research, expanding on the concepts and findings discussed throughout this thesis.

**File carver framework.** We intend to design and implement a modern file carver framework. This framework will accommodate both in-order and out-of-order fragmentation, a frequently occurring fragmentation pattern which was revealed in Chapter 3. This framework will integrate the JPEG validator introduced in Chapter 6.

Additionally, there are still unexplored and relevant areas where this file carver could bring improvements, particularly in the carving of sparse and NTFS-compressed files, areas that have seen limited research and development efforts to date. We have shown that NTFS-compressed files can be up to five times as likely fragmented compared to regular files, further increasing the need for a in- and out-of-order capable file carver.

**Expanding file format validation opportunities.** The algorithms presented in Chapters 5 and 6 allow recovery of fragmented high-entropy data for the JPEG file format. Given their impressive success rate, it would be worthwhile to investigate the applicability of bit-level validation as a fragmentation point detection mechanism for other high-entropy file formats.

**On a broader perspective.** With an updated perspective on file fragmentation in NTFS systems, there remains a gap in understanding concerning other file systems, such as those used in Linux (ext4) and Apple (APFS) environments, as well as embedded file systems in IoT devices. Increased knowledge in these areas could greatly benefit file fragmentation analysis and recovery techniques.

Regarding storage devices, Solid State Drives (SSDs) have altered the landscape of file recovery. The window of opportunity for successful file recovery on SSDs is significantly narrower compared to traditional hard disk drives (HDDs). While HDDs maintain their relevance in today's technological landscape, the prominence and future dominance of SSDs is inevitable. This shift is poised to transform the landscape of file carving until the next technological evolution takes place.

Lastly, the rise of Artificial Intelligence (AI) applications, including in the fields of Cyber Security and Digital Forensics, cannot be overlooked. While the use of AI must adhere to the same standards of integrity, transparency, and reproducibility as other methods for digital evidence, it is undeniable that AI will create new opportunities in the development of new file recovery techniques.

## Final thoughts

Reflecting on my PhD process, two moments particularly stand out for me. On two separate occasions during this PhD project, we encountered results that were so unexpected that we were convinced they could not be right. Consequently, we spent many hours scrutinizing our methodology and data, searching for errors. In both instances, the findings proved to be valid, marking the discovery of genuinely novel insights. Such moments are serendipitous and cannot be anticipated, which is what makes them so memorable.

Looking forward, the landscape of digital forensics is perpetually shaped by the advancements in technology and the ongoing digitalization of our society. Predicting where the next forensic breakthrough will occur is impossible. What remains certain is that discovering these breakthroughs requires both time and dedication. Researchers around the globe contribute to the field of digital forensics. Their collective work, encompassing everything from minor advancements to major innovations, not only refine existing methods but also establish the foundation for novel forensic approaches. All these research efforts either preserve or enhance our ability to uncover evidence and address legal questions about what might have happened at a crime scene. This thesis represents my personal contribution to this ongoing, collective pursuit.

# Bibliography

## Author's publications

[DFRWS-APAC21] Vincent van der Meer, Hugo Jonker, and Jeroen van den Bos. A contemporary investigation of NTFS file fragmentation. In *Forensic Science International: Digital Investigation*, 2021. DOI: 10.1016/j.fsidi.2021.301125. Special issue: Forensic Science International: Digital Investigation - Proceedings of the First Annual DFRWS APAC Conference 2021.

[DFRWS-EU24] Vincent van der Meer, Jeroen van den Bos, Hugo Jonker, and Laurent Dassen. Problem solved: a reliable, deterministic method for JPEG fragmentation point detection. In *Proceedings of the DFRWS EU 2024*. Elsevier, 2024. DOI: 10.1016/j.fsidi.2023.301687. Special issue: Forensic Science International: Digital Investigation - DFRWS EU 2024 - Selected Papers from the 11th Annual Digital Forensics Research Conference Europe.

[WIFS19] Vincent van der Meer, Hugo Jonker, Guy Dols, Harm van Beek, Jeroen van den Bos, and Marko van Eekelen. File fragmentation in the wild: a privacy-friendly approach. In *IEEE International Workshop on Information Forensics and Security, WIFS 2019, Delft, The Netherlands, December 9-12, 2019*, pages 1–6. IEEE, 2019. DOI: 10.1109/WIFS47025.2019.9034981.

[WSDF21] Vincent van der Meer and Jeroen van den Bos. JPEG file fragmentation point detection using huffman code and quantization array validation. In *ARES 2021: The 16th*

*International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021*, 14th International Workshop on Digital Forensics (WSDF 2021), 46:1–46:7. ACM, 2021. DOI: `10.1145/3465481.3470061`.

[WSDF23] Jelle Bouma, Hugo Jonker, Vincent van der Meer, and Eddy van den Aker. Reconstructing timelines: from NTFS timestamps to file histories. In *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023-1 September 2023*, 16th International Workshop on Digital Forensics (WSDF 2023), 154:1–154:9. ACM, 2023. DOI: `10.1145/3600160.3605027`.

## Artefacts

[artefact-Bou21] Jelle Bouma. Timestamp Analyser, version 1.0.1, 2021. URL: `https://github.com/JelleBouma/TimestampAnalyser`.

[artefact-Dol19] Guy Dols. PriFiWalk, 2019. URL: `https://github.com/GuyDols/PriFiwalk`.

[artefact-vdAke21] Eddy van den Aker. NTFS timestamp visualizer, 2021. URL: `https://github.com/EddyvdAker/NTFS-Timestamp-Visualizer`.

[artefact-vdBvdM23] Jeroen van den Bos and Vincent van der Meer. JPEG fragments, 2023. URL: `https://github.com/parsingdata/jpeg-fragments`.

## Datasets

[dataset-vdMee19] Vincent van der Meer. File Fragmentation Dataset. *Dataset is not published to protect privacy and prevent potential data breaches.* The dataset comprises file allocation and metadata for approximately 84 million files across 220 laptops. 2019.

[dataset-vdMee22] Vincent van der Meer. Collection of JPEG images from Wikipedia. *Dataset is not published due to uncertainties regarding the copyright status of individual images.* This dataset, containing approximately 230,000 JPEG images sourced from Wikipedia, was compiled for the purpose of internal validation of JPEG file recovery algorithms. 2022.

## Supervised student projects

[stud-Bor23] Nick Borgers. *Who touched this drive? Telling NTFS drivers apart based on what they think is NTFS.* Master's thesis, Open Universiteit, 2023. `https://research.ou.nl/en/studentTheses/`.

[stud-Bou19] Jelle Bouma. *Interpreting NTFS Timestamps.* Bachelor's thesis, Open Universiteit, 2019. Available upon request at the institution.

[stud-Das22] Laurent Dassen. *Constructing a JPEG-dataset.* Internship, Zuyd University of Applied Sciences, 2022. Available upon request at the institution.

[stud-Dol18] Guy Dols. *Fragmentation in the Wild.* Bachelor's thesis, Zuyd University of Applied Sciences, 2018. Available upon request at the institution.

[stud-Han20] Remco Hanegraaf. *On the (im)possibilities of recovering deleted files from SSDs.* Bachelor's thesis, Zuyd University of Applied Sciences, 2020. Available upon request at the institution.

[stud-Noo19] Robbert Noordzij. *Synthetic Fragmentation Experiments using WildFragSim.* Bachelor's thesis, Open Universiteit, 2019. Available upon request at the institution.

[stud-Pat19] Johannes Patti. *Visualizing File Storage on NTFS Volumes.* Bachelor's thesis, Zuyd University of Applied Sciences, 2019. Available upon request at the institution.

[stud-Pet21] Mart Peters. *Advanced File Format Validation for File Carving*. Master's thesis, Open Universiteit, 2021. `https://research.ou.nl/en/studentTheses/advanced-file-format-validation-for-file-carving-with-application`.

[stud-Smi23] Jasper Smitz. *Benchmark the file-carvers*. Internship, Zuyd University of Applied Sciences, 2023. Available upon request at the institution.

[stud-vDil21] Zowie van Dillen. *A Measured Evaluation of Artificial Filesystem Aging Tools*. Bachelor's thesis, Open Universiteit, 2021. Available upon request at the institution.

[stud-vKan19] Dewi van Kan. *Privacy Friendly Low Level Read Access to Android Phones*. Internship, Zuyd University of Applied Sciences, 2019. Available upon request at the institution.

[stud-Vol19] Yvonne Vollebregt. *File dating based on the Physical Location of the File*. Bachelor's thesis, Open Universiteit, 2019. Available upon request at the institution.

## Scholarly references

[AvdB11] Leon Aronson and Jeroen van den Bos. Towards an engineering approach to file carver construction. In *Proc. 35th Annual IEEE International Computer Software and Applications Conference (COMPSAC Workshops'11)*, pages 368–373, 2011. DOI: `10.1109/COMPSACW.2011.68`.

[AIM$^+$13] Nurul Azma Abdullah, Rosziati Ibrahim, Kamaruddin Malik Mohamad, and Norhamreeza Abdul Hamid. Carving linearly JPEG images using unique hex patterns (UHP). In *Proc. 1st Conference on Advanced Data and Information Engineering (DaEng'13)*, volume 285 of *Lecture Notes in Electrical Engineering*, pages 291–300. Springer, 2013. DOI: `10.1007/978-981-4585-18-7_33`.

[AIM13] Nurul Azma Abdullah, Rosziati Ibrahim, and Kamaruddin Malik Mohamad. Carving thumbnail/s and embedded JPEG files using image pattern matching. *Journal of Soft-*

*ware Engineering and Applications*, 6(3B):62, 2013. DOI: 10.4236/jsea.2013.63B014.

[AIR+22] Nor Amira Nor Azhan, Richard Adeyemi Ikuesan, Shukor Abd Razak, and Victor R Kebande. Error level analysis technique for identifying JPEG block unique signature for digital forensic analysis. *Electronics*, 11(9):1468, 2022. DOI: 10.3390/electronics11091468.

[AM21] Rabei Raad Ali and Kamaruddin Malik Mohamad. Rx_mykarve carving framework for reassembling complex fragmentations of JPEG images. *Journal of King Saud University-Computer and Information Sciences*, 33(1):21–32, 2021. DOI: 10.1016/j.jksuci.2018.12.007.

[BFV17] Brandon Birmingham, Reuben A. Farrugia, and Mark Vella. Using thumbnail affinity for fragmentation point detection of JPEG files. In *17th International Conference on Smart Technologies (IEEE EUROCON)*, pages 3–8. IEEE, 2017. DOI: 10.1109/EUROCON.2017.8011068.

[BJ19] Ahmed A. Bahjat and Jim Jones. Deleted file fragment dating by analysis of allocated neighbors. *Digital Investigation*, 28:S60–S67, 2019. DOI: 10.1016/j.diin.2019.01.015.

[BS04] Florian Buchholz and Eugene Spafford. On the role of file system metadata in digital forensics. *Digital Investigation*, 1(4):298–309, 2004. DOI: 10.1016/j.diin.2004.10.002.

[dBdS16] Johan de Bock and Patrick de Smet. Jpgcarve: an advanced tool for automated recovery of fragmented JPEG files. *IEEE Transactions on Information Forensics and Security*, 11(1):19–34, 2016. DOI: 10.1109/TIFS.2015.2475238.

[BYK+09] Jewan Bang, Byeongyeong Yoo, Jongsung Kim, and Sangjin Lee. Analysis of time information for digital investigation. In *Proc. 5th International Joint Conference on INC, IMS and IDC (NCM'09)*, pages 1858–1864. IEEE Computer Society, 2009. DOI: 10.1109/NCM.2009.258.

[BYL11] Jewan Bang, Byeongyeong Yoo, and Sangjin Lee. Analysis of changes in file time attributes with file manipulation.

*Digital Investigation*, 7(3-4):135–144, 2011. DOI: 10.1016/j.diin.2010.12.001.

[Car05] Brian Carrier. *File system forensic analysis*. Addison-Wesley Professional, 2005. ISBN: 978-0321268174. URL: https://dl.acm.org/doi/10.5555/1051914.

[CBJ+17] Alexander Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File systems fated for senescence? nonsense, says science! In *Proc. 15th USENIX Conference on File and Storage Technologies (FAST'17)*, pages 45–58, 2017. URL: https://dl.acm.org/doi/10.5555/3129633.3129639.

[Cho13] Gyu-Sang Cho. A computer forensic method for detecting timestamp forgery in NTFS. *Computers & Security*, 34:36–46, 2013. DOI: 10.1016/j.cose.2012.11.003.

[Cho14] Gyu-Sang Cho. An intuitive computer forensic method by timestamp changing patterns. In *Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS'14)*, pages 542–548. IEEE Computer Society, 2014. DOI: 10.1109/IMIS.2014.92.

[Cho16a] Gyu-Sang Cho. Data hiding in NTFS timestamps for anti-forensics. *International Journal of Internet, Broadcasting and Communication*, 8(3):31–40, 2016. DOI: 10.7236/IJIBC.2016.8.3.31.

[Cho16b] Gyu-Sang Cho. A new timestamp digital forensic method using a modified superincreasing sequence. *International Journal of Digital Crime and Forensics*, 8(3):11–33, 2016. URL: https://dl.acm.org/doi/10.5555/3273390.3273392.

[CKL+07] Kam-Pui Chow, Michael Y. K. Kwan, Frank Y. W. Law, and Pierre K. Y. Lai. The rules of time on NTFS file system. In *Second International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'07)*, pages 71–85. IEEE Computer Society, 2007. DOI: 10.1109/SADFE.2007.22.

[CO10] Neil J. Croft and Martin S. Olivier. Sequenced release of privacy-accurate information in a forensic investigation. *Digital Investigation*, 7(1-2):95–101, 2010. DOI: 10.1016/j.diin.2010.01.002.

[Coh07] Michael I. Cohen. Advanced carving techniques. *Digital Investigation*, 4(3-4):119–128, 2007. DOI: 10.1016/j.diin.2007.10.001.

[DC18] Fryderyk Darnowski and Andrzej Chojnacki. Writing and deleting files on hard drives with NTFS. *Computer Science and Mathematical Modelling*, 8:5–15, 2018. DOI: 10.5604/01.3001.0013.1457.

[DKM19] Emre Durmus, Pawel Korus, and Nasir D. Memon. Every shred helps: assembling evidence from orphaned JPEG fragments. *IEEE Transactions on Information Forensics and Security*, 14(9):2372–2386, 2019. DOI: 10.1109/TIFS.2019.2897912.

[DMT$^+$17] Emre Durmus, Manoranjan Mohanty, Samet Taspinar, Erkam Uzun, and Nasir D. Memon. Image carving with missing headers and missing fragments. In *2017 IEEE Workshop on Information Forensics and Security, WIFS*, pages 1–6. IEEE, 2017. DOI: 10.1109/WIFS.2017.8267665.

[DZ11] Xiaoqin Ding and Hengming Zou. Time based data forensic and cross-reference analysis. In *Proc. 2011 ACM Symposium on Applied Computing, Computer Forensics track (CF@SAC'11)*, pages 185–190. ACM, 2011. DOI: 10.1145/1982185.1982227.

[FA20] Tan Kin Fei and Nurul Azma Abdullah. Data carving linearly fragmented JPEG using file structured based technique. *Applied Information Technology And Computer Science*, 1(1):173–180, 2020.

[Gar07] Simson L. Garfinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4:2–12, 2007. DOI: 10.1016/j.diin.2007.06.017.

[Gar09] Simson L. Garfinkel. Automating disk forensic processing with sleuthkit, XML and python. In *Proc. 4th IEEE Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'09)*, pages 73–84, 2009. DOI: 10.1109/SADFE.2009.12.

[Gar10] Simson L. Garfinkel. Digital forensics research: the next 10 years. *Digital Investigation*, 7:S64–S73, 2010. DOI: 10.1016/J.DIIN.2010.05.009.

[GB10] Thomas Gloe and Rainer Böhme. The dresden image database for benchmarking digital image forensics. *Journal of Digital Forensic Practice*, 3(2-4):150–159, 2010. DOI: 10.1145/1774088.1774427.

[GJ17] Pavel Gladyshev and Joshua I. James. Decision-theoretic file carving. *Digital Investigation*, 22:46–61, 2017. DOI: 10.1016/j.diin.2017.08.001.

[GL21] Michael Galhuber and Robert Luh. Time for truth: forensic analysis of NTFS timestamps. In *16th International Conference on Availability, Reliability and Security (ARES'21)*, pages 1–10. ACM, 2021. DOI: 10.1145/3465481.3470016.

[GM15] Simson L. Garfinkel and Michael McCarrin. Hash-based carving: searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigations*, 14 Supplement 1:S95–S105, 2015. DOI: 10.1016/j.diin.2015.05.001.

[GSF+23] Mustafa Ghaleb, Kunwar Muhammed Saaim, Muhamad Felemban, Saleh Alsaleh, and Ahmad Almulhem. File fragment classification using light-weight convolutional neural networks. *arXiv preprint*, abs/2305.00656, 2023. DOI: 10.48550/arXiv.2305.00656.

[HKW18] Shuyuan Mary Ho, Da-Yu Kao, and Wen-Ying Wu. Following the breadcrumbs: timestamp pattern identification for cloud forensics. *Digital Investigations*, 24:79–94, 2018. DOI: `10.1016/j.diin.2017.12.001`.

[HLN+18] Graham Hudson, Alain Léger, Birger Niss, Istvan Sebestyén, and Jørgen Vaaben. JPEG-1 standard 25 years: past, present, and future reasons for a success. *Journal of Electronic Imaging*, 27(04):040901, 2018. DOI: `10.1117/1.JEI.27.4.040901`.

[HMK21] Hassan Jalil Hadi, Numan Musthaq, and Irshad Ullah Khan. SSD forensic: evidence generation and forensic research on solid state drives using trim analysis. In *Proc. 2nd International Conference on Cyber Warfare and Security (ICCWS'21)*, pages 51–56, 2021. DOI: `10.1109/ICCWS53234.2021.9702989`.

[JAH+16] Dae-il Jang, Gail-Joon Ahn, Hyunuk Hwang, and Kibom Kim. Understanding anti-forensic techniques with timestamp manipulation (invited paper). In *17th IEEE International Conference on Information Reuse and Integration (IRI'16)*, pages 609–614. IEEE Computer Society, 2016. DOI: `10.1109/IRI.2016.94`.

[JCH+18] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. File fragmentation in mobile devices: measurement, evaluation, and treatment. *IEEE Transactions on Mobile Computing*, 2018. DOI: `10.1109/TMC.2018.2869737`.

[JCS+16] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. An empirical study of file-system fragmentation in mobile storage systems. In *Proc. 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*, 2016. URL: `https://dl.acm.org/doi/10.5555/3026852.3026868`.

[KAD19] Martin Karresand, Stefan Axelsson, and Geir Olav Dyrkolbotn. Using NTFS cluster allocation behavior to find the location of user data. *Digital Investigation*, 29:S51–S60, 2019. DOI: `10.1016/j.diin.2019.04.018`.

[KAC15] Nur Fasihah Abdul Kadir, Shukor Abd Razak, and Hassan Chizari. Identification of fragmented JPEG files in the absence of file systems. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 1–6. IEEE, 2015. DOI: 10.1109/ICOS.2015.7377268.

[KS08] Martin Karresand and Nahid Shahmehri. Reassembly of fragmented JPEG images containing restart markers. In *2008 European Conference on Computer Network Defense*, pages 25–32. IEEE, 2008. DOI: 10.1109/EC2ND.2008.10.

[LCY$^+$11] Frank Y. W. Law, Patrick P. F. Chan, Siu-Ming Yiu, Kam-Pui Chow, Michael Y. K. Kwan, Hayson Tse, and Pierre K. Y. Lai. Protecting digital data privacy in computer forensic examination. In *Proc. 6th IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE 2011)*, pages 1–6, 2011. DOI: 10.1109/SADFE.2011.15.

[LFG06] Jan Lukás, Jessica J. Fridrich, and Miroslav Goljan. Digital camera identification from sensor pattern noise. *IEEE Transactions on Information Forensics and Security*, 1(2):205–214, 2006. DOI: 10.1109/TIFS.2006.873602.

[LSC$^+$11] Qiming Li, Bilgehan Sahin, Ee-Chien Chang, and Vrizlynn L. L. Thing. Content based JPEG fragmentation point detection. In *Proc. 12th IEEE International Conference on Multimedia and Expo (ICME)*, pages 1–6. IEEE, 2011. DOI: 10.1109/ICME.2011.6011883.

[MB12] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *TOS*, 7(4):14:1–14:20, 2012. DOI: 10.1145/2078861.2078864.

[MD09] Kamaruddin Malik Mohamad and Mustafa Mat Deris. Fragmentation point detection of JPEG images at dht using validator. In *First International Conference on Future Generation Information Technology (FGIT)*, pages 173–180. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-10509-8_20.

[MP06] Nasir D. Memon and Anindrabatha Pal. Automated reassembly of file fragmented images using greedy algorithms. *IEEE Transactions on Image Processing*, 15(2):385–393, 2006. DOI: `10.1109/TIP.2005.863054`.

[MRF19] Patrick Mullan, Christian Riess, and Felix Freiling. Forensic source identification using JPEG image headers: the case of smartphones. *Digital Investigation*, 28:S68–S76, 2019. DOI: `10.1016/j.diin.2019.01.016`.

[NA22] Rune Nordvik and Stefan Axelsson. It is about time–do exFAT implementations handle timestamps correctly? *Forensic Science International: Digital Investigation*, 42:301476, 2022. DOI: `10.1016/j.fsidi.2022.301476`.

[NLR13] Alastair Nisbet, Scott Lawrence, and Matthew Ruff. A forensic analysis and comparison of solid state drive data retention with trim enabled file systems. In *Proc. 11th Australian Digital Forensics Conference (ADFC'13)*, pages 1–10. SRI Security Research Institute, 2013. DOI: `10.4225/75/57b3d766fb873`.

[OLH21] Junghoon Oh, Sangjin Lee, and Hyunuk Hwang. NTFS data tracker: tracking file data history based on logfile. *Digital Investigations*, 39:301–309, 2021. DOI: `10.1016/j.fsidi.2021.301309`.

[Oli05] Martin S. Olivier. Forensics and privacy-enhancing technologies - logging and collecting evidence in flocks. In *Advances in Digital Forensics, IFIP International Conference on Digital Forensics, National Center for Forensic Science, 2005*, pages 17–31, 2005. DOI: `10.1007/0-387-31163-7_2`.

[PB20] David Palmbach and Frank Breitinger. Artifacts for detecting timestamp manipulation in NTFS on windows and their reliability. *Forensic Science International: Digital Investigation*, 32:300920, 2020. DOI: `10.1016/j.fsidi.2020.300920`.

[PM09] Anandabrata Pal and Nasir Memon. The evolution of file carving. *IEEE signal processing magazine*, 26(2):59–71, 2009. DOI: `10.1109/MSP.2008.931081`.

[PSM08] Anandabrata Pal, Husrev T Sencar, and Nasir Memon. Detecting file fragmentation point using sequential hypothesis testing. *Digital Investigation*, 5:S2–S13, 2008. DOI: 10.1016/j.diin.2008.05.015.

[PSS15] Raj Kumar Pahade, Bhupendra Singh, and Upasna Singh. A survey on multimedia file carving. *International Journal of Computer Science & Engineering Survey*, 6(6), 2015. DOI: 10.5121/ijcses.2015.6603.

[RG09] Vassil Roussev and Simson L. Garfinkel. File fragment classification-the case for specialized approaches. In *Proc. 4th IEEE Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'09)*, pages 3–14, 2009. DOI: 10.1109/SADFE.2009.21.

[RNP+17] Romi Fadillah Rahmat, Filbert Nicholas, Sarah Purnamawati, and Opim Salim Sitompul. File type identification of file fragments using longest common subsequence (LCS). *Journal of Physics: Conference Series*, 801(1):012054, 2017. DOI: 10.1088/1742-6596/801/1/012054.

[SM09] Husrev T Sencar and Nasir Memon. Identification and recovery of JPEG files with missing fragments. *Digital Investigation*, 6:S88–S98, 2009. DOI: 10.1016/j.diin.2009.06.007.

[SMC06] Bradley L. Schatz, George M. Mohay, and Andrew J. Clark. A correlation method for establishing provenance of timestamps in digital evidence. *Digital Investigations*, 3(Supplement):98–107, 2006. DOI: 10.1016/j.diin.2006.06.009.

[SZ12] Luigi Sportiello and Stefano Zanero. Context-based file block classification. In *Proc. 8th IFIP WG 11.9 International Conference on Digital Forensics*, volume 383 of *IFIP Advances in Information and Communication Technology*, pages 67–82. Springer, 2012. DOI: 10.1007/978-3-642-33962-2_5.

[TCR+17] Thanh Hai Thai, Rémi Cogranne, Florent Retraint, and Thi-Ngoc-Canh Doan. JPEG quantization step estimation and its applications to digital image forensics. *IEEE Trans-*

*actions on Information Forensics and Security*, 12(1):123–133, 2017. DOI: 10.1109/TIFS.2016.2604208.

[TFC+16] Yanbin Tang, Junbin Fang, KP Chow, SM Yiu, Jun Xu, Bo Feng, Qiong Li, and Qi Han. Recovery of heavily fragmented JPEG files. *Digital Investigation*, 18:S108–S117, 2016. DOI: 10.1016/j.diin.2016.04.016.

[TPB+16] Thein Than Tun, Blaine A. Price, Arosha K. Bandara, Yijun Yu, and Bashar Nuseibeh. Verifiable limited disclosure: reporting and handling digital evidence in police investigations. In *Proc. 24th IEEE Requirements Engineering Conference (RE'16)*, pages 102–105, 2016. DOI: 10.1109/REW.2016.032.

[US15] Erkam Uzun and Husrev Taha Sencar. Carving orphaned JPEG file fragments. *IEEE Transactions on Information Forensics and Security*, 10(8):1549–1563, 2015. DOI: 10.1109/TIFS.2015.2416685.

[VGG18] Robin Verma, Jayaprakash Govindaraj, and Gaurav Gupta. Df 2.0: designing an automated, privacy preserving, and efficient digital forensic framework. In *Proc. 13th ADFSL Conference on Digital Forensics, Security, and Law (CDFSL'18)*, pages 128–149. ADFSL, 2018.

[Wil08] Svein Yngvar Willassen. Hypothesis-based investigation of digital timestamps. In *4th Annual IFIP WG 11.9 Conference on Digital Forensics (Digital Forensics'08)*, volume 285 of *IFIP*, pages 75–86. Springer, 2008. DOI: 10.1007/978-0-387-84927-0_7.

[YPL+12] Byeongyeong Yoo, Jungheum Park, Sungsu Lim, Jewan Bang, and Sangjin Lee. A study on multimedia file carving method. *Multimedia Tools and Applications - MTA*, 61:1–19, 2012. DOI: 10.1007/s11042-010-0704-y.

[YT10] Hwei-Ming Ying and Vrizlynn L. L. Thing. A novel inequality-based fragmented file carving technique. In *Proc. 3rd International Conference on Forensics in Telecommunications (ICST'10)*, volume 56 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and*

*Telecommunications Engineering*, pages 28–39. Springer, 2010. DOI: 10.1007/978-3-642-23602-0_3.

[YXL$^+$17] Yitao Yang, Zheng Xu, Liying Liu, and Guozi Sun. A security carving approach for AVI video based on frame size and index. *Multimedia Tools Applications*, 76(3):3293–3312, 2017. DOI: 10.1007/s11042-016-3716-4.

## Web references

[web-Dev22] devConf. 2022. URL: https://devconf.nl/ (visited on 11/01/2023).

[web-Els24] Elsevier. CRediT author statement. 2024. URL: https://www.elsevier.com/researcher/author/policies-and-guidelines/credit-author-statement (visited on 05/01/2024).

[web-Foc20] Forensic Focus. Recovering evidence from ssd drives in 2014: understanding trim, garbage collection and exclusions. 2020. URL: https://www.forensicfocus.com/articles/recovering-evidence-from-ssd-drives-in-2014-understanding-trim-garbage-collection-and-exclusions (visited on 02/01/2020).

[web-For20] Various contributors. Digital forensics xml. 2020. URL: https://github.com/simsong/dfxml. (visited on 02/01/2020).

[web-Fut17] Futurum. Futurum: kennis en innovatiefestival. 2017. URL: https://web.archive.org/web/20171111135005/https://futurum.pro/ (visited on 11/11/2017).

[web-Has21] Calvin Hass. What is an optimized JPEG? 2021. URL: https://web.archive.org/web/20210211034949/https://www.impulseadventure.com/photo/optimized-jpeg.html (visited on 04/01/2021).

[web-McC19] John C. McCallum. Disk drive prices 1955+. 2019. URL: https://jcmit.net/diskprice.htm. (visited on 04/01/2019).

[web-MS09] Various authors. How NTFS works. Microsoft. 2009. URL: `https://learn.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc781134(v=ws.10)` (visited on 02/01/2024).

[web-MS20] Various authors. Defrag. Microsoft. 2020. URL: `https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/defrag` (visited on 02/01/2020).

[web-MS21] Microsoft. Default cluster size for NTFS, FAT, and exFAT. 2021. URL: `https://web.archive.org/web/20231205222859/https://support.microsoft.com/en-us/topic/default-cluster-size-for-ntfs-fat-and-exfat-9772e6f1-e31a-00d7-e18f-73169155af95` (visited on 04/01/2021).

[web-MS22] Various authors. File times. Microsoft. 2022. URL: `https://docs.microsoft.com/en-us/windows/win32/sysinfo/file-times` (visited on 12/01/2022).

[web-Net20] NetMarketShare. Operating system share by version. 2020. URL: `https://netmarketshare.com/operating-system-market-share.aspx?id=platformsDesktopVersions` (visited on 02/01/2020).

[web-NIO18] NIOC. NIOC congres 2018. 2018. URL: `http://www.nioc.nl/archief/2018/` (visited on 11/01/2023).

[web-OU23a] Open Universiteit. Learning and innovation in resilient systems. 2023. URL: `https://www.ou.nl/lirs` (visited on 11/01/2023).

[web-OU23b] Open Universiteit. PROMIS symposium. 2023. URL: `https://www.ou.nl/-/promis-symposium/` (visited on 11/01/2023).

[web-Sch22a] Joakim Schicht. Rawcopy. 2022. URL: `https://github.com/jschicht/RawCopy` (visited on 12/01/2022).

[web-Sch22b] Joakim Schicht. Mft2csv. 2022. URL: `https://github.com/jschicht/Mft2Csv` (visited on 12/01/2022).

[web-Wik19] Wikipedia contributors. List of filename extensions. Wikipedia, The Free Encyclopedia. 2019. URL: `https://en.wikipedia.org/wiki/List_of_filename_extensions` (visited on 04/01/2019).

[web-Wik23] Wikipedia contributors. File carving. Wikipedia, The Free Encyclopedia. 2023. URL: `https://en.wikipedia.org/wiki/File_carving` (visited on 11/01/2023).

## Technical standards and guidelines

[tech-II92] ITU-T and ISO/IEC. Digital Compression and Coding of Continuous-tone Still Images: Requirements and Guidelines, 1992. ITU-T Recommendation T.81 and ISO/IEC 10918-1.

[tech-KCG06] Karen Kent, Suzanne Chevalier, and Tim Grance. Guide to Integrating Forensic Techniques into Incident Response. Technical report, National Institute of Standards and Technology (NIST), 2006. DOI: `10.6028/NIST.SP.800-86`.

# List of figures

# List of tables

# List of algorithms

# List of abbreviations

| | |
|---|---|
| **AC** | Alternating Current (component in JPEG compression) |
| **AI** | Artificial Intelligence |
| **APFS** | Apple File System |
| **CRC** | Cyclic Redundancy Check |
| **DC** | Direct Current (component in JPEG compression) |
| **DCT** | Discrete Cosine Transform |
| **DFXML** | Digital Forensics XML |
| **DHT** | Define Huffman Table (JPEG marker) |
| **DQT** | Define Quantization Table (JPEG marker) |
| **EOI** | End of Image (JPEG marker) |
| **exFAT** | Extended File Allocation Table |
| **EXIF** | Exchangeable Image File Format |
| **ext4** | Fourth Extended File System |
| **FAT** | File Allocation Table |
| **FN** | File Name (MFT attribute) |
| **GDPR** | General Data Protection Regulation |
| **HDD** | Hard Disk Drive |
| **IoT** | Internet of Things |
| **JFIF** | JPEG File Interchange Format |
| **JPEG** | Joint Photographic Experts Group |
| **MCU** | Minimal Coded Unit |
| **MFT** | Master File Table |

| | |
|---|---|
| **NTFS** | New Technology File System |
| **NTP** | Network Time Protocol |
| **OoO** | Out-of-Order (as used for out-of-order fragmentation) |
| **OS** | Operating System |
| **PII** | Personally Identifiable Information |
| **PST** | Personal Storage Table (Outlook data file) |
| **QA** | Quantization Array |
| **RGB** | Red, Green, Blue color space |
| **RQ** | Research Question |
| **RST** | Restart marker (JPEG marker) |
| **SD** | Secure Digital |
| **SI** | Standard Information (MFT attribute) |
| **SOF0** | Start of Frame (Baseline DCT, JPEG marker) |
| **SOF2** | Start of Frame (Progressive DCT, JPEG marker) |
| **SOS** | Start of Scan (JPEG marker) |
| **SSD** | Solid State Drive |
| **TRIM** | Command used to inform an SSD which data blocks can be erased. |
| **UFS** | Unix File System |
| **USB** | Universal Serial Bus |
| **YCbCr** | Color space, used in JPEG files.<br>Y is luminance, while Cb and Cr are chrominance components. |

# Summary

The recovery of deleted computer files is an important part of digital forensic investigations. The process of recovering computer files from a storage medium (such as a hard disk, USB-drive, or memory card) in the absence of a file's metadata is also called *file carving*. This thesis improves the foundations of file recovery. It does so by expanding our knowledge of factors that affect the success of file recovery efforts and by designing, implementing, and validating relevant artifacts that demonstrate their feasibility and effectiveness.

The challenges in file recovery are manifold. They are influenced by the ever-increasing amount of data on storage media. The diversity of digital sources in our everyday lives also plays a significant role, alongside the emergence of new storage media such as solid-state drives (SSDs), the application of cryptography, and the use of cloud storage. Of particular importance is the challenge posed by file fragmentation, where files are spread across multiple fragments on different locations on the storage medium.

Prior to this work, the most recent survey measuring file fragmentation was severely outdated, leaving a gap in the current data for the field of digital forensics that this dissertation seeks to address. Consequently, this work begins with the design and development of an artifact to measure file fragmentation on real-world, actively used computers. Given the significant role of computers (including laptops) in our lives, which contain vast amounts of personal information, examining the contents of a storage device must be performed in a privacy-friendly manner. Our artifact was therefore developed using a privacy-by-design approach. We used this tool to gather data from over 200 laptops belonging to students from Zuyd University of Applied Sciences. This data collection not only demonstrated the functionality of our methodology but also resulted in the largest dataset on file fragmentation since 2007.

This data collection revealed that, although the average rate of fragmentation has decreased, largely due to the automated scheduled defragmentation processes,

the sheer volume of fragmented data has risen, a trend tied to the increasing capacity of hard drives. Remarkably, it also uncovered that nearly half of all fragmented files exhibit out-of-order fragmentation, a form of file fragmentation not accounted for in current file carving tools.

The collected dataset also enabled a detailed study of file timestamps, including their manipulation and the impact of file operations on these timestamps. By examining the state of a file prior to specific operations (effectively, analyzing the reverse effect of file operations on timestamps) we were able to reconstruct potential file histories. This methodology, along with its visualization, was showcased and automated through the development of two artifacts.

The dissertation then focuses on the JPEG file format, which is forensically the most prevalent type of visual evidence. Due to the lack of effective algorithms for identifying fragmentation points in JPEG files, recovering these files presents a considerable challenge. Following an in-depth analysis of the JPEG decoding process, we developed a validation algorithm for JPEG files. Distinguishing itself from many existing approaches, this algorithm operates on deterministic principles, guaranteeing consistent results with identical inputs. In forensic investigations, the reproducibility of findings is an important requirement, and our algorithm meets this standard by design.

We have implemented this algorithm with broad support for all currently utilized JPEG file format variations. To evaluate its performance, particularly in detecting fragmentation points within the 'entropy-coded data' sections of JPEG files, we compiled a substantial dataset of JPEG files encountered 'in the wild'. This algorithm underwent rigorous and extensive testing, applying it to a wide array of JPEG files that included both baseline and progressive formats.

The results are exceptionally convincing: in *average-case* scenarios, the fragmentation point was detected in 99.997% of cases (for baseline encoded JPEGs) within 4 kilobytes (the most common block size of NTFS). Even under the most challenging conditions, a detection rate of 99.4% was achieved. These outcomes underscore the effectiveness of the algorithm, leading us to conclude that the longstanding problem of JPEG fragmentation point detection has been effectively solved.

Looking forward, with in-depth knowledge of file fragmentation patterns and a proven functioning JPEG validator, we intend to design and implement a file carver framework that accommodates both in-order and out-of-order fragmentation. This will enhance the ability for forensic researchers to recover deleted (photographic) evidence.

# Samenvatting

Het herstellen van verwijderde computerbestanden is een belangrijk onderdeel van digitaal forensisch onderzoek. Het proces waarbij computerbestanden worden hersteld van een opslagmedium (zoals een harde schijf, USB-stick of geheugenkaart) zonder de bijbehorende metadata, wordt ook wel *file carving* genoemd. Dit proefschrift heeft als doel de fundamenten van bestandsherstel te verbeteren. Dat gebeurt door onze kennis over de factoren die het succes van bestandsherstelinspanningen beïnvloeden te vergroten, en door het ontwerpen, implementeren en valideren van relevante artefacten die hun haalbaarheid en effectiviteit aantonen.

Er zijn veel factoren die het herstel van verwijderde bestanden compliceren, zoals de steeds groter wordende opslagcapaciteit van apparaten, en de toenemende hoeveelheid locaties waarop gegevens worden opgeslagen. Daarnaast verliest de klassieke harde schijf steeds meer terrein aan nieuwe opslagmedia zoals solid-state drives (SSD's), en ook het gebruik van cryptografie en cloudopslag vormen extra barrières bij bestandsherstel. Een bijzonder complicerende factor voor bestandsherstel is bestandsfragmentatie, waarbij bestanden opgesplitst worden in meerdere fragmenten die op verschillende locaties op het opslagmedium terechtkomen.

Voorafgaand aan dit onderzoek was de meest recente studie naar bestandsfragmentatie al sterk verouderd, waardoor er een gebrek aan actuele gegevens ontstaan was voor het digitaal forensisch vakgebied. Daarom begint dit proefschrift met het ontwerp en de ontwikkeling van een artefact om bestandsfragmentatie te meten op computers die actief in gebruik zijn. Gezien de belangrijke rol van computers (waaronder laptops) in ons leven, die enorme hoeveelheden persoonlijke informatie bevatten, moet het onderzoeken van de inhoud van een opslagmedium op een privacyvriendelijke manier worden uitgevoerd. Ons artefact is daarom ontwikkeld met het privacy-by-design principe. Dit artefact hebben we gebruikt om gegevens te verzamelen van meer dan 200 laptops van studenten van Zuyd Hogeschool. Deze gegevensverzameling toonde niet alleen de werking van onze methodologie en het artefact, maar resulteerde ook in de grootste dataset over

bestandsfragmentatie sinds 2007.

Deze gegevensverzameling laat zien dat, hoewel de gemiddelde fragmentatiegraad is gedaald (grotendeels door de geautomatiseerde defragmentatieprocessen), de totale hoeveelheid gefragmenteerde data is gestegen. Dit is een trend die gekoppeld is aan de toenemende capaciteit van harde schijven. Opvallend genoeg bleek bijna de helft van alle gefragmenteerde bestanden out-of-order gefragmenteerd te zijn, een patroon van bestandsfragmentatie dat niet wordt geadresseerd door huidige file carving tools.

De verzamelde dataset maakt het ook mogelijk om timestamps gedetailleerd te bestuderen, inclusief hun aanpassing en het effect van bestandsoperaties op deze timestamps. Door de toestand van een bestand te onderzoeken voor specifieke bestandsoperaties en de inverse effecten van deze operaties op timestamps te analyseren, konden we potentiële bestandsgeschiedenissen reconstrueren. Deze methodologie, inclusief de visualisatie ervan, is gedemonstreerd en geautomatiseerd met de ontwikkeling van een tweetal artefacten.

Het proefschrift richt zich vervolgens op het JPEG-bestandsformaat, dat forensisch gezien het meest relevante bestandsformaat voor foto's is. Door het gebrek aan effectieve algoritmen voor het identificeren van fragmentatiepunten in JPEG-bestanden, vormt het herstellen van deze bestanden een significante uitdaging. Na een diepgaande analyse van het JPEG-decoderingsproces hebben we een validatiealgoritme voor JPEG-bestanden ontwikkeld. Dit algoritme onderscheidt zich van veel bestaande benaderingen doordat het werkt op deterministische principes, wat garandeert dat het consequent identieke resultaten oplevert bij identieke invoer. In forensisch onderzoek is de reproduceerbaarheid van bevindingen een belangrijke eis, en ons algoritme voldoet aan deze standaard.

We hebben dit algoritme geïmplementeerd met brede ondersteuning voor alle momenteel gebruikte JPEG-bestandsformaatvariaties. Om de prestaties te evalueren, met name bij het detecteren van fragmentatiepunten binnen de *entropy-coded data*-secties van JPEG-bestanden, hebben we een aanzienlijke dataset van JPEG-bestanden samengesteld die 'in het wild' zijn aangetroffen. Dit algoritme heeft een rigoureuze en uitgebreide test ondergaan, waarbij het is toegepast op een breed scala aan JPEG-bestanden, met zowel baseline- als progressive formaten.

De resultaten zijn bijzonder overtuigend: in *average-case* scenario's werd het fragmentatiepunt gedetecteerd in 99,997% van de gevallen (voor baseline JPEG's) binnen 4 kilobytes (de meest voorkomende blokgrootte van NTFS). Zelfs onder de meest moeilijke omstandigheden werd een detectiepercentage van 99,4% bereikt. Deze resultaten bewijzen niet alleen de effectiviteit van het algoritme, maar leiden ons ook tot de conclusie dat het lang bestaande probleem van het detecteren van

JPEG-fragmentatiepunten is opgelost.

Met het oog op de toekomst willen we, met de kennis van fragmentatiepatronen en een bewezen effectieve JPEG-validator, een framework voor file carving ontwerpen en implementeren dat zowel in-order als out-of-order fragmentatie ondersteunt. Dit zal de mogelijkheden van forensisch onderzoekers om verwijderde foto's te herstellen aanzienlijk verbeteren.

# About the author

Vincent van der Meer was born on January 8th 1982 in Delft, The Netherlands. He completed his pre-university education (VWO) in 2000 at Thomas More College in The Hague.

After enrolling in the Computer Science program at Delft University of Technology in 2000, Vincent began his career in the IT sector. He first worked as a functional application manager at OGD. In 2005, he transitioned to TOPdesk, where he served as a software engineer until 2007. That same year Vincent earned his MSc degree in Computer Science from TU Delft, with a specialization in Cryptography.

In late 2007, Vincent started a new phase in his professional career at APG, initially joining as an Information Analyst. His contributions to the company led to a promotion to a senior role in 2010, and by 2012, he had risen to the position of Business & Information Architect. Parallel to his career at APG, 2012 also saw Vincent expanding his academic involvement, taking on the role of Senior Lecturer at Zuyd University of Applied Sciences

In 2016, Vincent started in a PhD position at the Data Intelligence research center of Zuyd University of Applied Sciences. This step was further solidified in the following year when he became an external PhD candidate in the Computer Science department at the Open Universiteit. In 2017, Vincent was awarded the NWO Doctoral Grant for Teachers, recognizing and supporting his research potential.

Over the course of seven years, Vincent balanced his teaching responsibilities (focusing on Software Engineering, Enterprise Architecture, and Testing) with his academic research. As part of his PhD research, he guided six students to their BSc degrees and two to their MSc degrees in Computer Science. Additionally, he engaged in public outreach, presenting at various academic and industry conferences, and was keynote speaker at two of these industry events.

Vincent's research endeavors resulted in the publication of five peer-reviewed papers, of which he was the first author for four. These contributions have been published by IEEE, ACM, and Elsevier. An extraordinary accomplishment has been that two out of his five papers were honored with Best Paper Awards. His work on reconstructing timelines from timestamp information received the *WSDF @ ARES Best Research Paper Award*. His study that solved a long-standing open question in Digital Forensics, regarding JPEG validation, was awarded the *DFRWS EU 2024 Best Paper Award*. Lastly, Vincent became a program committee member for the International Workshop on Digital Forensics (WSDF 2024).

On a personal level, Vincent served as the secretary of the participation council (medezeggenschapsraad) at his children's school. Beyond this community involvement, he also enjoys hiking and top-rope climbing.