Doctoral theses at NTNU, 2022:43

Kyle Porter

Approximate String Matching and Filesystem Metadata Carving

A Study of Improving Precision and Recall for Assisting the Digital Forensics Backlog

Doctoral thesis

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

NTNU
Norwegian University of Science and Technology

Kyle Porter

Approximate String Matching and Filesystem Metadata Carving

A Study of Improving Precision and Recall
for Assisting the Digital Forensics Backlog

Thesis for the Degree of Philosophiae Doctor

Gjøvik, February 2022

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

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

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

© Kyle Porter

ISBN 978-82-326-5619-6 (printed ver.)
ISBN 978-82-326-6396-5 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)

Doctoral theses at NTNU, 2022:43

Printed by NTNU Grafisk senter

Life can only be understood backwards; but it must be lived forwards.

Søren Kierkegaard

Abstract

The technical aspects of digital forensics are often dependent upon the progress made in other scientific fields. One such area of study whose results are often applied to digital forensics is string matching theory. String matching is found in most forensic tools, and even plays a significant role in file carving. Despite being ubiquitous within digital forensics, the study of string matching and its theory within the context of digital forensics is relatively rare. The goal of this dissertation was to develop and implement string matching algorithms that attempt to produce more optimal precision and recall results for approximate string matching and filesystem metadata record carving. Benefits of improving recall include providing the investigator with more of the relevant contents of an examination, and benefits of improving precision will permit the investigator to view less false positive hits. By developing algorithms which improve or balance precision and recall, we can increase the efficiency of digital forensics examinations, and thereby assist in reducing the digital forensics backlog.

To improve approximate string matching, we developed a novel method of implementing the constrained edit distance (CED) metric into a nondeterministic finite automaton (NFA), thus allowing for a novel approximate string matching algorithm that has significant control over its approximation parameters. The traditional NFA for approximate string matching will only match keywords up to some k number of errors, but our novel NFA allows for approximate matching for every possible combination of edit operation errors (character insertions, deletions, and substitutions) up to some edit distance threshold $k = 2$, Elasticsearch's maximum allowed distance. This provides more approximation parameters available to the user, where there is the possibility that some previously unreachable

parameters can be applied to improve the precision of the search results without significantly and negatively affecting the recall of the results, or vice versa. For example, in applying the algorithmic implementation of our novel NFA on a list of terms derived from an inverted index of the Enron email dataset, we obtained new precision-recall results that maximized recall without decimating the precision.

Carving is also prone to frustrating precision-recall trade-offs. Usually, carving is associated with file carving, but we specifically carve for filesystem metadata records. This is useful in scenarios in which a filesystem is somehow made unusable, by either physical or logical damage to a volume. Furthermore, finding a record may allow for recovery of the file content whether or not a file is fragmented, and can may also allow for the recovery of files without any signature at all. Research into filesystem metadata record carving is niche, and only a few scientific studies have evaluated such methods. Of which, only the Ext4 and NTFS filesystems have been deeply studied, and while the recall of recovering inodes may potentially be high for these specific filesystems, it would appear that the precision is low.

Our novel filesystem metadata record carving algorithm searches for *dynamic signatures* within records, where the signatures are byte patterns that are characteristic of filesystem timestamps in metadata records for most filesystems. Timestamps within records are typically stored in a closely co-located manner, and oftentimes two or more of the timestamps exactly match. This characteristic can be used as a starting point for verifying if the surrounding data is indeed a metadata record. After carving for potential timestamps we run another script for the purpose of metadata record verification. We refer to the process of timestamp carving followed by filesystem specific parser verification as *Generic Metadata Time Carving* (GMTC), and the intention is that the method should be applicable to most filesystems even if the filesystem metadata records contain no static signature.

Results from our experiments have shown that this method of filesystem metadata carving has high precision and recall when searching for records with two or more equal timestamps for at least carving Master File Table (MFT) records from NTFS and inodes from Ext4. In fact, our tools successfully carved records from Ext4 and NTFS that commercial tools could not. We later developed a novel approximate matching potential timestamp carver that matches timestamps by their byte prefixes. This significantly increased the recall of our filesystem metadata carving methodology as stringologically similar timestamps should also be temporally similar. Surprisingly, results from these experiments show that we had 100% precision for all of our tests on realistic datasets due to the strictness of our verification parsers. By setting lower values of the timestamp prefix length, recall was significantly improved while also producing many more false positive potential timestamp locations and thus longer runtimes. When applying prefix-

based potential timestamp carving on a 476 GB disk image, the total time taken to conduct the GMTC method was between two and half, and three hours.

Acknowledgments

This work could not be completed without the following people.

Thank you Slobodan Petrovic for being a great supervisor, allowing me to have the opportunity to work on this PhD, and introducing me to interesting concepts such as LFSRs and string matching algorithms. Thank you Katrin Franke for being a great co-supervisor, and providing me with many opportunities to work on relevant topics and meet interesting people. Thank you both for your advice.

I would like to thank my wife Parisa Rezaee Borj for being a constant source of love, support, comfort, and hilarity. I also need to thank the rest of my family for their patience and support while I have been away all of these years studying in a far away country. Special thanks go to the RanDairs for their technical advice.

I would also like to thank my friends for their friendship and solidarity. In particular, thank you Kyle Sittig for your shared PhD experiences and lifelong friendship, thank you Jan William Johnson for the fun times and assistance, thank you Jonathan Ness for your constant kindness, and thank you Ctirad Sousedik for your honesty.

Lastly, I would like to thank my colleagues for their work and motivation. In particular, thank you Rune Nordvik for introducing me to filesystems and all of your work during our research, thank you Fergus Toolan and Stefan Axelsson for the advice on our research papers, thank you Stig Andersen for the motivation and interesting conversations, thank you Ivar Moen for the fun times and project opportunities. And thank you Jon Hardeberg for sparking the opportunity to for me to study in Norway.

Contents

List of Tables

List of Figures

Acronyms

APFS	Apple File System.
CED	Constrained edit distance.
cedas	Constrained edit distance approximate search (Algorithm).
CFTT	Computer Forensics Tool Testing Program.
CPU	Central Processing Unit.
DAWG	Directed Acyclic Word Graph.
DECA	Decision-theoretic carving.
DFA	Deterministic finite automaton.
DFRWS	Digital Forensics Research Workshop.
DHS	Department of Homeland Security.
DRbSI	Data Reduction by Selective Imaging.
Ext4	Extended File System 4.
ExtX	Extended File System 2, 3, or 4.
FBI	Federal Bureau of Investigation.
FNA	Filename Attribute.

FTK	Forensic Toolkit.
GMTC	Generic Metadata Time Carving.
GPU	Graphics Processing Unit.
GUID	Globally Unique Identifier.
LFSR	Linear Feedback Shift Register.
MAC	Timestamps of the latest modification (mtime) or last written time, access (atime) or change (ctime) of a certain file [69].
MBR	Master Boot Record.
MFT	Master File Table.
NFA	Nondeterministic finite automaton.
NIST	National Institute of Standards and Technology.
NLP	Natural Language Processing.
NTFS	New Technology File System.
NTLK	Natural Language Toolkit.
OS	Operating System.
RAM	Random Access Memory.
SIA	Standard Information Attribute.
VM	Virtual Machine.

List of Algorithms

Part I

Overview

Chapter 1

Introduction

1.1 Motivation

Digital forensics is a field of study that is dependent upon the technological advances of other fields. A particular technology that has been with digital forensics since the beginning is string matching. Keyword search, regular expression matching, and carving are all reliant upon string matching theory and technology. Despite the importance of string matching algorithms in the field of digital forensics, past research on the subject is relatively scant. This is surprising, as faster and more accurate searches means more efficient forensic examinations.

A seemingly perennial problem afflicting digital forensics and investigations is the massive amount of data that investigators are tasked with acquiring, examining, and analyzing for relevant and admissible evidence. This modern paradigm is the root of the cause of the so called "digital forensics backlog", which may cause hasty and erroneous forensic analysis and can significantly delay criminal cases [37]. Causes of the backlog are multi-fold, including the general adoption of digital devices by the public (including smart phones, tablets, wearable devices, etc.) [118], increasingly large storage on devices [112], increasing use of cloud storage [176], and the increasingly large variety and differing data needing to be processed [87]. In 2010, Garfinkel stated that "The Golden Age of computer forensics is coming to an end" [75], in that the standard digital forensics methods would not keep up with the increasing volume and complexity of digital technology. Eleven years later, it is clear that we have not overcome the challenges set out by Garfinkel. However, we are not in a dark age either. There has been a plethora of research contributing methods and technical solutions for overcoming the digital forensics backlog, especially the issue of data volume. One of the most direct ways to manage this problem is create or implement faster processing methods [17, 127, 196]. Other methods proposed wish to exploit automation and machine learning [20, 71, 87]. Another approach has been to utilize methods which provide more relevant information to the user, by way of controlling false positive or false negative results [25]. Our research has pursued the latter approach. Two string matching applications that are significantly affected by false positives and negatives are approximate string matching [41] and carving [55, 74, 113, 163].

Past research which has focused on controlling the false positive rates of approximate string matching expand on the classic Wu-Manber [209] and Baeza-Yates [13] algorithms by controlling the allowed types of character errors [40, 41, 43, 42, 97]. This dissertation extends this past research by providing a method such that a greater degree of control over the possible character errors in approximate matches, thereby having more fine-grained control over the number of false positive matches. Likewise, this dissertation also examines how to reduce the number of false positive matches when applying digital forensics carving, and specifically carving for filesystem metadata content. Past research focused on retrieving content from specific filesystems [55, 163, 128], whereas we developed a more general method that can ideally be applied to most filesystems, thereby reducing false negatives.

1.2 Research Topic

The topic of our research is to develop new string matching algorithms to improve precision-recall trade-offs for approximate string matching and filesystem metadata carving. Investigators benefit from improved recall since it provides them with more relevant content when examining some data source, and they benefit from improved precision as it will reduce the number of false positive hits that the investigator would have to manually review, thereby increasing their efficiency. The methods and algorithms that we have developed for this dissertation are actually quite generic, but our application space is digital forensics and investigations.

Exact string matching in digital forensics already has to cope with a cascade of false positive hits, where Beebe and Liu noted that 95% or more of the returned results for a literal string search are irrelevant [25]. The number of false positive search results for approximate string matching is, naturally, even greater. An advantage of approximate string matching over exact matching is that we can match for either misspellings or some derivatives of keywords. However, along with finding variations of keywords, we also find a larger number of false positive string matching hits. Since the number of false positive matches may become so severe, the search engine Elasticsearch sets the maximum edit distance of their approximate search functionality to 2 [61], therefore allowing only up to two character errors in keyword search.

To gain a deeper understanding of false positives and potential solutions for approximate string matching, it is necessary to introduce some string matching theory. Approximate string matching is defined as searching for a sequence of bytes or characters of length m within some larger sequence T, where matches of the pattern are considered to occur when a subsequence of bytes within T has some distance k or less from the search pattern, wherein the distance is defined by a particular distance metric. A common distance metric used for approximate string matching is the edit distance (also known as the Levenshtein distance) metric [115]. The edit distance metric measures the minimum number of character insertions, deletions, and substitutions to transform one string into another string. So an approximate string matching algorithm using the edit distance accepts some subsequence S of T as a match when S is less than or equal to some k edit distance from the search pattern. Due to the common occurrences of false positives even at relatively low edit distance values, we investigate how to utilize the *constrained edit distance* (CED) [152, 154] within approximate string matching algorithms. The CED places constraints on how edit operations can be applied when calculating the edit distance. So for example, constraints may be the allowance of a particular number of edit insertions or only allowing for some number of maximum consecutive deletions. If the user of an approximate string matching algorithm has prior knowledge of the types of errors to expect, then they can possibly decrease the number of false positive hits they will encounter [41, 43].

The usual way of designing approximate string matching algorithms is by utilizing automata theory, where directed graphs called *finite automata* may be used as models for string matching algorithms and have the ability to provide extremely fast processing [142]. The most seminal approximate string matching algorithms by Baeza-Yates [13] and Wu and Manber [209] simulate nondeterministic finite automata (NFA) using an edit distance threshold, and we take a similar approach. Our research has investigated how to implement the constrained edit distance metric into NFA for approximate string matching algorithms, allowing for very general edit operation constraints. This lets us explore the precision-recall trade-offs more fully between edit distances $k = 1$ and $k = 2$, which is useful when an edit distance search of $k = 1$ may yield too little recall and an edit distance search of $k = 2$ may yield too little precision. The caveat is that the user still needs to identify which edit constraint parameters to use which may potentially provide beneficial precision-recall trade-offs.

In the context of digital forensics, it would appear that approximate string matching has become less popular in recent times. For instance, National Institute of

Standards and Technology's (NIST) federated testing for string searching functionality included approximate string matching as a test category in 2008 [144], but more recently updates to the tests do not include approximate string matching at all [145]. However, many search engines and e-discovery software include approximate string matching functionality [58, 179].

Our second primary interest for applying string matching to digital forensics is carving. Different carving algorithms use exact string matching algorithms [181], multi-pattern string matching algorithms [17, 212], and regular expression matching algorithms [76, 196]. While carving usually refers to file carving, in which file header and footer signatures are searched for and the content in between is extracted [74], we use the phrase more generally to also entail filesystem metadata carving [55] or feature carving (email addresses, telephone numbers, social security numbers, etc.) [76]. We take a particular interest in improving the precision and recall of file system metadata record carving.

File carving is a standard tool in the investigator's toolkit for extracting files from unallocated areas of disk, but it has several challenges. A current issue in file carving is that in order to obtain full recall when finding the header or footer signatures of a file, one must carve for many signatures. Multipattern searches certainly make this process faster [17, 212], but files will at times have unexpected signatures or no signature at all (such as .txt files), and thus the search will miss the file. This will limit the recall of files extracted from a disk image. Another limiting factor on recall for file carving is the ability to correctly handle fragmented files. Some carving techniques assume that files are made up of one continuous run of data, and thus will fail to successfully carve fragmented files [74]. A study by Laurenson [113] that the carving recall for multiple commercial tools on the dfrws-2006-challenge.img[1] was on average roughly 55%, where the precision was on average about 54%. Furthermore, file carving does not connect files to their filesystem metadata record entry (such as Master File Table (MFT) records or inodes), or even the name of the file [38].

Hamm [92] has suggested that carving for filesystem metadata may be more effective for searching for files with elusive signatures. The secondary benefit is that carving for filesystem metadata records allows for the possibility of connecting file metadata to the actual file content. As all once allocated files on a disk image have at one point had an associated filesystem metadata record, then clearly searching for records is an effective way of finding file content as well. Dewald and Seufert [55] have shown that in cases where the filesystem has been rendered unusable that it is possible to carve for filesystem metadata records that do not con-

[1]http://old.dfrws.org/2006/challenge/

tain static signatures, and as the records may content information regarding a file's content location full file recovery is possible. Their method of carving involved searching for kinds of *dynamic signatures*, where as opposed to static signatures, the byte patterns may change but are characteristic of a kind of datastructure. Examples of dynamic signatures may be plausible timestamp datetime ranges, or that byte sequences in close proximity to each other have particular relationships, etc. The results of applying their methods to a 16 GB Ubuntu image showed that their method had a false negative rate between 0.003% and 47.6% depending on the signatures used. They did not provide precision or recall statistics, but it would appear their method has decently high recall, but often low precision. The low precision is indicated by the fact they found millions of potential records outside the expected areas of records. While there are a number of particular open source tools for filesystem metadata record recovery (mostly from Joakim Schicht [188][2]), relatively little research has been done on the subject, where only carving records from Ext4 [55] and APFS [163] seem to have been investigated scientifically.

Our filesystem metadata carving research has focused on developing a generic approach for discovering filesystem metadata records. The novel dynamic signature we search for is the general pattern of timestamps found in filesystem metadata records. Our observations were that records from many different filesystems store their timestamps in a way that they are closely co-located to each other, and that oftentimes there are duplicate timestamps within a single record. By searching for short byte sequences with this characteristic, we can identify the position of potential timestamps on a disk image. As this methodology generates many false positive timestamp locations as well, we developed a second step to validate if the locations given by the potential timestamps were actually a filesystem metadata record from a specific filesystem. Thus, this method allows for possible file recovery including its metadata content, can manage file fragmentation if the metadata has not been erased, and has minimal reliance on the filesystem itself (especially in the cases of a damaged disk image). Our research assesses the precision and recall for carving for filesystem metadata records using our methods, how well it works across different filesystems, and ways that precision or recall may be improved.

1.3 Research Questions

Bearing in mind the previous discussion on the research topic, we formulated the following research questions, which we used as a guide for this dissertation.

1. How can we improve approximate string matching algorithms with respect to precision, recall, or accuracy specifically in the context of digital forensics?

[2]https://github.com/jschicht?tab=repositories

In particular, how can the constrained edit distance be implemented into nondeterministic finite automata where it allows for general edit operation constraints?

2. As string matching is a component of carving, how can we apply exact or approximate string matching to carving such that we can improve the precision or recall for file or metadata recovery?

    (a) How can filesystem metadata carving be improved?

    (b) Which filesystems can we make an improvement for?

    (c) Assuming a filesystem is significantly damaged and the filesystem does not store a file's filename metadata inside its associated filesystem metadata record, how can we connect the filename metadata to its filesystem metadata record?

1.4 Publications

The following publications contribute to the dissertation, and make up the chapters in Part II of the dissertation. Papers 1 and 2 address Research Question 1, Paper 3 addresses Research Question 2 and all its subquestions, and Paper 4 addresses Research Question 2(a).

1. Porter, K., & Petrovic, S. (2017). On Application of Constrained Edit Distance Algorithms to Cryptanalysis and Digital Forensics. Norsk Informasjonssikkerhetskonferanse (NISK) (pp. 112-123). [173]

2. Porter, K., & Petrovic, S. (2018, January). Obtaining Precision-Recall Trade-Offs in Fuzzy Searches of Large Email Corpora. In IFIP International Conference on Digital Forensics (pp. 67-85). Springer, Cham. [174]

3. Nordvik, R., Porter, K., Toolan, F., Axelsson, S., & Franke, K. (2020). Generic Metadata Time Carving. Forensic Science International: Digital Investigation, 33, 301005. [151]

4. Porter, K., Nordvik, R., Toolan, F., & Axelsson, S. (2021). Timestamp prefix carving for filesystem metadata extraction. Forensic Science International: Digital Investigation, 38, 301266. [172]

The following publications did not contribute to the dissertation, but were published during the duration of the PhD studies.

1. Hansen, J., Porter, K., Shalaginov, A., & Franke, K. (2018). Comparing open source search engine functionality, efficiency and effectiveness with respect to digital forensic search. Norsk Informasjonssikkerhetskonferanse (NISK). [94]

2. Porter, K. (2018). Analyzing the DarkNetMarkets subreddit for evolutions of tools and trends using LDA topic modeling. Digital Investigation, 26, S87-S97. [171]

In Figure 1.1, we provide a diagram that shows the relationships between the published papers contained within this dissertation, our research questions, and our overarching general evaluative methodology for the majority of our publications. The purpose of our experiments is to discover how much relevant content is found (*recall*), and how much irrelevant content is returned (*precision*) when applying our novel search algorithms with some chosen set of parameters. Discovering these quantities will help determine if we have made improvements to approximate search methods, thereby answering the crux of our research questions. Note, that all publications are concerned with applying exact or approximate string matching methods to digital forensics, though Papers 1 and 2 are reliant upon automata theory, and Papers 3 and 4 are reliant upon filesystem knowledge.

1.4.1 Major Contributions

The following major contributions of this research are listed below, where we also provide context to the contributions.

1. We created a novel constrained edit distance NFA and approximate string matching algorithm that gives, to our knowledge, the highest known control over edit distance "fuzziness" by allowing any combination of edit operations to be considered in an approximate search. Results from some of the newly reachable approximation parameters showed that our algorithm can increase the recall as high as possible without significantly decreasing the precision. This answers Research Question 1.

    - Most other approximate matching NFA's have one automaton row for each allowed number of edit operations, whereas ours has one automaton row for every possible edit operation combination. Other constrained edit distance (CED) approximate string matching algorithms can only accept a subset of the possible languages that ours can accept. Our hypothesis was that there is some combination of constrained edit

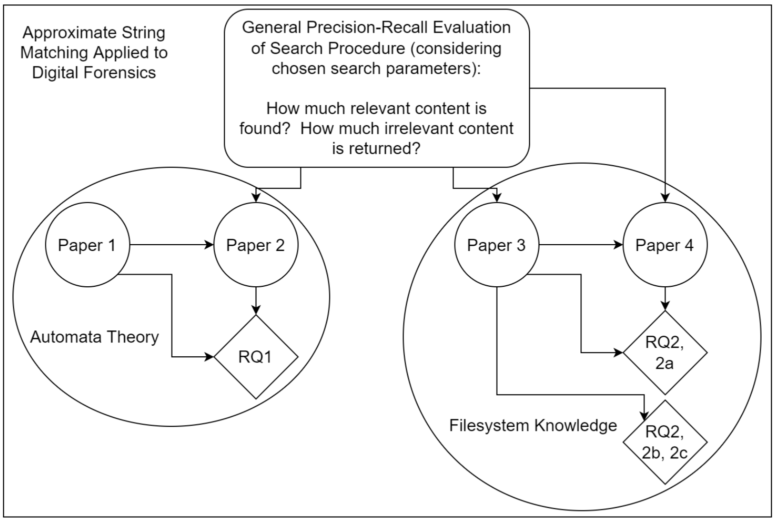

Approximate String Matching Applied to Digital Forensics
General Precision-Recall Evaluation of Search Procedure (considering chosen search parameters):
How much relevant content is found? How much irrelevant content is returned?
Paper 1
Paper 2
Paper 3
Paper 4
Automata Theory
RQ1
RQ2, 2a
RQ2, 2b, 2c
Filesystem Knowledge

Figure 1.1: The general outline of the dissertation. This figure shows the relationship between the different papers, which research questions they attempt to answer, and our overarching general evaluative methodology applied to Papers 2, 3, and 4. Note, all publications are concerned with applying exact or approximate string matching methods to digital forensics.

distance parameters that should yield a strong balance between precision and recall between the edit distances of one and two, limiting the presence of false positive matches.

- To test the precision and recall performance of the new algorithm, we searched for keywords on a list of terms derived from an inverted index of the Enron dataset, wherein every possible set of CED parameters was tested so that we can identify parameters useful for a digital investigative context. Our results showed that the identified set of parameters unique to our algorithm can increase the recall as high as possible without decimating the precision. Approximate matches accounting for two substitutions, two deletions, or a substitutions and a deletion are shown to drastically decrease precision. This was the contribution of Papers 1 and 2.

2. We created a novel string matching algorithm that carves for potential timestamps in filesystem metadata records, which in turn can be used as a generic "dynamic" signature to find and carve filesystem metadata records from at least several filesystems. This makes our method of filesystem metadata record carving more general than previous approaches, and allows acquisition

of metadata records even if the filesystem is corrupted. Results from empirically testing this algorithm followed by a filesystem specific parser to verify the timestamps and metadata records show that our method has near perfect precision in identifying metadata records and possibly high recall. These contributions answer Research Questions 2, 2a, 2b, and 2c. Source code and documentation of the research can be found on GitHub.[3]

- Our novel string matching algorithm searches a full image for repeated byte sequences of some size of m bytes, where the matching sequences are closely co-located together. We hypothesized that this is a generic characteristic of file system metadata record timestamps across many filesystems. This is the first phase of the file system metadata extraction methodology we created, *Generic Metadata Time Carving* (GMTC). Using the carved potential timestamp locations as input, file system specific parsers can validate if the locations are encapsulated by a metadata record or not.
- We applied the GMTC method to both NTFS and Ext4 synthetic disk images, where the filesystems had been made non-functional by reformatting the partitions with a completely different filesystem. Our experimental results showed nearly 100% precision in identifying metadata records, 100% recall for retrieving MFT records from a small and static NTFS disk image, and our tools succeeded in metadata extraction where commercial tools failed. The tools also provides a new method for connecting filename information to Ext4 inodes, where its success was dependent on the extent of the damage to the inode table and disk image. However, we have not studied other filesystems for connecting *Filename Content* to *Metadata Content* (see Section 2.3.1). These were the contributions of Paper 3.

3. We created an approximate (prefix matching) potential timestamp carving algorithm to increase the recall of our previous filesystem metadata carving method. The idea is that by only matching the prefixes of the potential timestamps, we can obtain higher recall than exact matching. This is because timestamps which are temporally similar should also be stringologically similar. Our results on realistic data showed an increase in recall anywhere from 86.4% to 87.1%, up to 8.8% to 97.9%, where results were dependent on the source of the data. We tested our methods on a challenging 2 GB NTFS image [145], a real but low population 59.5 GB Ext4 image from a Samsung S8, and a realistic but synthetic 476 GB NTFS image [134].

[3]https://github.com/RuneN007/cPTS

The contribution also answers research questions 2 and 2a. Source code and documentation of the research can be found on GitHub.[4]

- Used the prefix-based potential timestamp carver in the GMTC methodology, the recall compared to Paper 3 was greatly improved. For example, for the 2GB NTFS image [145] the exact matching GMTC method only obtained a recall of 8.8%, whereas the prefix-based version achieved a recall of 97.9%. The precision of all of our experiments remained at 100%, despite differing prefix lengths. Further investigation showed that this is due to the strictness of the parsers, but running the parsers with potential timestamp information can be 10 times faster or more than just running them alone. The timestamp carver effectively acts as a data reduction technique, controlling the number of potential timestamps the parsers must check.
- Additionally, the carving time of the prefix-based potential timestamp carver has the same time complexity as the exact matching potential timestamp carver, as the new algorithm only adds a constant number of steps. We believe that our GMTC tools are practical as well, as the entire process of applying our tools on a 476 GB disk image only took between 2.5 and 3 hours. This was the contribution of Paper 4.

1.5 Thesis Outline

This is a *compendium* dissertation, where the content is split between Part I and Part II. Part I contains information regarding the dissertation project, objectives, background, related work, results, and conclusions. Part II contains our published works, each occupying their own chapter.

We briefly cover the specifics of Part I. Chapter 1 has discussed the research topic, research questions, and major contributions of this dissertation. Chapter 2 discusses background information about our research topics and our methodologies. We first discuss our general scientific approach, followed by the evaluative methodologies. Section 2.2 specifically provides background information on string matching theory, constrained edit distance theory, and automata theory. Section 2.3 specifically provides background on the subject of digital forensics, and primarily discusses filesystems, with an emphasis on NTFS and Ext4 datastructures. Chapter 3 discusses related work with respect to string matching, approximate string matching, string matching in the context of digital forensics, file carving, metadata carving, and other approaches to minimize the digital forensics backlog.

[4]https://github.com/TimestampPrefixCarving/Peer-Review

Chapter 4 provides summaries of each of our four publications for this dissertation, and also highlights the contributions of each publication. We conclude in Chapter 5, where we answer our research questions, discuss the limitations of our research, and consider future work.

Chapter 2

Background and Methodologies

This chapter acts as reference material to understand the subsequent chapters in Part I of the dissertation. We begin by describing our general research methodology, then our methods to evaluate our research, and finally providing the necessary background information to discuss approximate string matching algorithms and filesystem metadata carving.

2.1 General Research Methodology

The general approach to our studies was simply to follow the scientific method to the best of our abilities. While the scientific method is systematic, Gauch [78] states that it is only a loose methodological framework. Outlining the scientific method may seem obvious to some, but we nonetheless list the steps of the method as we would like to discuss our approach to the different phases of the scientific method. The following steps are derived from Dykstra [59]:

1. Formulate a question from previous observations, measurements, or experiments.
2. Induction and formulation of testable hypotheses.
3. Making predictions from hypotheses.
4. Experimental testing and predictions.
5. Analysis and modification of hypothesis.

Generally speaking, the way specific questions and hypotheses were formulated was by performing a literature review. The beginning of our research called for a literature review of significant breadth and scope, as it was useful to understand the research subjects and their related schools of thought. However, typically when finishing an individual research project, we would again return to the literature review step. Specific research questions and hypotheses were also arrived at by way of analyzing the results of our research and discussions with colleagues.

As most of our research involves the assessment of string matching algorithms and their search results, we had the ability to consistently evaluate them with theoretical and quantitative methods. Thus, we used Big-O notation when deriving a created algorithm's asymptotic time complexity. This provides a theoretical upper bound on the running time (the number of elementary computational operations) of an algorithm as it consumes some unbounded input. It also provides a simple means to compare the efficiency of algorithms, and expected cost of using an algorithm. We note that we never provide space complexity assessments for our algorithms, as the search algorithms require very little working memory (either constant or restricted to constant) to operate.

The computational time complexity of an algorithm provides a strong basis of an algorithms expected behavior, but quantitative tests are still required so that we understand how the algorithms perform in practice. Thus, for all the created algorithms we measured their the time taken to complete their search task, where we averaged the results of multiple trials. To evaluate the performance of our methods, we measured the precision and recall of our experiments. This is described in the next section. Upon acquiring this information from our experiments, we could either confirm or reject our hypotheses, and potentially pursue new or modified research questions.

2.1.1 Evaluation Metrics: Precision and Recall

The intuitive notion of precision is to ask "*of hits returned from a string search, what percentage of the hits are relevant*"? Likewise, the intuitive notion of recall is to ask "*of the hits returned in the search, what is the total percentage of all the relevant items found*"? The two metrics will typically have an inverse relationship, creating a trade-off between precision and recall. Generally, as recall increases, precision will decrease [119]. Laurenson [113] also used these metrics to measure how successful commercial tools were at file carving.

The following metrics all require the notion of positive and negative samples. In digital investigative terms, an individual sample may be a single string hit provided by a keyword search, a hit for a relevant document, or a hit for a filesystem metadata record.

- A **positive sample** is a sample which is relevant to the investigator, or otherwise fits the class of information that the investigator set out to find.
- A **negative sample** is a sample that is not relevant for the investigator, and does not fit into the class of information that the investigator set out to find. For digital investigations, negative samples far outweigh the number of positive samples [119], which is why the analogy of searching for a needle in a haystack is so often used for digital investigations.

Our research developed several pieces of software that serve to return positive samples to the user. However, no test is without its errors. We describe important evaluative terminology below before defining precision and recall.

- **True Positive (TP)**: Positive sample is classified as positive (a true hit).
- **False Positive (FP)**: Negative sample is classified as positive. False positives are often a source of noise, making the usability of a tool difficult.
- **True Negative (TN)**: Negative sample is classified as negative.
- **False Negative (FN)**: Positive sample is classified as negative (a true miss). The occurrence of false negatives can be critical, as missing relevant evidence can be catastrophic for an investigation.

The following quantitative measures are used to measure the success of searching for positive samples in unbalanced datasets, where the class of negative samples heavily outweights the class of positive examples.

$$\textbf{Precision} = \frac{TP}{TP + FP} = \frac{|\text{relevant samples retrieved}|}{|\text{retrieved samples}|} \tag{2.1}$$

Precision states the percentage of hits identified as positive were actually positive.

$$\textbf{Recall} = \frac{TP}{TP + FN} = \frac{|\text{relevant samples retrieved}|}{|\text{total relevant samples}|} \tag{2.2}$$

Recall has a trade-off with respect to precision, as it identifies the total percentage of positive samples that were found. It is difficult to have both high precision and recall.

$$\textbf{F1-Score} = 2 \times \frac{(\text{precision})(\text{recall})}{(\text{precision}) + (\text{recall})} \tag{2.3}$$

The F1-score is the harmonic mean of the precision and recall, and is often used as a single metric to determine how well a classification task performed on an unbal-anecd dataset. We did not use this metric, since it is mostly used as a single metric to compare the overall effectiveness of different classifiers, rather than interpret the results of a test.

As indicated in the definitions above, we mostly rely upon the precision and recall for our evaluation metrics, as they are appropriate for unbalanced datasets and needle-in-the-haystack types of problems. Lillis and Scanlon [119] suggest that in the initial stages of an investigation precision is more important, and farther down the investigative timeline that obtaining higher recall becomes of higher priority. This is because the first hours of an investigation can be crucial to prevent further potential harm, and relevant evidence is needed as quickly as possible to find more leads. Later on in the investigation, the investigation may be more focused on finding additional relevant evidence for building a legal case.

We opted not to use other evaluation metrics such as the ROC curve and accuracy for several reasons. First, the choice for precision and recall has already been used for similar string matching [178] and digital forensics information retrieval tasks [25, 119]. Secondly, with respect to filesystem metadata carving, no standard approach appears to exist, as is shown by the differing evaluative metrics used in past research. For example, false positives, misses, and selectivity [55], or the recovery rate [163]. Furthermore, precision and recall are interesting metrics for "needle-in-a-haystack" types of problems. For our set of problems, the number of positives samples will always far outweigh the number of negative samples. Accuracy is of little interest in such scenarios since if 99.9% of the the samples are negatives, then a simple classifier that always guesses negative will achieve a 99.9% accuracy. ROC curves are also inappropriate to use for our research, as they are not well suited to datasets with large class imbalances [51].

2.2 String Matching

The string matching problem may be described as follows. Find all the occurrences of a given m length pattern $p = p_1p_2 \dots p_m$ in a large n length text $T = t_1t_2 \dots t_n$, where both T and p are sequences of characters from the alphabet Σ. An *alphabet* is a finite set of characters which changes depending on the application of the particular string matching problem. For example, for matching DNA sequences the alphabet is the set $\{A, G, T, C\}$, whereas the alphabet for file carving would be all possible 256 bytes (under the assumption we are not searching for Unicode or any other multi-byte encoding).

The study of string matching can be characterized by whether the string matching is exact or approximate, searching for a single pattern or multiple patterns simultaneously, or whether the search is *online* (only the search pattern is preprocessed) or *offline* (one may preprocess the search pattern and the text T). The novel search algorithms we created in the process of this dissertation are exact and approximate, single pattern matching, and exclusively online.

2.2.1 Edit Distance

The edit distance (often referred to as the Levenshtein distance) is a common metric used to quantify the similarity of strings for approximate string matching [115]. This distance and its variations are what we primarily use to measure the similarity between a search keyword and the subsequences in some text T when applying approximate string matching. The function for the edit distance is defined as $D(X, Y)$, the minimum number of elementary edit operations to convert some string X into another string Y [115]. The three elementary edit operations are defined as the *insertion* of a character into a string, the *deletion* of a character from a string, and the *substitution* of a character in a string for a different character. For approximate string matching, we consider a string X to match some string Y so long as $D(X, Y)$ is less than or equal to some threshold set by the user.

Other metrics used to measure the distance between strings are the Hamming distance (only substitution operations) [93], Damerau distance (allowing insertion, deletion, substitution, transposition operations) [50], q-grams [201], and many others.

2.2.2 Edit Constraints and the Constrained Edit Distance

The standard edit distance used in approximate pattern matching oftentimes produces too many false positives [41, 43, 42, 61], and it is for this purpose that we examine edit constraints [152]. Edit constraints limit the ways one string may be transformed into another by way of edit operations.

When measuring the edit distance between two strings X and Y while also considering the edit constraints in set C, [1] the distance function $D_C(X, Y)$ returns the *constrained edit distance* (CED). The types of edit constraints we may apply are fairly arbitrary if using a dynamic programming matrix,[2] and could be anything from the maximum number of allowed insertions [43], to the maximum allowed number of consecutive deletions [161]. There are cases in which the edit distance between strings X and Y may be small, but the constrained edit distance may be

[1]The set of edit operation constraints is often given the notation T, but since this is often the notation of the sequence of text characters being searched, we use C.

[2]Dynamic programming matrices [205] are the classic method for calculating the edit distance between two strings, and comes up in Section 3.1.2.

infinite due to the impossibility of transforming one string into another given the edit constraints.

We provide some more formalisms regarding the constrained edit distance, as defined by Oommen [152]. This requires the introduction of the null symbol ϕ, and is used to represent deletion and insertion operations. Let $\widetilde{\Sigma} = \Sigma \cup \{\phi\}$ and $x_i, y_j \in \widetilde{\Sigma}$, where x_i is the ith character from string X' and y_j is the jth character from string Y'. While strings X and Y may be of different lengths using the alphabet Σ, the strings X' and Y' are of equal length when using the alphabet $\widetilde{\Sigma}$ due to the null symbols being added into locations of the strings where an insertion or deletion occurs. The cost function $d_C(x_i, y_j)$ is mapped to a positive real number and defines the cost of the following elementary edit operations:

1. $d_C(x_i, \phi)$ is the cost of deleting character x_i from string X'.
2. $d_C(\phi, y_j)$ is the cost of inserting character y_j into string X'.
3. $d_C(x_i, y_i)$ is the cost of substituting x_i for y_i in string X'.

Using Oommen's formalisms we may express the edit operation transformation from string "aabcb" into "cakcb", where we can see the edit operations to transform string X into Y by looking at the adjacent characters between X' and Y'. For example, below we can see the first character "a" is substituted for "c", character "b" is deleted from X' denoted by the ϕ below, and character "k" is inserted into X' denoted by the ϕ above.

$$X' = \text{aab}\phi\text{cb}$$
$$Y' = \text{ca}\phi\text{kcb}$$

All work within this dissertation assumes that that $d_C(x_i, y_j) = 1$ for all $x_i, y_j \in \widetilde{\Sigma}$ except in the trivial case when $x_i = y_i$ for which $d_C(x_i, y_j) = 0$ (the case when a letter is substituted by itself).

2.2.3 Finite Automata

Our sophisticated approximate string matching algorithms use automata theory, where the string matching algorithm is either an implementation or simulation of a finite state machine, or *finite automaton*. As an algorithm parses through the individual characters from Σ of some text T, a finite automaton will accept or reject subsequences of $S \in T$ as matches for the search pattern.

A *deterministic* finite automaton may be defined as the tuple $(Q, \Sigma, \delta, I, F)$. Q is a finite set of states, where a state may be *active* or *inactive*. A common illustration of a state is an empty circular node (see Figure 2.1), where it is shown as active if its interior is darkened. Inactive states can only be made active by other states, and in the context of string matching an active state denotes that some $|S|$ length prefix or suffix of the search string has been identified. More than one state may be active at a time, and all states are updated each time the string matching algorithm parses in a new character from T. However, if an active state is not explicitly activated by another state during an update, then it will become inactive. I, a subset of all states Q, is the set of initial states that are active when the automaton is first initialized. This is typically illustrated with a large bolded arrow pointing to a state, where the origin of the arrow has no state. In practical terms, the initial states look for the beginning of the given search pattern. Any state from Q may have a transition between itself and another state. This transition is defined by the transition function $\delta : Q \times \Sigma \rightarrow Q$.[3] The input of this function is some active state, where if the current input character of the automaton from the alphabet Σ matches the character or possible set of characters given by the transition (these may be character from the search pattern), then a single state in Q is made active. This may be a state reactivating itself, or it may be activating a different state. A transition is illustrated by an arrow with a character or set of characters above the arrow. The subset of states F are known as the set of *terminal states*, wherein if they are made active, it signifies that a match has occurred. Terminal states are illustrated with a circular node with a smaller circle inside. The set of strings accepted by the automaton that cause a match are known as a language L [193].

There are two kinds of finite automata, deterministic and nondeterministic. The transitions of a deterministic finite automaton (DFA) can only, at maximum, activate a single state when reading in a character. In Figure 2.1, we show a DFA that can match the string “aabcb”. One can easily see that a DFA for even a simple pattern produces some rather complex transitions, as this is required to keep at least one of the states in the automaton alive.

A nondeterministic finite automaton (NFA) differs from the deterministic variation in several ways. The largest difference is that in an NFA a transition can activate multiple states simultaneously. Another important difference between an NFA and DFA is the addition of the *epsilon transition*, ϵ, into the alphabet Σ. Essentially, ϵ represents a transition between states that does not require an input character. So an active state with an outgoing epsilon transition will immediately make the subsequent state active. In Figure 2.2, we show a string matching NFA for “aabcb”, where no epsilon transitions are used.

[3]Nondeterministic finite automata use the transition function $\delta : Q \times \Sigma_\epsilon \rightarrow P(Q)$ [193], where $P(Q)$ is the powerset of Q. This means that the transition may activate any subset of states in Q, causing more than one state to become active simultaneously. Σ_ϵ defines an alphabet which includes the epsilon transition.

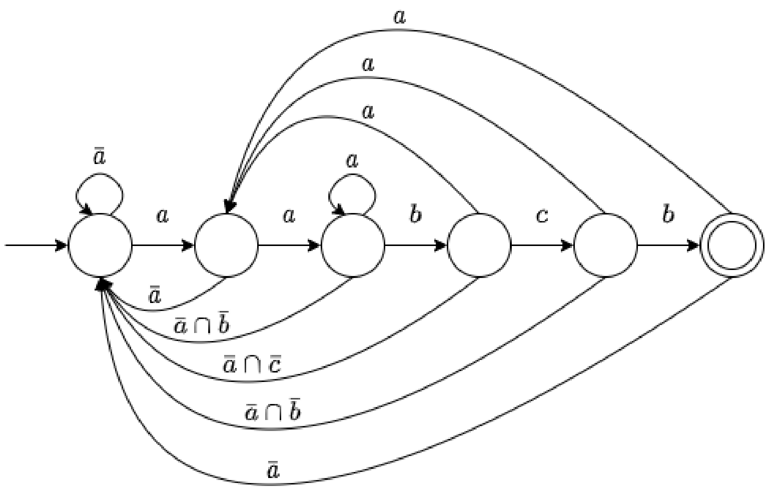


Figure 2.1: DFA for the pattern "aabcb", where the initial state is denoted by the lone arrow pointing to the state on the left, and the terminal state is denoted by circle enclosed within another circle. Example is inspired by Navarro and Raffinot [142]

.

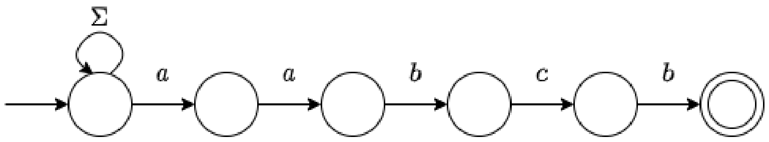


Figure 2.2: NFA without ϵ-transitions for the pattern "aabcb", where the initial state is always active. Note the transition which reactivates the initial state. The use of Σ means that the transition function accepts any character from the alphabet to activate the state it is pointing to. Example is inspired by Navarro and Raffinot [142].

NFAs are theoretically more complex, and DFAs are graphically more complex, but both DFAs and NFAs have the exact same expressive power. That is, NFAs and DFAs can recognize the same class of languages [193]. Furthermore, every NFA has an equivalent DFA, and they may be transformed into one another. This is useful since DFAs and NFAs have diametrically opposed efficiency qualities. DFAs are considerably faster than NFAs since they run linearly in the length of the input text, but DFAs require up to $2^{|NFA(Q)|}$ states where $|NFA(Q)|$ is the number states of the equivalent NFA [97]. The reason for the relatively slow speed of NFAs is because they are nondeterministic, so they cannot be directly implemented in standard von Neumann architecture and are typically simulated. Thus, both automata are utilized in different contexts.

While automata theory may seem abstract, they are often used practically in digital forensics. String matching algorithms and Regular Expression search are typically implementations or simulations of DFAs or NFAs. Finite state machines (FSM) have also been used to model the possible states that some digital system may contain (and all the possible transitions from state to state), where the FSM is intended to assist event reconstruction [82].

2.2.4 The Edit Distance NFA

While it is possible to model DFAs which can perform approximate string matching, it is much easier to create an NFA which does so. A common method of performing approximate string matching, as implemented by the approximate string matching suite *agrep* [208], is by using the nondeterministic finite automaton (NFA) for approximate matching as seen in Figure 2.3. It is an NFA which is capable of performing approximate string matching within a threshold of k edit operations.[4] We refer to this NFA as A_k.

In most approximate string matching NFA, horizontal arrows represent exact character matches. Diagonal arrows represent character substitutions, and vertical arrows represent character insertions where both transitions accept any character in Σ. Since this is an NFA, it also allows for ϵ-transitions, where transitions are made without needing a prerequisite character. The dashed diagonal arrows are ϵ-transitions, which represent character deletions. We typically draw such NFA without including the Σ or ϵ characters over the transitions, as it adds visual noise to the figures. For approximate search with an edit distance threshold of k, this automaton has $k + 1$ rows.

What makes the automaton A_k so effective for pattern matching is that it can check

[4]This is not to be confused with the *Universal Levenshtein Automaton* by Mihov [132], which is a fast DFA implementation of A_k

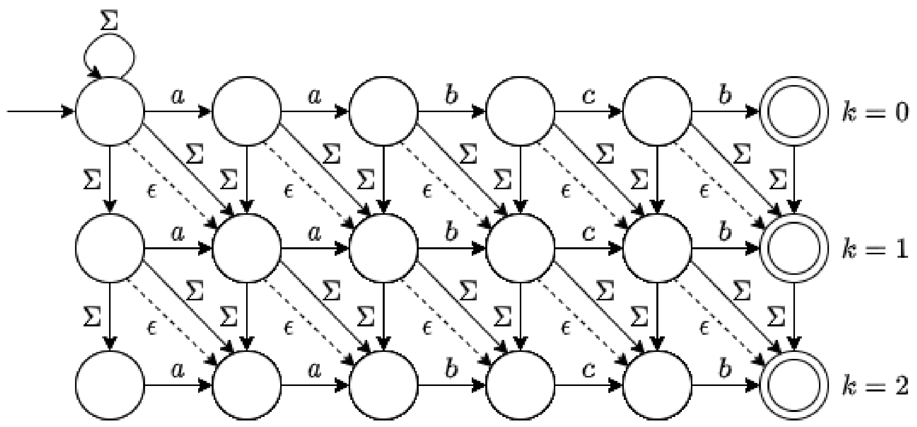


Figure 2.3: NFA A_k that matches the pattern "aabcb" allowing for two edit operations.

for potential errors in the search pattern simultaneously. For every character ingested by the automaton, each row checks for potential matches, insertions, deletions, and substitutions against every position in the pattern.

2.2.5 Bit-Parallelism NFA Simulation

Technically introduced in the context of compilers [57], bit-parallelism has been used to simulate NFAs for exact string matching [13], approximate string matching [209], as well as simulating dynamic programming matrices [136]. As stated previously, NFAs cannot be truly implemented in von Neumann architecture. Bit-parallelism for NFA simulation requires that bit-vectors represent each row of the automaton (typically held as a computer 32 or 64-bit computer word), and are updated by way of basic logical operations which correspond to an automaton's transition functions. Bit-vectors' binary nature make them a natural representation for the status of active or inactive states within a row of an automaton. Generally, this technique is recognized as one of the fastest methods for approximate string matching [65]. Bit-parallel simulation uses the intrinsic parallelism of the bit operations inside a computer word, and reduces the number of operations an algorithm performs at most by the number of bits w in the computer word [33]. In other words, bit-wise operations update all the bits of a bit-vector at once, and therefore updates the states of a row of an automaton or column of a dynamic programming matrix simultaneously. So if the number of characters in a search pattern is less than or equal to the number of bits in a computer word, then an algorithm's performance will be extremely fast.

The earliest algorithms to use bit-parallelism approximate string matching were by

Table 2.1: Bit-mask table for the search pattern “aabcb”

Character t_j	**Bit-mask** $B[t_j]$
a	00000110
b	00101000
c	00010000
*	00000000

Baeza-Yates and Gonnet [13], and by Wu and Manber [209], who also implemented the agrep Unix program [208]. The Wu-Manber Shift-And based algorithm [12] runs in $O(kn\lceil m/w \rceil)$, where n is the length of the text, m is the length of the pattern, k is the allowed number of errrors, and w is the size of the computer word. The bit-parallel algorithms by Wu and Manber [209] and Baeza-Yates [14] have served as the basis for many of the improved approximate string matching algorithms that were later invented.

To describe the Wu-Manber Shift-And algorithm (a simulation of the Levenshtein NFA), we introduce some fundamentals of bit-parallelism. Each row of the automaton for search pattern X is simulated with a binary vector of length $|X| + 1$,[5] where a bit of value 1 represents an active state of the automaton, and a bit value of 0 represents non-active state of the automaton. We also create a table of binary vectors $B[t_j]$ each of minimum length $|X| + 1$ used for comparing input characters t_j to the state of automaton, thereby possibly activating new states. The binary vectors within the table are called *bit-masks*. Bit-masks are representations of the input characters t_j, where bits set to one represent the position of the character t_j in the search pattern, and a 0 represents that the character t_j is not present in a location of the search pattern. For example, Table 2.1 gives the bit-mask representations of the characters of the search pattern “aabcb”. Characters that are not contained within the search pattern are represented by the * symbol. Notice that the positions of the letters are in the reverse order of how they are actually written. This is simply to make the Shift-And algorithm more efficient, so for the computer it is processing an NFA simulation that goes left rather than right [142]. Furthermore, the first bit in the bit-masks are set to 0, as this accounts for the initial state of the automaton.

With the previous preprocessing steps completed, we can simulate nondetermin-

[5]Bit-parallel simulations of NFA may also have less bits than the number of states in the row of its automaton. For instance, the initial states can be implied when implementing Shift-And or Shift-Or (described later in Section 2.2.5). Alternatively, the bit-vectors may only need to be the length Y of the diagonals in the automaton [12]

istic finite automata approximate string matching (see Figure 2.3), where a Shift-And version of the algorithm is shown in Algorithm 1. Each row of the automaton is represented by bit vector R_i, $0 \leq i \leq k$ where k is the total number of allowed errors. The algorithm updates each row of the automaton R_i with respect the transitions on the automaton. The initial state of R_0 is 0x00000001, R_1 is 0x00000010, R_2 is 0x00000100, and so on. This activates the initial state of the automaton and any states the initial state activates by way of ϵ-transitions.

Algorithm 1: Shift-And Bit-parallel simulation of the NFA for approximate string matching, based on Navarro and Raffinot [142].

```
    $R'_0 \leftarrow ((R_0 << 1)\ \&\ B[t_j])|\ 0x00000001$
    $R'_i \leftarrow ((R_i << 1)\ \&\ B[t_j])\ |\ R_{i-1}\ |\ (R_{i-1} << 1)\ |\ (R'_{i-1} << 1)$
    $R_0 \leftarrow R'_0$
    $R_i \leftarrow R'_i$
```

The update rule for R_0 handles the exact matching case with no errors. By shifting the row one bit to the left, and ANDing it with the bit-mask of character t_j, we update the states in this row of the automaton. This is ORed with 1 so that we can re-activate the initial state for all iterations. The second row of the algorithm must be iterated through (in order from 1 to k) to update the other rows of the automaton. The rows of the automaton account for possible edit operation errors. The result of the exact matching per row is then ORed with some bit-wise operations that represent different edit operation transitions. Reading the second row of the algorithm from left to right, the first expression accounts for exact character matches, the second expression accounts for insertions, the third expression accounts for substitutions, and the fourth expression accounts for deletions. A match is encountered when any of the bit-vectors representing a row of the automaton has its $|X| + 1$th bit set equal to one, which represents that a terminal state has been activated.

We note that this algorithm can be improved by using the Shift-Or algorithm, where the use of an OR rather than an AND (as well as negating the Boolean values of the bitmasks) will allow for an algorithm that saves a computational step for each iteration of the algorithm [13]. We did not implement our algorithms using the Shift-Or construction, as it is more difficult to plan and build than Shift-And constructions.

This string matching theory is required to understand Paper 1 and 2. Papers 3 and 4 also use a kind of string matching algorithm, but due to the simplicity of the matching task, automata theory and bit-parallel algorithms are unnecessary (and even less efficient). We do however use the same string matching terminology.

2.3 Digital Forensics

The purpose of this section is to introduce the subject of digital forensics, and to provide context to our research. After covering some of the basics and principles of the field, we narrow our discussion to briefly describe filesystems, as they are integral to Papers 3 and 4.

To define digital forensics, we use the often quoted definition of digital forensics from the first Digital Forensic Research Workshop (DFRWS) by Palmer [157], which concisely captures what is entailed within digital forensic activities.

> "The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations." [157]

Forensics, by definition, is inherently scientific. In practice, however, appropriately scientific methods have not always been used to collect or produce evidence. The failure to adhere to a scientific approach to evidence, and the collection and analysis thereof, has resulted in the introduction of pseudo-science into expert testimony of actual court cases [99]. The acceptance of pseudo-science may lead, and has led, to false convictions and imprisonments [101]. In the United States, the previous standard for the admissibility of scientific testimony was the *Frye general acceptance test*,[6] but in 1993 the United States Supreme Court introduced the *Daubert Standard*, a much stricter standard for allowing expert testimony [79]. The standard, which is still in use by United States Federal courts [189], requires the following: the theory or technique has been tested, whether it has been peer reviewed or published, its error rate, whether the standards of the method or technique are maintained, and whether it has been widely accepted by its relevant scientific community [1, 79, 189]. Digital forensics is subject to the Daubert Standard as well, and work to *Daubert-proof* digital forensics is ongoing [125].

The quote from Palmer [156] refers to a series of phases entailed in a digital investigation, and though different digital forensics process models exist [9, 22, 35], we will be referring to the defined process model later in the same work by Palmer. We quickly summarize the phases of the process model, highlighting where our contributions lie.

[6]Frye v. United States, 293 F. 1013, 1014 (D.C. Cir. 1923)

1. **Identification** involves the detection of a crime or anomaly that warrants further investigation.

2. **Preservation** involves the processes necessary to preserve any digital evidence in its original form, and documentation of the evidence handling, thereby providing a *chain-of-custody*. This phase includes the read-only imaging of hard disks, solid state drives, etc.

3. **Collection** includes the acquisition of digital evidence onto an investigator's workstation. The collection and preservation phases are often merged into a single phase.

4. **Examination** refers to steps taken to investigate the data directly. This includes aspects such as keyword search, document filtering, or file carving. All our research papers presented in this dissertation fall into this phase of a digital investigation. Essentially, it is the search for relevant data.

5. **Analysis** is the process of determining if relevant data is potentially evidence.

6. **Presentation** is the last phase of the process model, which includes documentation of the actions taken when identifying relevant evidence, and even providing testimony.

Many other terms can be used when describing an investigation and the types of digital forensics it may require. For instance, a crime may be physical with a human victim, and devices may be sources of evidence. Alternatively, the crime may be digital, where the victim is a device and may also be used as a source of evidence. An investigation may occur on a device that cannot be turned off (for example, if it is using disk encryption), and thus *live forensics* must be performed because the information is volatile (the data may be lost if the device loses power). On the other hand, if a device can be turned off and imaged, one will be performing *post-mortem* analysis on non-volatile information. Further complicating digital forensics and investigations is that digital forensics can be broken down into various (but often overlapping) sub-disciplines. Such disciplines include Network Forensics, Filesystem Forensics, Memory Forensics, Email Forensics, Mobile Forensics, Cloud Forensics, etc.

So, our research would fall into the category of examination techniques applied to filesystem forensics and email forensics, where the data has been written to disk and is non-volatile. However, as some of the string matching tools and theory we have come up with is very general, they can likely be applied elsewhere as well.

As our work is significantly concerned with filesystems, we provide some background information regarding filesystems, and in particular NTFS and Ext4.

2.3.1 Filesystems

The purpose of this subsection is to provide the reader with a primer on the subject of filesystems. We introduce the subject with discussing how disks may be formatted.

A physical disk image is a bit-for-bit copy of the physical disk, which may be composed of single or multiple *volumes*. Carrier [34] defines volumes as "a collection of addressable sectors that an Operating System (OS) or application can be used as storage." *Sectors* are the smallest storage unit on a disk, typically being 512 bytes long. The smallest addressable unit on a disk is known as either *blocks* or *clusters* depending on the filesystem, which are made up of one or more sectors. Volumes do not always need to be a consecutive sequence of sectors on a physical device. Volumes, such as PC hard drives, may be divided into *partitions*, "a collection of consecutive sectors in a volume" [34]. A partition is always a volume, but a volume is not always a partition (for example, in the case of a USB storage device). We refer to a volume that contains one or more partitions as the *parent volume*. Any particular volume may or may not contain a file system, and partitions are kept track of by a partition table at the beginning of a parent volume.

A simplified diagram of a disk layout with partitions is shown in Figure 2.4. A physical disk acts as the parent volume, where the disk has been partitioned. A partition table is used (GUID partition table as an example [131]) to keep track of the partitions on the parent volume, where the primary version of the table is located at the beginning of the disk and a copy is located at the end of the disk. The partitions are represented by the rectangles with rounded corners. This example shows a dual boot disk, containing an Ext4 and NTFS operating system. The first two partitions make up the contents of the Linux OS, and the next two partitions make up the contents of the Windows OS. The last partition on the disk has no filesystem.

A filesystem essentially provides a structure to a volume that allows an operating system to retrieve and and keep records of files on the volume in an efficient way (including itself). Microsoft and Apple operating systems typically use their proprietary filesystems of NTFS and APFS for volumes they utilize, but the volumes may be formatted with a different filesystem such as Ext4. The different filesystems themselves are actually quite distinct, though according to Carrier there are five different abstract categories that are found in most filesystems: the File System Category, the File Name Category, the Metadata Category, the Content Category,

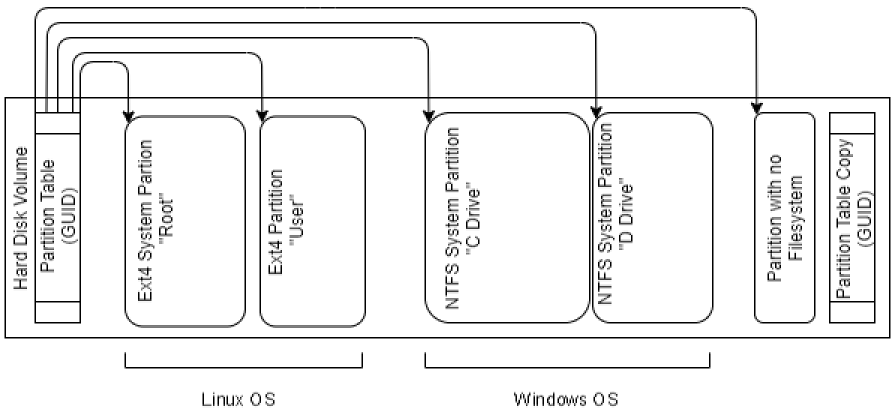

Hard Disk Volume
Partition Table (GUID)
Ext4 System Partion "Root"
Ext4 Partition "User"
NTFS System Partition "C Drive"
NTFS System Partition "D Drive"
Partition with no Filesystem
Partition Table Copy (GUID)
Linux OS
Windows OS

Figure 2.4: Simplified example view of a disk layout.

and the Application Category [34]. We first review the different abstract categories, as it is helpful to describe the data structures in NTFS and Ext4. We begin by describing the most granular components of a filesystem. First, the Content Category, and then we follow with the File Name and Metadata Categories.

The Content Category describes the actual files held on a volume, and the locations on the volume of where the file data is stored. The smallest storable data unit for NTFS is called a *cluster* and in ExtX filesystems (Ext2, Ext3, and Ext4) they are called *blocks*, but for NTFS and Ext4 they typically have the same default size of 4096 bytes (8 512-byte sectors). Files that are written to disk and known to the filesystem are known as *allocated files*, and files that are no longer known to the filesystem are known as unallocated files. Any clusters or blocks that are not allocated by the filesystem is known as unallocated space,[7] and the act of searching through unallocated space for files is known as *file carving*. Filesystems will often keep track of the allocation status of their blocks or clusters by utilizing *bitmaps*. These are datastructures where individual bits represent the allocation status of all blocks or clusters on the file system. Files, whether they are allocated or not, may be stored in a contiguous stretch of blocks or clusters, or multiple separate stretches of blocks or clusters. The latter phenomena is known as fragmentation, and is the cause for many of the challenges regarding file carving.

The Metadata Category is perhaps the most commonly known kind of filesystem data structure, where filesystems use descriptive data structures per file which we

[7]We do not consider cluster slackspace or file slackspace to be unallocated space. *Slackspace* is the end of some allocated space that a file or cluster is not using. Thus, it is a source of potentially residual data.

in general refer to as *filesystem metadata records*. NTFS refers to these data structures as Master File Table (MFT) entries, and ExtX filesystems refer to these data-structures as inodes. Typically speaking, file system metadata records record some commonly desirable attributes regarding a file. They will record *MAC timestamps* of a particular file, which refers to the last time the file was modified, the last time the file was accessed, and the time at which the file was created (though, *MAC* also encapsulates other categories of timestamps as well [69]). Different filesystems may contain different categories of timestamps. Other important information a filesystem metadata record holds are the size of the file, as well as Content Category information such as the clusters of the image that the file is stored in. NTFS and Ext4 also differ with regards to their deletion policy when it comes to filesystem metadata records, as MFT records will keep the storage locations of the file and Ext4 will erase the storage locations when a file is deleted.

The File Name Category refers to the data structures that contain a file's name and/or record number, an index number that used by the filesystem to refer to that particular record. NTFS and ExtX handle their File Name content very differently, as NTFS combines the Metadata and File Name information and ExtX keeps them separate.

The Application Category often refers to a filesystem's journaling datastructures, which record how a filesystem was most recently updated. Such journaling systems are often relatively small in size (only the most recent changes to the filesystem are recorded), and will only sometimes contain complete filesystem metadata records.

The File System Category contains data structures or files that provide information about the actual filesystem itself. This may include layout and size information of filesystem critical data structures. For example, NTFS uses the $MFT file, the Master File Table, which keeps track of where all the known MFT records exist on the volume. An example of Ext4 File System data includes the Superblock, which contains information regarding the total number of inodes, blocksize, and other important information regarding the location of the inodes. Data structures within the File System Category are usually critical for a fully functional filesystem, as they define parameters that will be used for calculating locations of metadata and files.

We now discuss two filesystems that are frequently examined in our research.

2.3.2 NTFS

NTFS is a proprietary filesystem developed by Microsoft, and NTFS version 3.1 has been the default filesystem for Windows operating systems from Windows XP

[7] through its future iterations, including Windows 10.[8] In this subsection, we discuss how NTFS is generally structured, and some of the important datastructures we utilize. We describe this information using the abstract categories defined in the previous subsection.

The most critical *File System* metadata file in NTFS is the *Master File Table* ($MFT). The file is composed of individual MFT entries for every single allocated file and directory on the filesystem. MFT entries (or alternatively, records) are NTFS's filesystem metadata record, where each allocated file or directory has at least one associated record containing its *Metadata* information, some *Filename* information, and possibly *Content* information. Even $MFT has an entry for itself, being the first record in the table. The operating system reserves 12.5% of the space of the volume for the MFT [106], allowing for the possibility of having billions of files. In cases where $MFT is very large, it is possible that the table will be fragmented across different parts of the volume. Other MFT records for the *File System*, *Content*, and *Application* categories take up the beginning records of the MFT. Of particular note is $MFTMirr, which is a short file located elsewhere on the volume that contains duplicates of filesystem critical metadata records, including $MFT, $MFTMirr, $LogFile, and $Volume [66]. In the case that the $MFT record is overwritten or otherwise corrupted, the $MFTMirr record can be used as a back up. The $LogFile is a critical *Application* file, as it contains *transactions* of how MFT records have recently been changed or updated. We discuss this file in more detail later in this section. Then the $Bitmap record points to the Content Category Bitmap file, which is a map of which clusters of the volume are allocated or unallocated.

MFT records are the primary source of *Metadata* information for NTFS. Each record is two sectors long (1024 bytes), where the record is composed of a standard MFT header, and followed by a number of *attributes*. It is both frequent and possible that not all 1024 bytes of an MFT record are used. We do not cover all of the possible attributes found in an MFT record, but we discuss the ones that are relevant to our research. The MFT record header is 56 (0x38 hexidecimal) bytes long, where its first 4 bytes make up a MFT record signature. The signature is written in ASCII as either FILE or BAAD, and these keywords can be used to perform a text search for MFT records. Attributes that follow the MFT record header all begin with a standard attribute header. The header contains a 4 byte attribute type identifier, the total length of the attribute, and other important information for the filesystem. Note, that attributes with identifiers that are of greater value than other

[8]It appears that NTFS 3.1 is simply referred to as *NTFS* these days, according to the Microsoft NTFS documentation https://docs.microsoft.com/en-us/windows-server/storage/file-server/ntfs-overview.

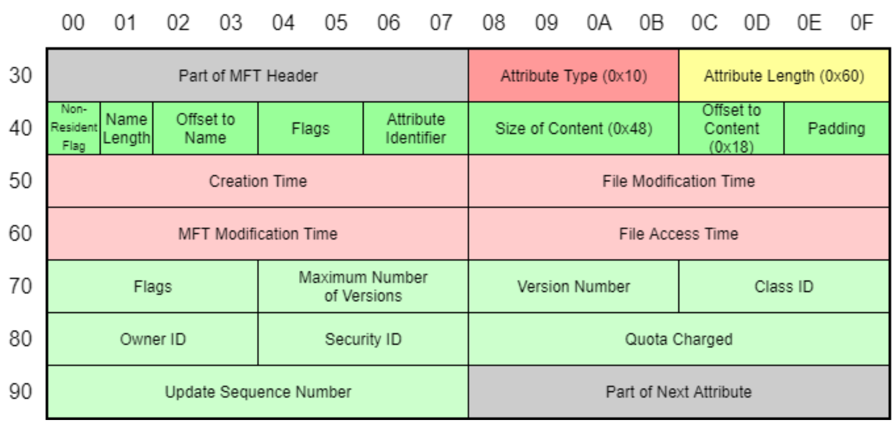


Figure 2.5: A color coded layout of a Standard Information Attribute (SIA) datastructure from an MFT record [34, 66]. The hexadecimal denotes the offset from the beginning of the MFT record. Attributes highlighted in pink represent data that helps us either carve or verify the filesystem metadata record. Attributes highlighted in yellow denote that we may use this information for further analysis. Attributes in green show the remaining data in the SIA. The darker colors show the attribute header, and the lighter colors show the attribute content. The attributes in grey are other parts of the MFT record.

attributes must always follow the attributes with smaller values. Furthermore, our research does not deeply consider all possible attributes, and when scanning an MFT record we can use the attribute type identifier and the attribute lengths to skip to attributes of interest.

The first attribute following the MFT header is the *Standard Information Attribute* (SIA). This attribute exists in all MFT records, and always has a length of 96 (0x60) bytes. It's attribute type identifier has the value 16, but like all other numeric values stored in NTFS, the value is written in *little-endian* (0x10000000). That is, the largest byte is on the rightmost side of the byte sequence, as opposed to big-endian which stores its largest byte value on the leftmost side. The only other items we take a great interest in for the SIA are its timestamps. It contains four different timestamps, each 8 bytes long and all contiguously written: the creation time, the file modification time, the MFT modification time, and the file access time. NFTS timestamps describe the count of 100 nanosecond intervals from January 1, 1601 [67]. Figure 2.5 illustrates the datastructure of the SIA attribute.

The next attribute we take a strong interest in is the *Filename Attribute* (FNA), where its attribute type identifier is 48 (0x30000000). This attribute contains *File Name* information such as its filetype (whether it is a file, directory, etc), parent directory and other indexing information, timestamps, as well as a given filename.

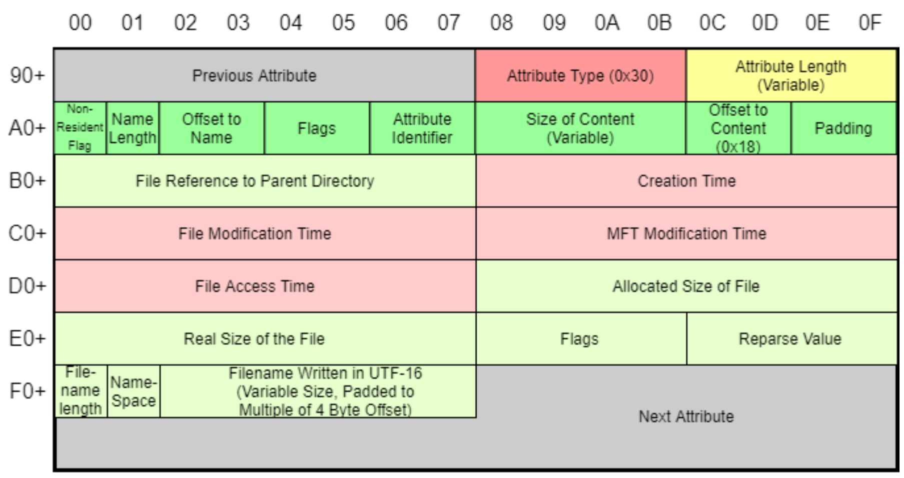


Figure 2.6: A color coded layout of a File Name Attribute (FNA) datastructure from an MFT record [34, 66]. The hexadecimal denotes the offset from the beginning of the MFT record. The vertical access writes its offsets as 0x90+ and etc since 0x90 is the earliest we may encounter the FNA, as it is possible there is an Attribute List attribute between the SIA and FNA. Attributes highlighted in pink represent data that helps us either carve or verify the filesystem metadata record. Attributes highlighted in yellow denote that we may use this information for further analysis. Attributes in green show the remaining data in the FNA. The darker colors show the attribute header, and the lighter colors show the attribute content. The attributes in grey are other parts of the MFT record.

Unlike the Standard Information Attribute, the size of the attribute is variable due to the dynamic nature of the filename length. The types of timestamps recorded in an FNA are identical to that of the SIA, and likewise, they are recorded in the same order contiguously. However, the timestamps themselves are often different. This is because timestamps in the FNA are rarely changed compared to the SIA timestamps, where they are typically only updated when an MFT record is created or the filename is changed [44]. It is possible that a file, and therefore its MFT record, has more than one name, and so there is the possibility for an MFT record to have multiple Filename Attributes. Figure 2.6 illustrates the File Name Attribute datastructure.

The final attribute that we take an interest in is the *Data Attribute*, where it includes *File Content* information such as pointers to the file data, or even the file data itself. The attribute type identifier is 128 (0x80000000), and the size of the attribute will vary dependent upon the written length of the pointers to the data. If the record's corresponding file is small and can fit in what is left of the 1024 byte record, then the file data will be recorded here. This is known as a *resident file*. Files which are larger than this space are known as *non-resident files*, and the data attribute will contain pointers to the location of the data along with the size of the file fragments stored at the listed locations. This is known as a *datarun*. Typically, a directory will not contain a Data attribute [66]. As a final note, we do not consider *Alternate Data Streams*, which are additional data attributes [34].

When a file is deleted in NTFS, the MFT record, and any non-resident attributes or non-resident data are simply unallocated, but the actual MFT record remains completely intact [34]. Finding the MFT record may allow for file recovery if the MFT record has not already been overwritten by another record, or if the file itself has not already been overwritten.

The next datastructure we briefly describe from NTFS is the $LogFile file, a journaling file for NTFS. The only aspect of the file that interests us is the possibility that the journal contains complete MFT records. The log is organized into 4096 byte sized pages, where each page has a 48 byte header with the magic number "RCRD". We primarily focus on the "logging area" that records transactions regarding the creation, modification, renaming, or other kinds of operations done on a file [34]. The transactions may be split into one or more transaction records, which vary in length and content, but contain a standard header that features two important attributes that describes a part of a transaction: the Redo Operation and the Undo Operation [45]. These are important, as they indicate the type of transaction being performed (potentially over a series of records), and when a file or directory is being created a full MFT record will be stored in the transaction record [48]. Either the Redo and Undo Operation values must be 0 and 2 respectively, or 2 and 0 respectively. It is also possible that a transaction record spills over the length of a $LogFile page, thus cutting a transaction into two with its header.

We have omitted many details regarding MFT records, but the interested reader can read Carrier's [34] book on filesystem forensics. The only remaining details regarding MFT records we wish to mention is it is possible that an MFT record will have either large attributes or more attributes that can fit into the space of the 1024 byte constraint. This will cause the attribute to be non-resident, and stored on a different cluster.

2.3.3 Ext4

Ext4 is an extended version of Ext2, and Ext3 (collectively, these filesystems are refered to as ExtX) commonly found in Linux distributions. While there certainly are items unique to Ext4, there are also many similarities between the file systems. The general idea in Ext4 is that unlike NTFS which organizes all of its filesystem metadata records in a single logical space, Ext4 divides its filesystem

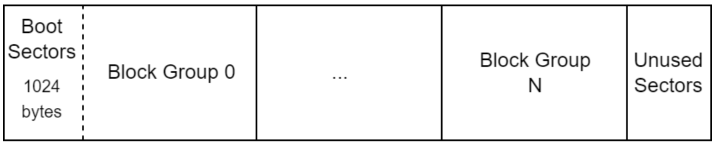


Figure 2.7: Typical Ext4 volume layout, adapted from Fairbanks [64]. The volume is composed of N block groups, starting from block group. If the block is greater than 1024 bytes, then the boot sector padding at the start of the volume is included in block group 0, where it is followed by the superblock and group descriptor table [62]

metadata records across *block groups* that exist throughout the volume. These block groups have their own datastructures for managing their respective files, filesystem metadata records (the *inodes*), and also contain redundant information in the case of filesystem corruption.

We begin by describing Ext4 datastructures from the *File System* category. The *superblock* for an ExtX volume is a datastructure following a 1024 byte padding from the start of the volume, and contains prerequisite variables such as the block size (default size is 4096 bytes), the number of blocks per block group, the total number of inodes, and the maximum number of inodes per block group. The superblock is contained within a single block, and is followed by the *group descriptor table*, which holds information about where various filesystem datastructures can be found in each block group in the filesystem. Typically, the padding, superblock, and group descriptor table are all found at the beginning of block 0 [62]. Figure 2.7 shows the typical layout of an Ext4 volume.

Block groups are made up of several distinct sections. If the filesystem is not *sparse*, then each block group will contain a backup superblock and group descriptor table in the beginning of its first block [62]. Following the possible duplicate datastructures is a block bitmap which keeps track of allocated blocks within the given block group, and there is also an inode bitmap which keeps track of the allocation status of inodes in inode table of the given block group. What follows is a portion of the *inode table*, the datastructure containing the inodes for all the allocated files and directories on the filesystem (and possibly inodes for unallocated files as well). The remaining data of the block group is made up of the file content where actual files and directories are stored. Figure 2.8 illustrates the structure of a typical block group.

The final item needing to be mentioned regarding the layout of Ext4 is the notion of *flexible block groups* (alternatively referred to as *flex-groups*). Flex-groups cause some number of consecutive block groups to become logically merged, wherein the individual block group bitmaps and inode tables are appended to one another,

Super Block	Group Descriptor Table	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks
1 Block	Many Blocks	1 Block	1 Block	Many Blocks	Many More Blocks

Figure 2.8: Typical block group layout, adapted from Fairbanks [64]. The notes regarding the sizes of the different sections of the block are from the Ext4 Disk Layout Wiki [62].

and the remaining space in the flex group is reserved for files and other data content [62, 55]. In cases where flex-groups are used, there will be far less fragmentation of the inode table

Inodes, the filesystem metadata records for ExtX, contain much less information than their NTFS counterparts. For Ext4, the default size of an inode is 256 bytes, but they may be 128 bytes long as well. All inodes have a file mode flag (to determine if the inode corresponds to a file, directory, symbolic link, etc), filesize and block count, and timestamps. The timestamps used in ExtX are four bytes long little-endian, and consecutively listed as Access Time, Creation Time, Modification Time, and Deletion Time. ExtX timestamps are measured in seconds since January 1, 1970 (in Ext4, sub-second values of the timestamps are stored in a different location within the inode) [169]. Figure 2.9 is a diagram of an inode, where we color code its different attributes dependent upon how we utilize them.

In Ext4, inodes have two possible methods for recording where they store their file or directory data. Inodes can use the system utilized by Ext2 and Ext3 that held a series of block locations either directly or indirectly pointing (via *indirect blocks*) to its respective data, or inodes can use the Ext4 specific approach by using *extents*. The former approach directly lists such pointers, whereas the latter approach contains a header followed by up to 4 index nodes of the *extent tree*, a tree-like datastructure for recording offsets to file data. The index nodes either point to leaf nodes holding data, or it may point to other extent nodes. Of note, the extent header contains the signature (commonly referred to as a magic number) 0xF30A, as well as other attributes describing the extent tree such as the current depth of the tree [62]. We highlight this because the extent magic number is the only searchable signature for Ext4 inodes, and if an inode opts not to use extents, then it has no signature to search for. Inodes may also be found outside the inode table. Ext4's *Journal* is a datastructure will include complete inodes of recently updated files.

One of the greatest differences between MFT records and inodes is that inodes do not contain *File Name* information, such as their filename or inode number (essentially an indexing number for their files or directories). This information is actually

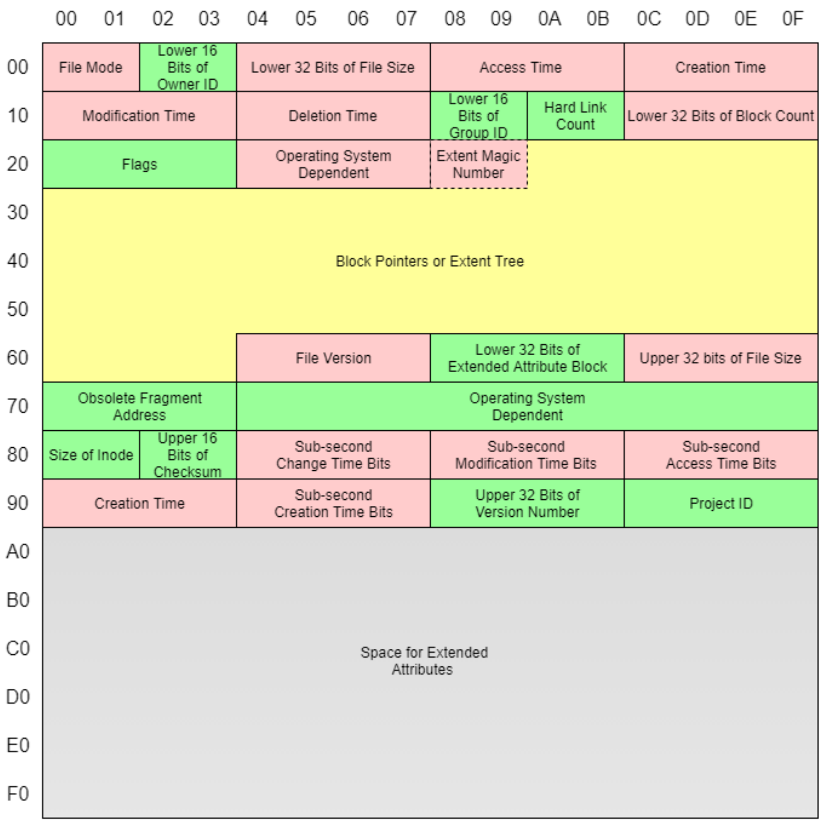

00
01
02
03
04
05
06
07
08
09
0A
0B
0C
0D
0E
0F
00
10
20
30
40
50
60
70
80
90
A0
B0
C0
D0
E0
F0
File Mode
Lower 16 Bits of Owner ID
Lower 32 Bits of File Size
Access Time
Creation Time
Modification Time
Deletion Time
Lower 16 Bits of Group ID
Hard Link Count
Lower 32 Bits of Block Count
Flags
Operating System Dependent
Extent Magic Number
Block Pointers or Extent Tree
File Version
Lower 32 Bits of Extended Attribute Block
Upper 32 bits of File Size
Obsolete Fragment Address
Operating System Dependent
Size of Inode
Upper 16 Bits of Checksum
Sub-second Change Time Bits
Sub-second Modification Time Bits
Sub-second Access Time Bits
Creation Time
Sub-second Creation Time Bits
Upper 32 Bits of Version Number
Project ID
Space for Extended Attributes

Figure 2.9: Diagram showing an Ext4 inode datastructure, with offsets from its start shown in hexadecimal [34, 62]. Attributes highlighted in pink denote attributes that we can utilize for either carving or verifying that data is indeed an inode. Attributes in yellow denote that we may use the information for further analysis. Attributes in green show the remaining data in the inode. The area in grey denotes where extended attributes of the inode may be stored, but this space is often used, and therefore can be a kind of slackspace used for data hiding [83].

stored in the directory data in structures called *directory entries*. Directory entries contain the name and inode number of the file or directory that is contained in the currently observed directory. The first two directory entries per directory always contain directory entries of the same "name" and length, which acts as a kind of header signature for directories. These directory entries refer to the "." directory (the current directory), as well as the ".." directory (the parent directory). Following these entries are the other files and directories stored in the directory, and they can be parsed out dependent upon the filesystem's rules for storing directory entries (the use of hash trees causes this task to become more difficult without the appropriate File System Category information) [170].

Ext4's policy for deleting files is less forensics friendly than NTFS. The actual file blocks remain on the volume, the inode will also remain, but it also has its filesize, extent header information, and its extent index nodes zeroed out. If the inode is not using extents, then the block pointers are zeroed out. Both the inode and file content are made unallocated [167].

2.4 Carving

When carving is being discussed, what is typically meant is *file carving*, the act of extracting files from the content space of a volume without help from the information from the File System or Metadata Categories. However, when we refer to "carving" we approach the term much more generally, in that we can carve for data other than files. We see carving as searching for specific byte sequences on a volume or disk image where the volume is seen as one long byte array T, and where the File System Category is mostly unused for the search. For instance, one can also carve a volume for inodes or MFT records, or even particular regular expressions that would yield email addresses or phone numbers. Due to the popularity of file carving, we describe the subject first.

2.4.1 File Carving

File carving is a useful method to recover files in unallocated or disk slack space, where the key mechanism to finding the files is to capitalize on the fact that specific file types use standardized data structures that often include both a header byte signature and a footer byte signature [155]. In the simplest case, a file may be unfragmented, and the range of blocks or clusters containing the header and footer signatures can be extracted from the volume. This extraction would contain the complete file. However, file carving isn't always so simple. Some files may not include a footer, so if we don't have access to a file's metadata record we cannot know how many blocks or clusters to extract even if we find the header. Another issue is in the case that a file is fragmented: how should one know where one part of a file ends and another part of the file begins [36]? In 2007, the Digital Forensics Research Workshop (DFRWS) even held a competition to improve file carving in fragmented scenarios [46]. A modern study by Van der Meer et al. [203] on the nature of fragmented files on NTFS, where 220 private Windows laptops were studied, that the average number of fragments for .png files was 2.8, the average number of fragments for .pdf files was 9.8, and the average number of fragments for mp4 files was 28.8.

2.4.2 Metadata and Other Kinds of Carving

Files are not the only useful data one can carve for. Hamm [11, 92] made the suggestion "Carve for records, not files." The suggestion is motivated by the fact that recovering the file itself may be unnecessary, as discovering logs of user history can be just as or more revealing about a user's behavior. Hamm also states that carving for records is useful because file carving is not effective for a number of different filetypes (for example, .txt files do not have a signature). Other types of metadata such as filesystem metadata records and social security numbers may be carved for as well. Metadata and feature carving are futher discussed in Sections 3.2.2 and 3.2.3.

Given the prerequisite information provided by this chapter, the readers should have a sufficient background to understand the context of our research papers and contributions.

Chapter 3

Related Work

In this chapter, we look at work related to exact and approximate string matching, string matching in the context of digital forensics, file carving, metadata carving, and other methods to help lighten the digital forensics backlog.

3.1 String Matching

In this section, let m refer to the length of the search pattern, n refer to the length of the text T, and k refer to the number of allowed edit operations. We first describe common string matching algorithms found in digital forensics and then focus specifically on approximate string matching algorithms.

3.1.1 Common String Matching Algorithms

The Boyer-Moore [29] exact string matching algorithm was one of the first non-naive and fast string matching algorithms, and even runs in sublinear time on average (less than $O(n)$), with a worst case time complexity of $O(mn)$. The ability to run in sublinear time is due to the fact that the algorithm will skip characters if they do not appear to correspond with the search pattern. Boyer-Moore is still relevant today in digital forensics. For example, it is used within the popular file carving tool Scalpel [181, 212] to identify file headers and footers.

Regular expression matching searches in $O(n)$ or $O(mn)$ dependent upon the type of automaton construction [143]. It is also a function within most digital forensic suites [94], and is typically how digital forensic analysts search for data that has an expected pattern such as license plate numbers, telephone numbers, credit-card numbers, or common misspellings.

Digital forensics suites also typically have the option to search for multiple keywords at once, and this is handled by an old but still very relevant algorithm, Aho-Corasick [5]. It allows for *multi-pattern matching* in linear time ($O(n)$).

3.1.2 Edit Distance Approximate String Matching Algorithms

The first approximate string matching algorithm was developed by Sellers [191], which directly utilized a dynamic programming matrix[1] [205], and ran in $O(mn)$ time [138]. Ukkonen [200] suggested that rather than using a dynamic programming matrix, a deterministic finite automaton (DFA) could be used, but noted that such constructions would require a massive number of states [138].[2] It was not until Baeza-Yates [14] introduced the concept of bit-parallelism for simulating nondeterministic finite automata (NFA) that there was a surge in bit-parallel approximate string matching research.

The first two implementations of bit-parallel algorithms simulating an edit distance NFA (see Figure 2.2) were invented by Wu and Manber [209] (Shift-And) and Baeza-Yates et al. [13] (Shift-Or) where they run in $O(\lceil m/w \rceil nk)$ time. The wider popularization of this class of algorithm came about with the implementation of *agrep*[3] by Wu and Manber [208], an approximate string matching *grep* command line program that handled approximate string matching and regular expressions search. A number of improvements to bit-parallel string matching algorithms have since been made.

The *Bit Parallel Diagonalization* algorithm [12] is similar to the edit distance NFA, except rather than have bit-vectors represent rows, they represent the diagonals of the automaton. The idea is that by using the diagonals of the NFA, any active state in a diagonal immediately causes the subsequent states in the diagonal to become active due to ϵ-transitions. In this way, a simulation may be done where the diagonals may be successfully calculated in parallel, and the worst case running time is $O(\lceil (m-k) \cdot (k+2) \rceil / w)n)$. This algorithm was later optimized to $O(\lceil (m-k) \cdot (k+1) \rceil / w)n)$ worst case running time by Hyrro [100].

One of the fastest bit-parallel approximate string matching algorithms, the "Bit Parallel Matrix", runs on average in $O(\lceil k/w \rceil n)$ and in worst case $O(\lceil m/w \rceil n)$ [136]. While it uses bit parallelism, it does not simulate an NFA, but a dynamic programming matrix, where bit-vectors represent the vertical, horizontal, and diagonal differences between cells. Boolean values can represent the differences between cells since adjacent cells in a dynamic programming matrix do not exceed

[1] Dynamic programming can be used to create a matrix in order to calculate the edit distance between two strings.

[2] Eventually Schulz and Mihov [190] would create the *Levenshtein Automata*, a fast DFA implementation for edit distance string matching.

[3] https://www.gnu.org/savannah-checkouts/gnu/grep/manual/grep.html

1. While the Bit Parallel Matrix algorithm has a better worst case time complexity, it is considered to be less flexible than the Bit Parallel Diagonalization algorithm when searching for complex patterns or using different distance functions [142].

The last relevant string matching research we discuss is work done by Holub [97], who noted that bit-parallelism can be used not only to simulate edit distance NFAs, but they can simulate other NFAs as well by omitting transitions. This insight more or less led to our concept of applying edit operation constraints to edit distance NFAs. Holub also wrote the most recent review of applying finite automata to string matching [98]. A comprehensive survey on the subject of bit-parallel string matching can be found in the work by Faro and Lecroq [65].

Recently, bit-parallel approximate string matching has received less attention, but there are still improvements being made to the string matching field. Modern trends include the study of managing large volumes of data and repetitive strings. A frequently researched method compresses the index of text T with algorithms such as the Burrows-Wheeler [32] transform or Lempel-Ziv [214] compression, which allows the compressed index to be searched for patterns [140, 186].

3.2 String Matching Research Applied to Digital Forensics

In this section we focus on past research that has attempted to improve string matching algorithms within the context of digital forensics and investigations. There are two major challenges often addressed when it comes to digital forensics string matching research. The first challenge is how can we reduce processing times for string matching in digital investigations. The second challenge is validating string matching functionality in digital forensics tools, and identifying under which conditions they fail. A lesser addressed challenge, which is the one we address, is understanding how to limit false positive rates for approximate string matching (or fuzzy searching) for digital forensics.

3.2.1 File Carving

Most novel research into string matching with respect to digital forensics focuses on how to improve file carving and regular expression search.

One of the first widely known file carving tools was *Foremost* [111]. It was developed by the United States Air Force Office of Special Investigations, where the tool searched for file header and footer signatures [155]. Richard III and Roussev [181] improved upon Foremost, thus giving birth to the tool *Scalpel*. The creation of Scalpel focused on implementing the following principles: reduce search time by using a fast string matching algorithm, disk reads to RAM should be minimized, and writes to disk should also be minimized. Richard and Roussev provide the details that Scalpel in its worst case scenario for carving individual header/footer byte sequences runs in $2T$ time [181].[4] Scalpel's methodology of significantly reduced memory-to-memory copies and disk I/O produced a much faster carver than Foremost, and also allowed for arbitrarily large files to be carved despite available RAM [181].

There are several items that need to be discussed when analyzing Scalpel's initial performance. The first item is that this implementation may carve out file fragments (containing header or footer fragments) but it does not automate the reassembly of fragmented files. The second item, while not mentioned directly in the paper, is that Scalpel uses a modified version of the Boyer-Moore algorithm [17, 29, 127, 212] that runs on average in sublinear time, and in worst case runs in $O(mn)$ time, where m is the length of the keyword and n is the length of the byte sequence being searched. Notably, Boyer-Moore is a single keyword exact string matching algorithm, so the runtimes being assessed do not take into account searching for the entire database of file headers and footers that a digital investigator may have to search for. Amending this issue with multi-pattern string matching would later become a subject of other research. The initial versions of Scalpel performed full file carving by default, but later Richard et al. [180] created the concept of *in-place file carving*, which only inspects file content without copying it to the disk. This adjustment was necessary as file carving has many false positive "files", thus occupying a substantial amount of disk space. Technically speaking, we perform *in-place filesystem metadata record carving*, since we only extract features from these data structures, and do not write actual records to disk. Even prior to the release of Scalpel, Roussev and Richard III began studying how *distributed processing* could benefit digital forensics search tasks, especially with increasingly large datasets to analyze [184].

The next carving innovations focused on the fact, as Bayne et al. states, "pattern matching is a trivially-parallelisable problem" [17]. This was first investigated by Marziale et al. [127], wherein the focus of the research was to modify Scalpel to support multi-threaded processing using GPUs and multi-core CPUs. According to their experiments, they came to the conclusion that any task used in digital forensics software that uses binary string searches can greatly increase their search speed by utilizing multi-threading.

Zha and Sahni [212] addressed Scalpel's use of single pattern string matching, and that header-footer carving speed could be significantly increased if a multi-pattern string matching algorithm was used such as the Multi-pattern Boyer-Moore

[4]In terms of time complexity, it seems the Boyer-Moore search time complexity is assumed to be in its average case where it runs in linear or sublinear time, so while Scalpel requires two disk reads in the worst case, the act of carving for a single header or footer is would take $O(T)$.

or Aho-corasick [5] algorithms. The motivation is that searching for matching header-footer patterns will run in $O(hn)$ for single pattern matching algorithms, where h is the number of header-footer patterns needed to be matched. Multi-pattern search, on the other hand, only runs in linear time. They named their implementation *FastScalpel*, and they additionally managed to create a setting that utilized multi-threaded multi-pattern string matching. While the use of multiple cores improved their search times when using exact matching Boyer-Moore and many search patterns, multiple cores did not significantly improve multi-pattern string matching. At the time, they reasoned that the bottleneck was not due to the time required to search for headers and footers, but the time required to read from disk. This hypothesis was later disproved by the authors, as they successfully increased performance of multi-threaded multi-pattern search [213]. Ultimately, using FastScalpel was 17 times as fast as Scalpel 1.6 when searching for 48 signatures. FastScalpel can also hide the search time entirely by the read time when performing in-place file carving by utilizing asynchronous disk reads. They suggest that using this method will perform in-place file carving in about the same time as it takes to read the disk.

A similar work by Bayne et al. [17] created the *OpenForensics* framework for using the Parallel Failureless Aho-Corasick (PFAC) algorithm [121] utilizing multi-threading in CPUs and GPUs to increase the speed of file carving. The key idea is that the PFAC algorithm is designed for massively parallel execution, where threads are asynchronously run per each logical core on a CPU, and each thread can “individually request a new block of data when they are freed” [16]. Their experiments showed that OpenForensics’ read speed (measured in MiBps) was at times close to the theoretical read spead of storage devices, with room for improvement for searching for many patterns simultaneously.

Liao [116] conducted a survey of software-based (as opposed to hardware implemented) string matching algorithms applied to digital forensics, wherein Liao selected ten common string matching algorithms and compared their results for carving file headers and footers. They provide a comprehensive comparison in the different algorithms’ preprocessing and searching time complexity, as well as the timing results for each of the search algorithms to search for 12 different header-footer pairs over two different disk images. Their results showed that the Shift-Or [13] algorithm and the Rabin-Karp [104] outperformed the modified Boyer-Moore algorithm used by Scalpel [181]

File Recovery and Alternative Kinds of File Carving

Here we mention some definitions recently introduced for the task of file recovery, as well as non-sequential file-carving methodologies.

Casey et al. [38] produced a common vocabulary for file recovery results, with the purpose that software developers and digital forensics professionals can understand research results with as little confusion as possible. We use these common file recovery classifications in Paper 4 [172]. When evaluating recovered data, Casey et al. look at the following features. Was the name of the file recovered? Was the metadata of the file recovered? Was the content of the file recovered? In the case that an MFT record was recovered that contained a resident file, we could recover its filename, metadata, and content. This classification is known as *fully recovered*. However, if the file was non-resident, the content is only potentially recovered. Our work to recover MFT records typically falls under *fully recovered*, *potentially recovered*, or even *partially recovered* in the cases where we cannot discover the content. Using typical file carving techniques that ignore the name and metadata of the file will be classified as *content recovered*. Recovering an inode leads to an interesting case where the filename is not discovered, and the content is potentially discovered.

The methods of file carving so far have been fairly (and often literally) straightforward. Other approaches to file carving exist that attempt to study how to triage (see Section 3.3.2) disk images when looking for file content. Karresand et al. [105] studied the cluster allocation algorithm used in NTFS by recording where user data is allocated to the disk after the operating system had been installed. The experiment utilized 32 newly installed Windows 7+ operating systems, where files were randomly created, deleted, expanded, or shrank on a Windows Virtual Machine (VM). By shutting down the VM and inspecting the $Bitmap, they could see which clusters' allocation status had changed for that particular file operation. Their results show that content is most likely to be stored right after the initial 12.5% of the disk reserved for the MFT (to prevent fragmentation) [130]. Typically, the probability of storing file content decreases as one goes further through the partition. Later work by Karresand et al. [106] showed that Windows' allocation strategy is to fill "holes in the already written area instead of claiming the unused space at the end of a partition".

Another approach for providing a triage method to file recovery was performed by Gladyshev and James [81]. The authors see file carving as a *decision theory problem*, akin to a battleship game. When playing one-dimensional battlship, one would not start at one end and sequentially go to the opposite end. Using previously known information about content distribution on the disk, sections of the disk known to have data are visited and sections of the disk typically known for not containing data are skipped. For instance, Gladyshev and James' algorithm *DECA* (decision-theoretic carving) takes into account estimates of the distribution of JPEG data on a disk.

3.2.2 Feature Carving

Garfinkel first proposed the concept of *forensic feature extraction* [73], where the goal is not to carve out files or other filesystem data structures, but "pseudo-unique identifiers". These identifiers may be credit-card number, email addresses, dates, etc. Extracting these features from a disk image may reveal who the particular image belongs to, and by comparing features and their statistics from different images cross-correlation between drives may be performed. This was later implemented in the *bulk_extractor* [30, 76], which has the added benefit of not being filesystem specific. The purpose of the tool is to provide a means of triage, unearthing potential clues for investigators as to who or what to investigate further in a case. Interestingly, the bulk_extractor employs a context sensitive stop list, where certain keywords are ignored if they appear in expected operating system files. This reduces a large potential of irrelevant or misleading features.

The bulk_extractor also utilizes parallelism to speed up analysis, as pages from the analyzed disk read into memory can be analyzed independently. Garfinkel called this an "embarrassingly parallel problem" [76]. Version 1.2 of the bulk_extractor used FLEX[5] for regular expression search, and Garfinkel notes how it offers high speed and a low false positive rate [76], but it is unclear how much modern versions of the tool still rely upon it.

Stewart and Uckelman [196] designed and implemented *lightgrep*,[6] a multi-pattern regular expression grep-like search tool with a focus on speeding up regular expression matching for searching large quantities of data. Like most other string matching algorithms, lightgrep simulates nondeterministic finite automata. Of note, the tool supports multi-pattern search that runs in worst case $O(mn)$. Stewart and Uckelman compared their tools to Encase's search ability, where the test included searching for a set of regular expressions on a 32 GB disk image. Lightgrep was faster by a factor of 10 in all cases. However, this study is from 2011, so we are uncertain to how lightgrep compares to modern digital forensics tools. Stewart and Uckelman [197] also conducted research on how to explicitly utilize multi-pattern string matching for various text encodings, eventually supporting all Unicode characters and over 180 different encodings. Version 1.4 of the bulk_extractor[7] [76] uses lightgrep as a possible method for searching for regular expressions [30]. The documentation does not explicitly say if parallel processing and lightgrep can be used simultaneously.

[5]https://github.com/westes/flex
[6]https://github.com/jonstewart/liblightgrep
[7]https://github.com/simsong/bulk_extractor

3.2.3 Metadata Carving

A far more niche field of carving is carving for filesystem metadata itself. Metadata may directly give an account of how a user was utilizing a computer over a period of time, and may also allow for full file recovery.

One of the first methods for searching for filesystem metadata was conducted by Mueller [135], where he directly searched for timestamps as strings in unallocated space. As NTFS records its timestamps as little-endian 8-byte sequences that count the nanoseconds passed since January 1, 1601, finding 8-byte sequences that fall within plausible datetime ranges is a reasonable way to encounter filesystem metadata. Similar to our later work [151][172], Mueller notes the contiguous nature of four timestamps per MFT record. To search for such timestamps he created a plug-in for *Encase* called *EnScript* that could search for NTFS timestamps in a particular date range using a *grep* expression, additionally only allowing timestamp hits to occur if two or more contiguous timestamps were found. Finding contiguous timestamps should reduce the false positive rate for finding MFT records.

McCash also noted upon how repetitive strings (such as timestamps) found in unallocated space were actually signs of “old MFT entries, INDX structures, or registry keys or values” [128]. Thus, he concluded that by first finding timestamps in an NTFS partition he could work backwards to find the embodying metadata data-structures. Using Mueller’s EnScript, one can find timestamps and then search the surrounding data for metadata signatures or flags which may verify the metadata structure. This approach is similar to our NTFS specific parser of our Generic Metadata Time Carving Papers 3 and 4 [151][172]. One of his final points is that he finds “it somewhat disappointing that many forensic professionals seem unaware of the plethora of redundant timestamps that are present in NTFS ” [128]. This is a subject we would later capitalize on for various filesystems, not just NTFS.

Pomeranz [168] was the first to describe a method of how to find files and filesystem metadata using dynamic signatures based on the context of a disk’s byte sequences. If a file is deleted in Ext3 its block pointers will be deleted as well, so searching for this file’s particular inode isn’t so useful for file carving. But for large files, one can search for a file’s *indirect block*, filesystem metadata that contains up to 1024 4-byte block pointers to the other locations of at least part of the file. Assuming little to no fragmentation of the file and 4096 byte-sized blocks, indirect blocks begin at the 13th block of a file. The first pointer in the indirect block should point to the very next block. So by checking if the first four bytes of the Nth block on the volume is equal to the value $N + 1$, you have potentially found an indirect block. This method can be used to search for any indirect block in a file, again assuming there are no fragmentation issues. If the first indirect block of the file is found, one can also go back 12 blocks to locate the beginning of the file.

Dewald and Seufert [55] focused on carving inodes from Ext4, where File System Content such as the superblock or group descriptor table are inaccessible. They note that their approach is similar to that of Pomeranz [168], as they search for complex string patterns. This is necessary as inodes have no magic number, except in the cases that the inodes have extents. Once an inode is found, and assuming that this copy of the inode has not had its extents or block pointers wiped, metadata based data retrieval can be utilized. For carving inodes, they tested several different signature types and their combinations, including plausible timestamp intervals and that the interrelations between an inode's timestamps should be sensible. For example, if a file was deleted then the deletion time should be greater than the modified time. Other possible signatures test if the candidate timestamp falls within a user defined time frame, the first two bytes of a potential inode are the flags for a file or directory, or if the identified access rights are common. Up to this point, we have described the *content mode* of their tool, but they also have a metadata mode that reconstructs the directory tree, which allows them to connect metadata content to file content.

Applying their tool's metadata mode to a non-reformatted 16 GB Ubuntu image with 201269 inodes in the inode table, they achieved True Negative Rates anywhere from 0.004% when only using the extent header signature, to 47.6% when checking if the number of link counts is larger than 0. This statistic essentially refers to the probability that their tool misidentifies an inode as not an inode. However, this is only applied to the contents *within* the inode table. Of the potential inodes discovered within the inode table, the timestamp consistency signature yielded 8213 false positives, plausible timestamp dates signatures yielded 7140 false positives, only 2 false positives for a signature of the number of hard links, and the extent flag and header signatures led to 8074 false positives [55]. However, outside the inode table they often encountered millions of potential inodes, and even inside the inode table they at times encountered billions of potential inodes that were not properly aligned within the inode table. These statistics imply that their tools may potentially have high recall, but very low precision. They note the most *restrictive* signatures (those that encountered the least amount of potential inodes) were searches with common access rights, timestamp intervals, and extent headers.

Dewald and Seufert's [55] content mode, which is meant to identify inodes, only requires the blocksize as a starting parameter. The metadata mode is used to discover inodes' File Name content, allowing for this information to be connected to their respective inodes. Their method for connecting File Name content to

Metadata content requires prior knowledge of the beginning and ending offsets of the inode table, the number of inodes per block group, the number of of block groups, and flex group size. Both content mode and metadata mode took nearly 2.5 minutes to run their analyses on an image greater than 5 GB.

A follow-up work by Plum and Dewald [163] researched file recovery from APFS, where a similar filesystem metadata carving methodology was utilized. Their experiments compared results from their tool to *photorec*[8] for carving photographs from disk images from 2 GB to 500 GB in size, and their tool recovered 42.6% of the files while *photorec* only recovered about 8.1%.

Much of the work done with respect to filesystem metadata carving is non-academic. We would be remiss to not mention the open-source work done by Schicht,[9] where he has created a large number of tools to extract specific metadata structures primarily from NTFS. This includes an MFT parser, LogFile parser, INDEX record parser, among others.

3.2.4 String Matching Validation Research in Digital Forensics

The purpose of this subsection is to discuss how past researchers have suggested to validate string matching functionality in digital forensics tools.

Lyle [125] brought up the issue of forensic tools often not having published error rates, therefore appearing to fail the Daubert Criteria [1, 189]. In particular, he noted that string searching tools for digital forensics appeared to have few standards in how they are implemented for digital forensics. Of potential sources of errors, he notes the possibility of various encodings and diacritical marks (such as accents or tildes). Furthermore, he states errors are unlikely to occur due to some random event, but are more likely to occur due to the design limitations of the search tool. In the cases of false positives, they are easy to detect simply by checking if the search hit matches the search keyword, and he therefore suggests having an error rate for string matching “may be of little value”, and instead suggests having a clear outline of how the search method works and its limitations. His work does not cover approximate string matching methods or identifying dynamic signatures, where having an error rate may be more interesting since false positives are not quite as trivial as the exact matching case.

Despite these concerns, Computer Forensics Tool Testing Program (CFTT) [146] (a joint project between NIST, DHS, and other agencies) has attempted to establish a methodology for testing computer forensic software tools, including string searching [147]. The battery of string matching tests are tested on a digital forensics

[8]https://www.cgsecurity.org/wiki/PhotoRec
[9]https://github.com/jschicht?tab=repositories

tool (such as Encase or Autopsy), wherein the tool's search engine and search string parameters are evaluated against a standard dataset with a known set of strings. For instance, a tool's ability to conduct regular expression search, logical AND search (two search strings must be in a file), stemming search, Unicode text search, and non-English character search are all tested. The results of these tests indicate in which circumstances these tools can be expected to fail; where false negatives occur. Results for popular tools such as Axiom, Encase, and FTK have already been compiled.[10]

Related to the CFTT research [146] is the search function ontology by Guo et al. [88], where the purpose was to map out the functionality, domain, and target (keyword) of a searching function of a forensic tool to consider how it should be validated. Ideally, their approach could be applied to any kind of digital forensics function. Their ontologies have a large overlap with the work produced by the CFTT, as they wish to possible test keywords with different encodings, regular expression search, different search modes (Boolean, Soundex, proximity searching, etc), and searching for content in both allocated and unallocated space. Beckett et al. [18] made a similar onotology.

3.2.5 Approximate String Matching in Digital Forensics

Even though we research approximate string matching algorithms, we acknowledge that they do not often appear in the digital forensics literature. Approximate string matching (or fuzzy matching as it is sometimes referred to) appeared to be more widely present in the digital forensics zeitgeist nearly two decades ago, as shown in the NIST CFTT draft report as an optional requirement for standardized testing of digital forensics tools' string matching capabilities [144]. The later versions of the document removed fuzzy matching from their standardized tests altogether [147]. However, some digital forensics and investigative tools still use approximate string matching, such as Intella [102] for e-mail e-discovery. Other digital forensics tools such as FTK and Autopsy do not currently appear to support approximate string matching, but they both use search engines that do, dtSearch [3] and Elasticsearch [61] respectively.

One of the first works to investigate the use of approximate string matching in digital forensics, and utilizing very similar methods to ours, was by Chitrakar and Petrovic [40]. They used a constrained edit distance (CED) row-wise bit-parallel NFA algorithm that allowed the user to control the total number of Sankoff *indels* [187] (insertion or deletion edit operations) accepted by the automaton, and they applied their algorithm for searching spam email keywords. The NFA simulation however used integer counters to keep track of edit operations which oc-

[10]https://www.dhs.gov/publication/st-string-search-tool

curred, meaning it was not a full NFA simulation and thus not gaining all the benefits of bit-parallelism. They also implemented a dynamic programming approximate string matching algorithm, based on the work by Sankoff, and applied the algorithm to the same set of keywords. They hypothesized that the bit-parallel implementation would run faster than the dynamic programming implementation, which was true if the number of indels was greater than the number of substitutions. No measure of accuracy, precision, or recall was given.

A follow-up work by Chitrakar and Petrovic [41] introduced constraints on the total number of specific edit operations to a row-wise bit-parallelism approximate string matching algorithm, and applied the algorithm to searching *Snort* intrusion detection rules. They posit that attacks with minor modifications to existing Snort rules (as small as a single byte) will then be able to bypass a rule based intrusion detection system. Their hypothesis was that they could obtain a lower false positive rate than classic Wu-Manber [209] edit distance based approximate string matching algorithms given that they had some prerequisite knowledge about the probability of edit operations that may occur on the intrusion detection rules. They constructed four different scenarios with different probabilities of edit-operation distortions applied to the searched Snort string, where they tested their new algorithm with varying maximum values of the different edit operations. In the majority of cases, the row-wise bit-parallel search matched fewer false positives.

Chitrakar and Petrovic's next work was similar to their previous work, where rather than placing constraints on the number of specific edit operations, they placed constraints on allowing different types of edit operations, and applied the algorithm to detecting modified SQL injection patterns [43]. While still allowing for some k number of allowed errors, they essentially removed particular edit-operation transitions from the NFA that they wished to avoid. This enabled them, for example, to search for strings tolerating k errors, but only from substitution and deletion edit operations. Furthermore, this algorithm is a true simulation of an NFA. Assuming the defender has prior knowledge of how the SQL strings may be modified, they showed that the constrained edit distance algorithm can indeed encounter less false positives than by using the edit distance.

Nrgrep by Navarro [139], created in 2001, is similar to Chitrakar and Petrovic's previous work [43] as it allows users to set an edit distance threshold, transpositions (swapping neighboring characters), and allows the user to omit certain edit operations when applying approximate string matching. Chitrakar and Petrovic note the difference between the algorithms being that theirs is based upon the Wu-Manber [209] implementation, while Navarro's utilizes Backward Nondeterministic DAWG (Directed Acyclic Word Graph) matching [141].

The algorithms presented in the previous papers most closely resemble our novel algorithm in Paper 2 [174], as constraints are applied through disabling transitions in an NFA. What distinguishes our algorithm from Chitraker and Petrovic's algorithm is that their NFAs have a single row per error, whereas we have a row per error combination, which allows us to define edit operation constraints as any combination of edit operations up to an edit distance of 2 [174].[11]

3.3 Other Approaches to Minimize the Forensic Backlog

The majority of related work we have discussed so far has focused on either improving or verifying string search and file carving, but there are many other works which have attempted to confront the digital forensics backlog by other means. For instance, there has been much research devoted to applying machine learning, data reduction, and triage methods to digital forensics. As this research essentially refers a large portion of the body of digital forensics literature, we attempt to highlight seminal works, and especially those which have emphasis on some category of text analysis. We first discuss relevant machine learning research, followed by data reduction research, and triage research.

3.3.1 Machine Learning

Beebe and Clark [20] foresaw the difficulties in handling large datasets in 2005 (at the time, a terabyte was considered to be massive), and began considering how data mining could be used to speed up the digital forensics process. When the authors refer to *data mining*, they are referring to the process of retrieving information that utilizes *machine learning* or *artificial intelligence*. As stated by Beebe and Clark, manually reviewing keyword search hits could take days, and they wished to utilize more sophisticated learning methods that could potentially capitalize on the latent knowledge within the investigated data.

Beebe and Clark [20] provide four different examples of popular learning tasks. In terms of machine learning, one of the most popular kinds of models are those used for *classification*. Such models are trained on large sets of *training data* that have class labels (known as supervised learning), and given some unknown sample the model predicts which class the sample belongs to. Modern popular models for performing such tasks include deep neural networks, and can be used for things like explicit image detection [162] or age detection [8]. Another popular machine learning task is *clustering*, the task of being trained on an unlabeled dataset (known as unsupervised learning), and will partition the data into clusters based on sample similarity. For example, after a keyword search one could cluster the resulting

[11] As of now, the construction of our NFA for constrained edit distance cannot be automated, but it can be extended to support higher edit distances.

returned documents in order to detect similar documents [23]. A third task mentioned by the authors is *regression*, the task of fitting a mathematical function to numerical data over time, which may allow for predictions of future values. While this is a common task in the field of machine learning (for example, predicting stock prices or housing markets), the method has seen little use in the field of digital forensics. One of the last tasks Beebe and Clark mention is *association rule mining*. This task is concerned with identifying items which commonly appear together in a dataset. An example of this is to identify names or places that commonly appear together over a series of different documents [6]. Beebe and Clark also mentioned the desire to use *Natural Language Processing* (NLP) as a means to enhance information retrieval. A similar work by Pollitt and Whitledge [165] also suggested data mining to be a potential solution to the forensic backlog.

Future works by Beebe focused on realizing intelligent digital forensics, and began with considering a new process model for text string searching [21]. The crux of the model was that the initial phases of keyword search was kept, but to gain knowledge from the hit results of the search would be organized in some fashion. Later studies by Beebe et al. [24, 25] would apply clustering techniques to the hit results of a keyword search (including k-Means [53], Latent Dirichlet Allocation [28, 52], and Self Organizing Maps [109]). As stated by Beebe and Liu [25], the effectiveness of information retrieval techniques are typically measured in precision and recall, but they measured their results using the concept of the *average precision*, which takes into account the order in which relevant hits are viewed. Testing on the M57 patents dataset [72], clustering yielded a 15 times increase in average precision over the non-clustered results, where over 6000 relevant search hits were located after only reviewing 0.5% of the total search results. Noel and Peterson [148] conducted a similar study applying Latent Dirichlet Allocation to multi-disk search, and they noted the benefits in being able to find documents topically similar to their search keyword, without actually containing that particular word. However, they warned against replacing traditional search with such clustering search, as the time to train the topic models could take hours.

Franke and Srihari [71] also suggested the merging of machine learning and digital forensics, coining the term *Computational Forensics*, which has a wider remit in applications of computational methods than previous suggestions. In addition to the previously mentioned methods, they also suggest using computer vision, computer graphics, and robotics. However, they also note that the use of statistical algorithms are dominant, where it is possible to calculate error rates.

Another suggested use of machine learning being applied to digital forensics has been Natural Language Processing, which traditionally came from the field of linguistics, but whose methods have greatly been improved by statistical models

[137]. One of the most comprehensive studies on applying NLP to digital investigations is by former director of the FBI's Regional Computer Forensic Laboratory Program, Pollitt [164]. Though the dissertation is highly theoretical, it provides a professional's perspective on applications of NLP technologies. One of the primary concept's of Pollitt's work is to automate and extract the "narrative" from forensic evidence. In his experiments, he applied a number of different NLP techniques to the Enron email dataset [47], including Named Entity Recognition, Sentiment Analysis, and Summarization using *NTLK*[12] [122]. Named Entity Recognition, the ability to automate the process of extracting proper nouns from text, may be the most popularly applied NLP method to digital forensics [39, 123, 210], especially in combination with association rule mining [6, 103, 124, 202, 211].

3.3.2 Data Reduction and Triage

The concept of *data reduction* is vague, the term itself appearing in digital forensics literature as early as 2006 [73], being implied but not explicitly mentioned by Beebe in 2009 [19], and finally appearing more as a tangible concept by Quick and Choo in 2014 [175]. As far as we can tell, there is no common standard definition of the term. Liebler et al. defined it as "to separate the wheat from the chaff, e.g., to filter in forensically relevant data" [117].

Pollitt generally defines *triage* as "a way to maximize the use of scarce resources by prioritization" [166]. However, Pollitt also notes that in the digital forensic context triage can be done in several ways. The data being acquired can be reduced, certain areas of the disk or folders can be searched for based on past experience, or specific kinds of data can be searched for [166]. Thus, we can say that data reduction may be a method to conduct triage, but not vice versa.

Quick and Choo [175, 177] provide methods for conducting triage by way of data reduction. Their method of *Data Reduction by Selective Imaging* (DRbSI)[177] does not acquire the full image from a physical disk, but only selects relevant filetypes or datastructures to be examined and analyzed. For instance, the user may only choose to recover photographic images, or filesystem files or datastructures such as the $MFT or inode table. Videos may also be reduced in size by converting videos into a series of photographic thumbnails saved in a single JPEG file, allowing for fast review. Their results reduced the median time required to collect and process the logical images to 14 minutes, as compared to the median time of about 8 hours to process a complete disk image. Quick and Choo also suggest that data mining and the use of open or closed source data can be used on the selectively extracted data [175]. They make an important point that this data reduction methodology is not meant to replace the typical forensic process, but to be performed

[12]https://www.nltk.org/

prior or in conjunction with the typical process. Other forms of triage applied to file carving have been mentioned in Section 3.2.1 [81, 105].

Data reduction can also be used to make forensic processes faster, without necessarily conducting triage. This usually involves applying cryptographic hashes or types of compression. A common practice in digital forensics is to either maintain or use a hash database [77], in which illegal files such as malware or child pornography first have their cryptographic hash calculated, and then stored in a database typically as a unique 128, 256, or 512 byte value [182]. This way, when coming across some storage media as evidence, investigators can automate the process of testing if the media contains contraband material by hashing files on the device and checking for matches on the hash database. This also prevents the investigator from actually running malicious programs or seeing pornographic files. This standard procedure poses several research problems and has produced several derivatives.

A primary problem with hash based file matching is when there is a need to identify file fragments or nearly identical files. If a file is modified by a single byte, a hash of a file is completely changed. To cope with this limitation, researchers developed *approximate hash based matching*, also known as “approximate matching” [31] (not to be confused with approximate string matching). The principle of this type of hashing to is produce the same *similarity digest* for two minorly different files (bytestreams, documents, etc). While there are many different types of approximate hash based matching algorithms and tools (see Harichandran et al. [96] for more details), the two most popular tools appear to be *ssdeep*[13] by Kornblum [110], and sdhash[14] by Roussev [183].

An alternative solution for dealing with similar or fragmented files was produced by Garfinkel and McCarrin [77], *hashdb*.[15] This algorithm is used in the bulk_extractor to create a sector level hash database. So rather than match files on their full content, one can match files based on the hashes of uncommon sectors of a file (a common sector of a file may include an “empty” sector).

This has only been a limited discussion of data reduction, triage, and other methods such as machine learning to attempt to help resolve the digital forensics backlog.

[13]https://ssdeep-project.github.io/ssdeep/index.html
[14]https://github.com/sdhash/sdhash
[15]https://github.com/NPS-DEEP/hashdb

Chapter 4

Summaries and Main Contributions of Publications

The purpose of this chapter is to summarize the work done for each research paper in the dissertation, highlight their contributions, and also draw connections between the works. As the reader should be equipped with the background knowledge of the previous chapters, we discuss our contributions freely.

4.1 Paper 1: On Application of Constrained Edit Distance Algorithms to Cryptanalysis and Digital Forensics

The goal of this paper was to conduct a survey on the theory behind the constrained edit distance (CED), the theory behind bit-parallel approximate string matching, and identify where such techniques have been applied in cryptanalysis[1] and digital forensics.

We first took a deep dive into the theory regarding the CED, and the effects of applying constraints on edit operations when comparing strings. We looked at specific methods for calculating the constrained edit distance, such as dynamic programming [152], and the methods for approximate string matching using automata theory [142] simulated with bit-parallelism [14]. The knowledge regarding bit-parallel nondeterministic finite automaton (NFA) simulations for string matching built a basis for Paper 2 [174], which includes the development of our novel approximate string matching algorithm.

Past applications of the CED and it's associated algorithms for cryptanalysis focus

[1]Cryptanalysis refers to "breaking" a cipher and identifying the key used for encryption.

on recovering the key of stream ciphers. In particular, stream ciphers combining output from Linear Feedback Shift Registers (LFSR)[2] are vulnerable to *correlation attacks*, where the attack tries to statistical identify correlations between the individual or combinations of LFSR bitstream output and the cipher's keystream output [129].[3] The correlation probability can be calculated utilizing the Hamming distance [93]. However this only works if the lengths of the bitstreams are equal, unlike stream ciphers with irregularly clocked LFSRs, thus the interest in utilizing the edit distance and constrained edit distance for *generalized correlation attacks*. As particular categories of irregular clocked LFSRs are restricted in how often they can perform deletion edit operations, Petrovic and Golic [161] applied constraints on consecutive deletions for improving this category of cryptanalysis. Paper 1 is our only paper which discusses cryptology.

For digital forensics, most applications of the constrained edit distance (CED) were primarily by Petrovic and Chitrakar [40, 41, 43, 159, 160]. Such applications include the use of the CED as a metric for a digital forensic triage stage, wherein large fragments of raw data have their CED calculated between themselves and a desired search string [159]. This way, only fragments with the shortest CED are further investigated. A similar solution was also applied to selecting relevant Snort signature rules [160]. More recent studies had focused on applying row-wise bit-parallel based NFA simulations for constrained approximate search to digital forensic contexts. The majority of the work focused on how different edit constraints could be applied to string matching NFA, and applied the methods to spam filtering or intrusion detection [40, 41, 43, 42]. In all cases, it is required that the user of the algorithms has prior knowledge of the probabilities of edit operations for particular attack signatures or spam messages in order to see a reduction in false positives.

Lastly, we noted how we may apply Oommen's CED formalisms [152, 153] to approximate matching NFA.

4.1.1 Contributions of Paper 1

To our knowledge, Paper 1 is the only research paper that reviews where the constrained edit distance has been applied within information security fields. A contribution the paper makes is noting how edit constraints can be applied to nondeterministic finite automata (NFA) for approximate string matching by modifying or removing the NFA's transitions. This reduces the number of ways one string

[2]Linear Feedback Shift Registers are simple pseudo-random number generators that are often used in combination to make more complex pseudo-random number generators.

[3]Keystream output is obtainable through known plaintext attacks, where by XORing the ciphertext and the plaintext, one obtains the keystream.

may be transformed into another. Prior to this, the constrained edit distance was always implemented into an NFA by using integer counters, rather than being inherent in the automaton itself. Despite its simplicity, it was these observations that led to the approximate string matching algorithm in Paper 2.

4.2 Paper 2: Obtaining Precision-Recall Trade-Offs in Fuzzy Searches of Large Email Corpora

In Paper 2, we developed a novel NFA for approximate string matching that allows for the highest control over edit operation constraints that we know of, and then we developed the bit-parallel string matching algorithm *cedas* (constrained edit distance approximate search) that simulates the NFA. Since approximate string matching at an edit distance of 2 may produce too many false positives, and an edit distance of 1 may consider too few approximate matches, we designed this algorithm under the expectation that some set of edit operation constraint parameters could obtain a useful precision-recall trade-off that lies between an edit distance of 1 and 2. To test this algorithm and our hypothesis, we searched for approximate matches of keywords relevant to the Enron scandal,[4] where the searches were performed on a list of indexed terms from the Enron dataset [47], a large real-world email dataset.

The work delves a bit more into automata theory, as it was required to explain how *cedas* works. The algorithm is an extension of the classic NFA for approximate matching (see Figure 2.3), which matches strings up to some threshold of edit operations. The classic NFA for approximate matching has one row for every allowed error. So the first row allows matches with no errors, the second row allows matches with a single edit operation error, and etc. For the new NFA provided in this work, each row of the automaton matches a string in which a specific combination of edit operations have occurred. Let L be the language (set) of strings accepted by an automaton, and $L_{(i,e,s)}$ be the language of strings accepted by a specific row of the automaton, where (i, e, s) represents the number of insertions, deletions, and substitutions respectively. Our novel NFA with an edit distance threshold of two would have a specific row that accepts each of the languages $L_{(0,0,0)}$, $L_{(1,0,0)}$, $L_{(0,1,0)}$, $L_{(0,0,1)}$, $L_{(1,1,0)}$, $L_{(1,0,1)}$, $L_{(0,1,1)}$, $L_{(2,0,0)}$, $L_{(0,2,0)}$, $L_{(0,0,2)}$ (see Figure 7.2). When all terminal states of this automaton are not disabled, the approximate string matching results are the same as the standard approximate matching NFA with an edit distance of two. In other words, the languages matched by the classic approximate matching NFA and the our novel NFA produce the same cover set[5] of language L for an edit distance of 2, but made up of different subsets. These rows

[4]https://en.wikipedia.org/wiki/Enron_scandal

[5]A cover of set X is a union of non-empty subsets of X that contains X [207].

are able to accept these specific languages due to constraining (or carefully selecting) the NFA transitions (the edit operations) between the rows of the automaton.

The NFA is simulated using row-wise bit-parallelism (Algorithm 8), as described in this and the previous paper, and a general form of the algorithm would run in $O(k^3n)$ time, where it is required that the search term is smaller than a computer word. Here, k is the threshold of edit operations, and n is the length of the text being searched. We argue that since k should not surpass 2 in the first place, the runtime complexity is acceptable, and our implementation of the NFA runs in linear time due to a maximum edit distance of 2.

In our experiments, we we created a list of terms derived from an inverted index of the Enron dataset, and searched it for 28 different keywords related to the Enron case (Cuiaba, BobWest, launder, collusion, ArthurAndersen, etc). Each keyword was searched for under all possible edit operation constraints between the edit distances of 1 and 2. In total, there were 64 different constrained edit distance searches per keyword. Each of the searches would return a list of approximate matches of the keyword, and we manually identified what were the positive hits for the keyword from the lists. We recorded the precision and recall for each of these searches. The harmonic means of the precision and recall were taken for each edit operation constraint combination. Keyword length was limited to 6 letters or greater, as smaller keywords (such as money) produced massive lists of approximate matches (often greater than 10000).

The results from this experiment essentially showed that any search with two substitutions, two deletions, or a combination of a deletion and substitution will reduce the precision of the approximate search significantly (anywhere from 20% to 35%). To maximize the increase in recall, while not completely destroying the precision, one should not include the approximate searches that allow for the previously mentioned poorly performing edit operations. However, we note that we only increased recall by 0.55% and reduced precision by 6.97% compared to the results for only allowing insertion and deletion edit operations. In terms of runtime, our results were about 6 times slower than *agrep*. A summary of the search results can be seen in Figure 4.1.

The new algorithm allows for full control over combinations of specific edit operation constraints. As previous approximate string matching algorithms can also be described in terms of some combination of constraints, they should only be able to match a subset of the approximate hits *cedas* can match. The exception to this is any approximate matching algorithm that also uses transpositions as an edit operation. The results of applying this algorithm show how one would likely maximize their approximate search recall without completely minimizing their precision.

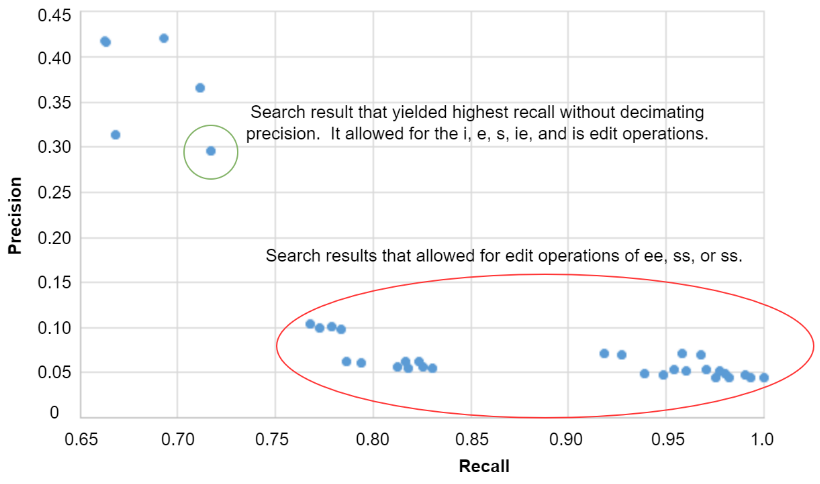


Figure 4.1: Summary of experimental results. Notation describes which edit operations are allowed in the approximate search: i for insertion, e for deletion, s for substitution, and pairs of letters denotes combinations of edit operations.

4.2.1 Contributions of Paper 2

The major contributions of Paper 2 can be categorized into our theoretical and empirical contributions to string matching and digital forensics research.

The theoretical contribution made in this paper was the development of an NFA that implements a constrained edit distance such that edit constraints may include any combination of edit operations. This allows us to have more approximation parameters than any previous non-weighted edit distance approximate string matching algorithm that we are aware of. We are also unaware of any approximate string matching NFA that has a row for each edit operation combination, rather than having each row account for a specific number of edit operations. The implication of this is that we are able to search the precision-recall trade-off more fully than previous algorithms, and thus capable of obtaining certain trade-offs that were once unobtainable, and perhaps allow for better precision or recall.

The practical contribution of this paper was showing that by performing approximate searches on every possible edit operation constraint parameter of the implementation of our NFA on real world data, that we identified which edit operation constraints will significantly reduce the precision. Our results showed that approximate matches accounting for two substitutions, two deletions, or a substitution and a deletion may decimate the precision for this specific dataset (early 2000s English text). Furthermore, we also identified a set of parameters in our experiment that allowed for the recall to be as high as possible without rending the precision to be impractical, though the improvement in recall was less than 1%.

4.3 Paper 3: Generic Metadata Time Carving

This work uses a simple string matching algorithm to carve for generic "dynamic signatures" for identifying filesystem metadata record timestamps from various filesystems. Our observations of metadata record timestamps were that they are typically closely co-located and that timestamps are often identical. Using the potential timestamps as signatures, we hypothesized that we could test the data surrounding the potential timestamps to verify if the potential timestamp does indeed belong to a filesystem metadata record or not. We refer to our novel method *Generic Metadata Time Carving* (GMTC).

The motivation behind this idea was due to some of the limitations of file carving. File carving only extracts the Content of a file, but not its File Name or Metadata information. By extracting a metadata record we have the possibility of full file recovery. It also cannot handle file fragmentation well, whereas file recovery via a file's metadata record has the possibility to contain all the locations a file's fragments. File carving depends upon maintaining a database of file signatures, and tools such as Scalpel [181] uses an exact string matching algorithm rather than a multi-pattern one. Thus, if a file signature is missing from the database, the file will not be recovered. Furthermore, needing to search for each header-footer file signature pair in the database individually will require the tool to read through the disk multiple times, severely lengthening the search time for unallocated files. The need of our GMTC method may be useful in scenarios in which critical File System Content as described by Carrier [34] is inaccessible.

The string matching algorithm used to perform this task actually deviates from our previous string matching methodology,[6] and only performs byte sequence equivalency tests. Essentially, we search through the entire disk image where for most every non-overlapping byte sequence of length m (where m is the supposed length of a timestamp and this sequence is referred to as a *candidate timestamp*), we check the non-overlapping m length byte sequences within the next k bytes for some number of matches. We refer to these sequences as *test sequences*. If

[6]When we began this research, we actually implemented the algorithm using the bit-parallel techniques discussed before. However, since filesystem metadata timestamps have no overlap, there is no need to search through the underlying data byte by byte. Thus, we opted to use a the simpler (and faster) sliding window approach described in the paper.

the number of matches exceeds some user defined threshold, we have found a *potential timestamp*. We do not perform timestamp equivalency tests on candidate timestamps that are composed of a single repeated byte (i.e. 0x00000000 or 0xFFFFFFFF). All locations of the potential timestamps, described as byte offsets from the beginning of the disk image, are recorded to a text file. This process is somewhat general, as we do not need to make assumptions about a specific filesystem other than the length m of the timestamp, and a decently good guess for k. For instance, the search does not need to consider the date range of the timestamps that it is searching for. Algorithm 2 explicitly shows the process, but a more detailed version of the algorithm is shown in Appendix A.2.

We show the method visually in Figure 4.2. The byte sequence in the green box is the current candidate timestamp, it's search window of length k is shown as the yellow box, and the three 4-byte sequences within the yellow box are test sequences. The equivalency test simply counts how many of the test sequences the candidate timestamp matches. In Figure 4.2, if the required number of matches was two, the above test would have failed and the search would move ahead by four. If the required number of matches was one, then the above test would have been passed, the offset to the candidate timestamp would be recorded, and search would move ahead by k bytes.

We take a brief moment here to discuss how we chose our parameters. The length of our timestamps m, the length of the search window k, and the number of required potential timestamp matches h. If the user knows the filesystem they are analyzing, they would use the parameters which best fit that filesystem. For NTFS (and for our experiments), this would be $m = 8$ because it has 8-byte timestamps. The length of k would be 24 so that the 24 bytes in front of the first timestamp in a SIA or FNA would contain the next three 8-byte timestamps. Finally, h was chosen through experimentation for both NTFS and Ext4. For NTFS, $h = 3$ as we often encountered high precision and recall. For potential timestamp matching, the higher value of h is, the more precise your potential timestamp matching results will be. This is because it is a characteristic of filesystem metadata records to contain identical and co-located byte sequences (the timestamps). For Ext4, we observed that setting h to 3 limited recall, so we opted for using $h = 2$. Similarly, we set $m = 4$ for searching for 4-byte inode timestamps, and $k = 12$ so that the search window possibly contains the next three timestamps following the first timestamp. Realistically, we do not know what the length of the timestamp we will encounter (a safe bet is either 4 or 8), but so long as we use a value for m that divides the length of a timestamp, k is a multiple of m and long enough to encounter co-located timestamps, then it should be possible to identify a variety of different sized potential timestamps. We discuss this subject as a limitation and

Algorithm 2: Basic Potential Timestamp Carving Algorithm.

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

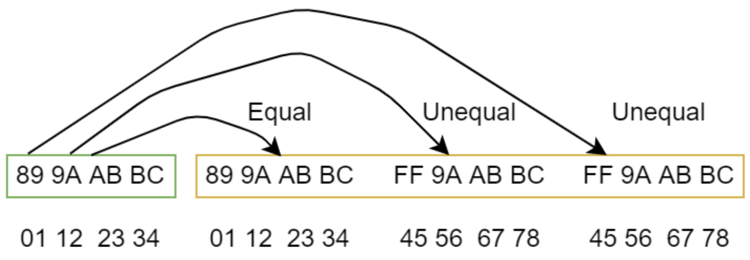


Figure 4.2: Example of a timestamp equivalency test of the potential timestamp carver.

future work in Chapter 5. The current implementations of our tools only support timestamps of length $m = 4$ and $m = 8$.

The Big-O time complexity given in the paper is $O(\left\lfloor \frac{|T|}{m} \right\rfloor \times \frac{k}{m})$, $|T|$ being the length of the disk image. This assumes the worst possible case, in which each non-overlapping m-length byte sequence has a timestamp equivalency test, and no potential timestamps are found. The time complexity does however neglect to account for the cost of interpreting byte sequences as little-endian, as well as the cost of checking for sequences of repeated bytes, which may increase runtime. Furthermore, we should specify that this was meant to be the time-complexity given for a general potential timestamp carving method, where the values of m and k could be any value. Since the implementation of the potential timestamp carver limits potential timestamp sizes to 4 and 8 (the maximum is 8 since the largest numerical type for storing timestamps is a C++ long long), and the value k is in most practical situations less than 1024 bytes, the effective time complexity of the implementation of the algorithm reduces to $O(T)$.

While this algorithm will likely find many filesystem metadata records, it will also contain many false positive potential timestamps. Thus, we created filesystem specific parsers that utilize the potential timestamp offsets given in the previously produced results file that verifies if these offsets are encompassed by a filesystem metadata record. This is similar to the work done by McCash [128] (see Section 3.2.3), who searched for NTFS timestamps with one program and verified the metadata with a different program. In this fashion, the filesystem specific parsers act as filters or classifiers which state whether a specific filesystem record occurs at the offset. Thus far, we have written filesystem specific parsers for NTFS and Ext4. The NTFS and Ext4 parsers both check the surrounding data of the potential timestamp, looking for Attribute flags in MFT records or filetype nibbles in the inode. We perform several other tests attempting to ensure that the perceived records are internally consistent, including that the relationships between the timestamps are plausible (for instance, a file should not be deleted before it is created), and that time timestamps fall within reasonable years (typically between the year 2000 and 2021). If the perceived filesystem metadata record passes all of the validity tests, the record's information is saved as a row in the output csv file.[7] The previous methods encompass the GMTC method, and a visualization of the method can be seen in Figure 8.2.

The Ext4 parser has additional complexity that is unique, as we also attempt to recover the File Name information associated with an inode, much like Dewald and Seufert [55]. The basic idea is that when we encounter directory inodes, and they have not been marked for deletion, it is possible for us to visit its associated directory entries. After verifying that the directory is indeed a directory (by checking the first two entries are for "." and ".."), we are able to obtain the inode number of the current directory as it is associated with the "." entry. We then can parse the rest of the directory entries and record the filename and inode number pairs to a Python dictionary. With the given offset to the current directory and its inode number, we can make guesses as to what the names and inode numbers of the subsequent inodes we encounter are within the same block group. Alternatively, once we have recorded the File Name content of an inode as well as its semi-unique file version number and creation time, we can look up an inode's File Name Content based upon this semi-unique information. This is useful in cases that inodes are discovered outside the inode table. Figure 4.3 illustrates the methods for connecting File Name content to Metadata content, and elaborates on some of the details.

In order to test the effectiveness of our Generic Metadata Time Carving (GMTC) method we created a 2GB image formatted with NTFS and a 2GB image formatted with Ext4 and flex-groups. The NTFS image contained 50 files, and the Ext4 image contained 25000 text files and 50 directories, where we added an additional property to the inodes so that they would also contain their inode number. To damage the images, we reformatted NTFS to exFAT, and reformatted Ext4 to NTFS. The task then was to use our tools to recover the filesystem metadata records from the original filesystems. For the NTFS experiments, we obtained all MFT records known to be on the image, and only encountered a single false positive MFT record. The Ext4 experiment encountered no false positive inodes, and we could not provide an estimate of the recall of inodes found throughout the disk, as there was the possibility of duplicate inodes (we encountered 57427 inodes). The number of non-duplicate inodes found was 5755. In terms of the effectiveness of linking the filename to the inode from the damaged image, we had a maximum precision of 28.92%. When connecting the filename information to the inode on the intact Ext4

[7]Logging information regarding the encountered data when performing the validity checks is also written to a text file.

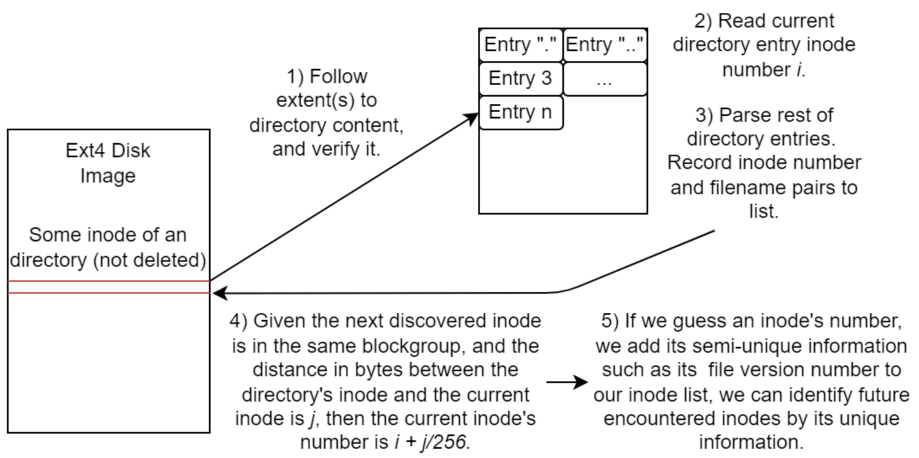

Ext4 Disk Image
Some inode of an directory (not deleted)
1) Follow extent(s) to directory content, and verify it.
Entry "."
Entry ".."
Entry 3
...
Entry n
2) Read current directory entry inode number *i*.
3) Parse rest of directory entries. Record inode number and filename pairs to list.
4) Given the next discovered inode is in the same blockgroup, and the distance in bytes between the directory's inode and the current inode is *j*, then the current inode's number is *i* + *j/256*.
5) If we guess an inode's number, we add its semi-unique information such as its file version number to our inode list, we can identify future encountered inodes by its unique information.

Figure 4.3: Novel method for connecting Ext4 files' filename and inode number pair to its associated inode entry.

partition, we obtained 100% precision and recall. Comparing our GMTC method with other popular data recover tools such as Encase and X-Ways, our's was the only one capable of recovering both MFT records and inodes from the reformatted images. In terms of runtimes, it took roughly 13 seconds to run the timestamp carver on a 2 GiB image, the NTFS parser took less than one second, and the Ext4 parser took about 8 seconds.

According to our results, we can use timestamps as a dynamic signature for carving filesystem metadata records, and we showed we can do so with no or very few assumptions about File System metadata. We emphasize that using the potential timestamp carver in isolation would unlikely benefit the forensic investigator due to the large number of false positive potential timestamps.

4.3.1 Contributions of Paper 3

The major contribution of Paper 3 was the identification of a "dynamic signature" for timestamps that exists in most filesystems. That being, a series of byte sequences that are identical and closely co-located. The implications of this are that we can potentially carve for file system metadata records of any filetype for many different filesystems, and that we have the potential of full file recovery. It also means that we can find metadata records not only in spaces such as the MFT or inode table, but we can find historical or duplicate metadata records from across the entire disk image, which may allow for deleted files to be recoverable.

Our experiments proved that the Generic Metadata Time Carving method worked on at least two different filesystems, NTFS and Ext4, even when the filesystem was corrupted. This is evidenced by the high precision and recall of our experiments. Dewald and Seufert [55], while their experimental methodology appears to be more complex, at times they encountered over 8000 false positive inodes within the blocks of the inode table itself using their content mode, whereas we only encountered a single false positive MFT record across the entire disk image. Even though comparing the recall of our experiments to Dewald and Seufert is less one-to-one, other than that our dynamic signature works on multiple filesystems, we had full recall for our NTFS experiment and the non-reformatted Ext4 image. Dewald and Serfert's [55] recall for the different signatures often contained thousands of false negatives.

Prior to our results, the similar methodology by McCash [128] searched directly for NTFS timestamps in plausible time ranges, applying a metadata filter afterwards. So, we have generalized McCash's approach by searching for more general potential timestamps, without even needing to know which datetimes we are searching for.

Another contribution of Paper 3 was that we created a novel method to connect Ext4 inodes to their filename data. Our results indicate that if the filesystem is not significantly overwritten, then connecting this metadata will work well. However, if significant portions of the filesystem metadata are overwritten, then its successfulness is limited. Previously, Dewald and Seufert [55] tried to connect this information by reconstructing the directory tree using their content mode, whereas we assign inode filenames and numbers primarily by using locally found information. Both methods utilize discovered directory entries, but Dewald and Seufert require more prior knowledge of filesystem parameters than ours.

4.4 Paper 4: Timestamp Prefix Carving for Filesystem Metadata Extraction

This work is a follow-up work to Generic Metadata Time Carving [151]. One of the major limitations of the previous work was that we could only carve for filesystem metadata records that contained precisely matching and repeating timestamps. We expand upon this idea, where we essentially perform approximate timestamp matching by way of allowing timestamp prefix matching. Rather than perform an exact timestamp equivalency test for the potential timestamp carver, we test if the first p bytes of the candidate timestamp is equivalent to the p length prefixes of the byte sequences it is tested against.

The method is shown visually in Figure 4.4. Like before, the byte sequence in the green box is the current candidate timestamp, it's search window of length k

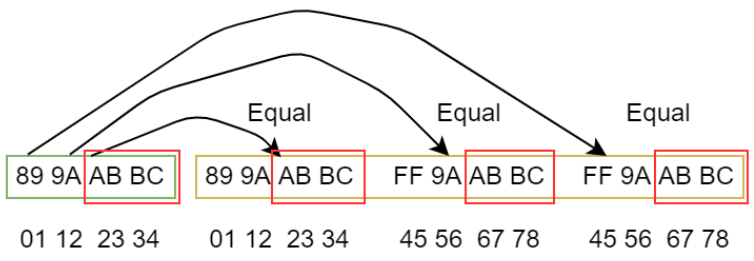


Figure 4.4: Example of a prefix-based timestamp equivalency test of the prefix-based potential timestamp carver.

is shown as the yellow box, and the three 4-byte sequences within the yellow box are test sequences. Rather than comparing the candidate timestamp and the test sequences directly, their prefixes are compared for equivalency. If the prefix length is two, then the candidate timestamp will match all of the given test sequences. The search for timestamps continues as the exact matching approach does. The prefix-based timestamp equivalency test is shown explicitly in Algorithm 3, where the full search algorithm is shown in Appendix B as Algorithm 13.

Algorithm 3: Prefix matching algorithm.

```
Input: Big-endian unsigned long long forms of candidate
       timestamp $x$ and test sequence $y$
Output: Number of matches for the candidate timestamp either increases
        or stays the same
$m$ # Length of timestamp;
$p$ # Size of prefix;
xorResult = $x \bigoplus y$;
if ((xorResult >> (8*(m-p))) == 0) then
    matchCount += 1;
end
```

The idea is that timestamps that are stringologically similar should be temporally similar as well. Our hypothesis was that by allowing for approximate timestamp matching we could increase the recall of filesystem metadata records carved for, and that as the prefix length p decreased the precision should decrease. Note, the most significant bytes of a timestamp usually contain information regarding the month and year of a timestamp.

In order to test our hypothesis we applied our prefix-based GMTC method on three realistic images, where we applied all possible values of prefix length p (this includes exact matching). The first image is a 476 GB NTFS image from the Digital Forensic Corpora [134], where it was chosen due to its large size and realistic forensic data. The second image was a 59.5 GB Ext4 image from our own personal collection, and it was again chosen because of its size and its assumed realistic data. The third image was a 1 GB NTFS image from NIST's Deleted File Recovery website [145], and it was chosen due to its purposefully chaotic timestamps. An important difference about these test images as opposed to the images in Paper 3 [151] is that the timestamps are varied and a substantial amount of file content on then, thus being far more realistic.

While we run both the prefix-based potential timestamp carver and updated filesystem specific parsers on the entire disk images, we limit our measure of precision and recall to constrained regions of the image. This is because we have no ground truth information regarding how many filesystem metadata records are spread throughout the disk or even in each partition. We utilized location-based data recovery[8] for measuring the precision and recall of locating MFT records from the $MFT and $LogFile, and we also measure the precision and recall for locating inodes from the inode table. As we may encounter many other filesystem metadata records not located in these spaces (for instance, in Volume Shadow Copies) we also account for which file contained hits for these records.

As our prefix-based potential timestamp carving method only adds a constant number of steps to the original algorithm, the time complexity of the algorithm is the same (effectively linear). Furthermore, we performed our prefix-based GMTC method on each of the disk images twice to measure the time taken to carve potential timestamps and the time taken by the parsers to verify metadata records, where we then averaged the results of the runs. We only conducted two timing trials due to the long runtimes we would often encounter.

The results of our experiments generally showed a significant increase in recall for locating filesystem metadata records, while maintaining 100% precision over all tests (no false positives). A summary of recall and runtime results are shown in Figure 4.5 and Figure 4.6 respectively. When assessing the precision and recall for MFT records carved from the 1 GB NTFS image's MFT table, the exact timestamp matching method obtained 8.8% recall, while the smallest prefix-based method obtained 97.9% recall. The total time taken to apply the exact timestamp GMTC method was 8.226 seconds, while the total time taken for the smallest prefix matching GMTC method was 15.019 (where there was very little difference in time for carving potential timestamps). When assessing the precision and re-

[8]Location-based data recovery compares the actual byte offset locations of files or metadata to the locations of the metadata we have identified. This way, we don't have to actually compare the metadata information to match records, only matching locations.

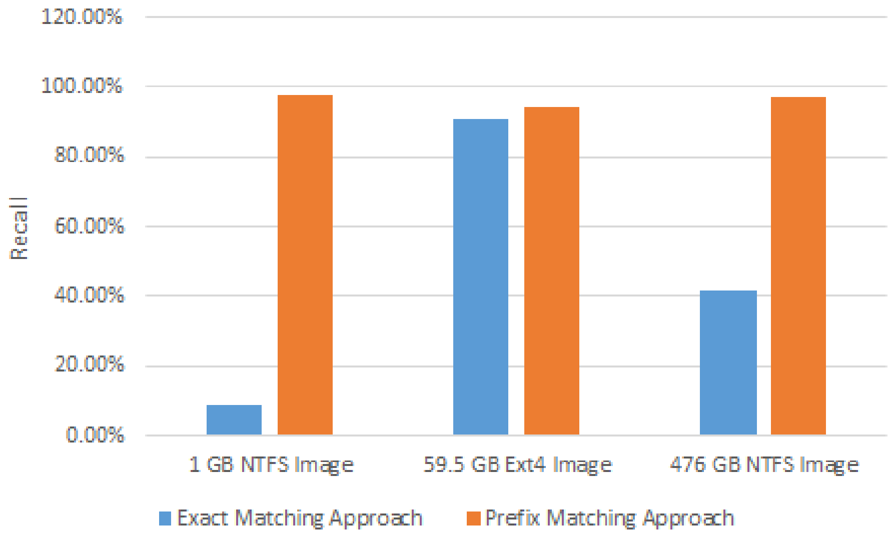


Figure 4.5: Recall results for the exact timestamp matching approach and the the shortest prefix-based timestamp matching approach for all tested images.

call for carving for inodes from the 59.5 GB Ext4 image's inode table, the exact timestamp matching GMTC obtained 91% recall, while the smallest prefix based method obtained 94.2% recall. The total time taken to apply the exact timestamp GMTC method was about 24 minutes and 7 seconds, while the total time taken for the smallest prefix matching GMTC method was about 28 minutes and 39 seconds. This experiment showed about a 100 second decrease in time from $p = 4$ to $p = 1$ for potential timestamp carving. For assessing the precision and recall for carving MFT records from the 476 GB NTFS image's MFT table, the exact timestamp matching method obtained 41.6% recall, while the smallest prefix based method obtained 97.3% recall. When assessing the precision and recall for carving MFT records from the same image's $LogFile, the exact timestamp matching method obtained 86.4% recall, while the smallest prefix based method obtained 87.1% recall. The total time taken to apply the exact timestamp GMTC method was about 2 hours and 31 minutes, while the total time taken for the smallest prefix matching GMTC method was about 2 hours and 27 minutes.

Other effects of reducing the prefix length parameter p was a substantial increase in potential timestamps (see Figure 4.7), where the increase in potential timestamps when decreasing p from its maximum to its minimum was anywhere from a four times as many potential timestamps to being magnitudes greater. Exact timestamp carving on the 1 GB NTFS image identified 892 potential timestamps, whereas $p =$

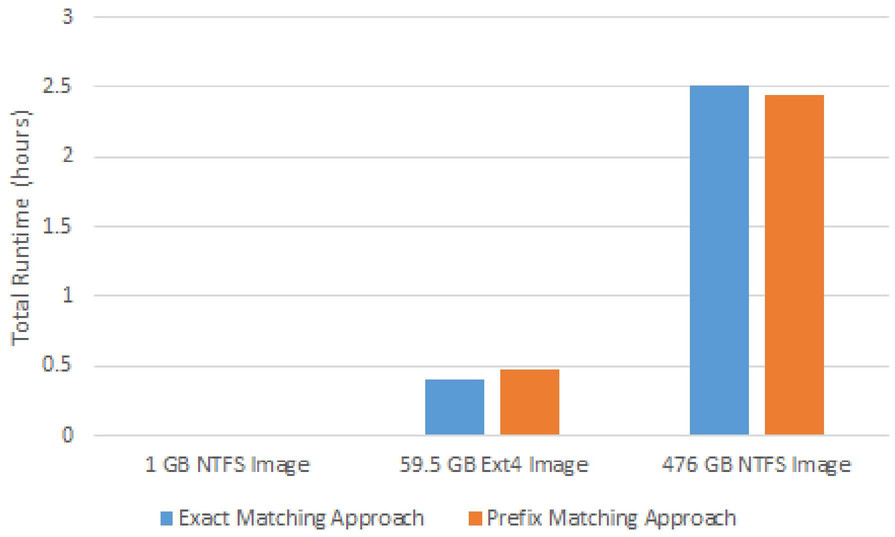


Figure 4.6: Average runtime results for the exact timestamp matching approach and the the shortest prefix-based timestamp matching approach for all tested images.

1 prefix-based carving identified 2061768 potential timestamps. Exact timestamp carving on the 59.5 GB Ext4 image identified 10630945 potential timestamps, whereas $p = 1$ prefix-based carving identified 97555603 potential timestamps. Exact timestamp carving on the 476 GB NTFS image identified 12235330 potential timestamps, whereas $p = 1$ prefix-based carving identified 57934625 potential timestamps. In most cases, the increase in potential timestamps caused the filesystem specific parser to take much longer to finish. The exception to this was the large NTFS image tests, but we believe that this is likely due to our implementation of reading through large files with the NTFS parser.

The total lack of false positive records was surprising, and for this reason we investigated the matter further. The fact that the total number of potential timestamps in our tests would grow by magnitudes at times (implying many false positive timestamps) and that the precision for carving filesystem metadata records remained at 100% implied that it is the filesystem specific parsers that ultimately controls the precision of the GMTC method. Likewise, the roll of the potential timestamp carver would appear to be controlling the number of byte offsets that the parser would have to check, affecting both the recall and the runtime of the parser.

To test our hypothesis, we ran a short experiment where we searched for potential

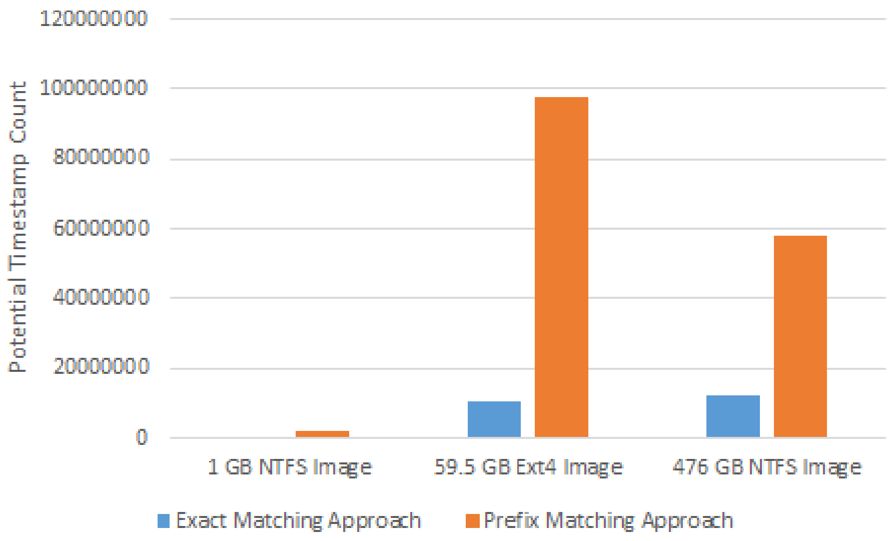


Figure 4.7: Count of identified potential timestamps by the exact timestamp matching approach and the the shortest prefix-based timestamp matching approach for all tested images.

timestamps and metadata records on random data. The idea was that while we expected to find many potential timestamps, we suspected we would encounter no false positive records. Furthermore, our expectation was that the GMTC method would be faster than only searching with the filesystem specific parsers. The experiment encrypted the 59.5 GB Samsung image with Kleopatra,[9] and we ran the potential timestamp carver with the previous NTFS and Ext4 settings and a prefix size of $p = 1$ on the encrypted image, generating the locations of the potential timestamps and recording the time taken for the search. We then ran each of the filesystem specific parsers over the image with their respective potential timestamp lists, recording the time taken and the number of record hits. A second experiment was conducted that applied the filesystem specific parsers without potential timestamp information, running the script over every byte in the image except the first and last 1024 bytes. The results of the experiments showed that we encountered no false positive record hits for any of the tests, but the interesting results were the time taken by the tools and the number of potential timestamps generated.

When searching for MFT records, we carved 360990 potential timestamp locations, and the runtime was about 17 minutes. Applying the NTFS parser afterwards took about 7 minutes to run, but applying the parser without potential

[9]https://www.openpgp.org/software/kleopatra/

timestamp information took approximately 5.75 hours. When searching for in-odes, we carved 182405692 potential timestamp locations, where the runtime was nearly 17.5 minutes. When applying the Ext4 parser afterwards, the parser took about 17.5 minutes, whereas applying the Ext4 parser without potential timestamp information took about 17.5 hours.

The results regarding running the parsers directly on the encrypted image prove that the filesystem specific parsers do indeed ultimately control the precision of the GMTC method, as we encountered no false positives. This is understandable as our filesystem specific parsers have a multitude of verification checks (including plausible datetime checks), and that filesystem metadata records are generally highly structured. The disadvantage of applying the parsers without potential timestamp information is that the run significantly slower than when they have potential timestamp information. This is due to the fact that the potential timestamp carver seems to act as a data reduction technique for possible byte offsets of the parser to check. For example, when running the parser by itself on the encrypted image, it must check nearly 59500000000 different bytes offsets. While using the potential timestamp locations for carving for 8 byte timestamps, it only has to check 360990 byte offsets.

The conclusion drawn from our results was that prefix timestamp carving does indeed work, its runtimes are practical, and it in some cases greatly increases the recall of identifying filesystem metadata records as compared to the exact matching timestamp carving. Furthermore, we have shown that prefix-based timestamp carving runs in more or less the same time as exact timestamp carving. A result that surprised us was that the precision remained at 100% for all prefix values p for both NTFS and Ext4. The results of our additional and brief experiments shows that the precision of the GMTC method is ultimately controlled by the filesystem specific parsers. However, running them without the help of the potential timestamp carver is not ideal, as filesystem specific parsers take significantly longer without first reducing the number byte offsets to check.

4.4.1 Contributions of Paper 4

The major contribution of Paper 4 was our creation of a novel approximate matching potential timestamp carver by only matching the p most significant bytes of candidate timestamps and test sequences, and this led to a significant increase in recall over the exact potential timestamp carving method while maintaining 100% precision. Stringologically similar timestamps are often temporally similar. Furthermore, the prefix-based potential timestamp matching algorithm has the same time-complexity as the original algorithm, since we only added a constant number of steps.

This was also the first instance of obtaining the results of the GMTC method on realistic and large datasets, proving that the time taken to run our tools is practical as the longest time for the prefix-based GMTC method only took 2 hours and 48 minutes on a 476 GB disk image.

An inadvertent contribution we made was a deeper understanding of the rolls of the potential timestamp carvers and the filesystem specific parsers, as we needed to answer why all of our results obtained 100% precision for identifying MFT records and inodes. Our further investigations showed that the filesystem specific parsers ultimately control the precision of the GMTC method, and appear to be very strict (which may not always be the case for all filesystems). However, the timestamp carving algorithms act as a kind of data reduction mechanism, significantly reducing the number of byte offsets the filesystem specific parser would need to check, and thus significantly reducing the total time to parse for filesystem metadata records.

Chapter 5

Conclusion

The purpose of this research was to study how to improve the precision or recall of string matching methods that are applied to digital forensics. Specifically, the methods concerned being approximate string matching and filesystem metadata record carving. Most previous research has focused on making improvements for string matching by increasing the speed of the search algorithms in order to assist the digital forensics backlog [17, 196, 212], but our approach was to focus on either making search algorithms more precise, or to improve their recall. Both precision and recall are valuable metrics for digital forensics, as precision allows investigators to encounter less false positive results, and recall allows an investigator to obtain a greater amount of relevant results [120]. Although, there is typically a trade-off between precision and recall, thus it is important to balance them.

While our focus was not to improve the speed or time-complexity of the search algorithms, we nonetheless employed techniques to ensure hasty retrieval.

Here, we answer our research questions given our theoretical and empirical results.

5.1 Research Question 1

How can we improve approximate string matching algorithms with respect to precision, recall, or accuracy specifically in the context of digital forensics.

In particular, how can the constrained edit distance be implemented into Non-deterministic Finite Automata where it allows for general edit operation constraints?

Chitrakar and Petrovic [40, 41, 43] had already shown that if one is aware how some byte sequence or text may be modified, then by applying constraints on possible edit operations to avoid uncommon string modifications an approximate string matching algorithm can produce more accurate results than by using the standard edit distance. Prior to our work, the constrained edit distance used as a metric for approximate string matching was only explicitly treated by Petrovic and Chitrakar. A limitation of their work was that edit operation constraints were often implemented with integer counters, preventing their algorithms from truly utilizing bit-parallelism, and also limiting the possible number of applicable edit operation constraints. Others such as Holub [98] had actually applied edit operation constraints by modifying the transitions in their bit-parallel simulations of NFAs, but did so without referencing the constrained edit distance by Oommen [152]. Thus, the focus of our approximate string matching research was to develop a nondeterministic finite automaton (NFA) that could support general edit operation constraints, and still support bit-parallelism. Allowing for more possible constraints permits for a deeper exploration of the precision-recall trade-offs for approximate string matching, where the hope is that we can find edit-operation constraints with superior precision-recall trade-offs.

In Paper 2 [174], we arrived at the idea that unlike the typical bit-parallel NFA simulations for approximate string matching [13, 209] that used an NFA where each row was associated with a single error, we can manipulate the transitions of an NFA such that each row is associated with specific a combination of edit operations. Furthermore, we theoretically proved that our novel NFA matched the same language as the classic edit distance based NFA, only that the rows partitioned the matches into a greater number of subsets. By disabling the terminal state within a row of our NFA, we can control the subsets of hits we wish to match or not match. However, the NFA is not entirely general either, as we cannot take into account edit operation possibilities defined by edit operations occurring in a specific sequence, or having some contiguous number of repeated operations.

Using our novel approximate string matching algorithm we showed that we can allow for combinations of edit operation constraints that are unreachable by other constrained edit distance string matching algorithms (possibly excluding transpositions). These new constraint parameters allow for further exploration of precision-recall trade-offs in approximate string matching. Typically for constrained edit distance algorithms, one must know how a text will be potentially modified in order to apply edit operation constraints that will improve accuracy. Since we did not and could not have that information from the Enron email dataset [47], we applied all possible constraint parameters (limiting the edit distance between one and two) to approximately search keywords related to the Enron case. The results from our tests showed that any search that allowed for two substitutions, two deletions, or combination of a deletion and substitution reduces the precision significantly. Using constraint parameters unique to our algorithm, we are able to push the recall as high as possible, without decimating the precision. The caveat however is that the parameter that allowed for these constraints was only slightly better in terms of recall, where we had a gain of approximately 0.55% recall, and had a drop of precision by 6.97%.

The implication of our results is that it can be used to obtain better precision *or* recall over older approximate string matching algorithms since they only allow for a subset of possible edit constraints we consider. However, this will only be true if the user has some knowledge of which edit operation constraint parameters should be used. Applying this algorithm to digital forensics, at the very least we can use the edit operation parameters to maximize the recall of approximate string matching, without minimizing the precision for some English datasets in cases where the user wishes to reasonably cast their net as wide as possible.

5.2 Research Question 2

2. As string matching is a component of carving, how can we apply exact or approximate string matching to carving such that we can improve the precision or recall for file or metadata recovery?

Given past research, we divide the act of carving into three distinct categories: file carving, metadata carving, and feature carving. All of the different methods of carving utilize some kind of exact, multi-pattern, or regular expression string matching algorithm. The search algorithms and their improvements in the context of digital forensics typically focused on the reducing the runtime of carving tasks directly [15, 17, 127, 196, 212, 213] rather than improving precision or recall. Improving precision and recall for file carving has typically been approached from trying to handle file fragmentation [107, 155], rather than applying or modifying string matching algorithms.

As Lyle [125] described, string matching errors are unlikely to occur due to random events, rather it is the limitations of the search tools that will typically cause false negatives. A reduction in false negatives in feature carving tasks have been seen in tools from Stewart and Uckelman [197] by using multi-pattern string matching to consider matches for various Unicode encodings. We can see that recall of carving activities may be increased provided that we search for more possible signatures. However, carving methods that use dynamic signatures do not search for a specific signature, but for either ranges of byte sequences or the relationships between byte sequences that is characteristic of a datastructure. As such signatures are unlikley to be unique to a given datastructure, so searching for dynamic signatures is likely to produce false positives and non-perfect precision, as evidenced by Dewald and

Seufert [55].

Our research focuses on using exact and approximate string matching methods to search for dynamic signatures that will help us carve filesystem metadata records; but we are not the first to do so. Others such as Dewald and Seufert [55], Plum and Seufert [163], Mueller [135], McCash [128], and Pomeranz [168] have all already used dynamic signatures to search for either files or metadata. However, ours was the first to attempt to improve the recall of filesystem metadata record carving by searching for a generic dynamic signature appears across various filesystems [151, 172]. Our Generic Metadata Time Carving (GMTC) work, as McCash [128] had previously shown, that the offsets to the dynamic signatures (in our cases potential timestamps) may be used as starting locations to verify if the signature is indeed encompassed by the datastructure being searched for.

As the following research subquestions are highly intertwined, we provide a single answer in response to both questions.

2.a. How can filesystem metadata carving be improved?

2.b. Which filesystems can we make an improvement for?

Our work, and the work of Dewald and Seufert [55] and Plum and Seufert [163], all have the advantage that we can identify metadata records without necessarily needing static metadata record signatures, we can potentially carve files with uncertain header or footer signatures, and we can all potentially have the ability to recover fragmented files without the need of critical File System Content (see Section 2.3.1). The primary improvement that our GMTC methodology makes over previous metadata carving research is that our approach is intended to be fairly generic, so that that it can be applied to various filesystems. At the offset of our research, our intention was not to have greater precision or recall than other tools that were designed for a specific filesystem, but that we could obtain relatively high recall for any filesystem, even those that had not been studied yet. Previously, only Ext4 [55] and APFS [163] and their possible dynamic signatures had been studied scientifically.

The key to making our approach generic was to search for potential filesystem metadata record timestamps. To reiterate, to search for potential timestamps we search for closely co-located and repetitive byte sequences of lengths less than or equal to 8 bytes. As noted in Paper 3 [151], it would appear that the majority of popular filesystems store the metadata record in this fashion (see Table 8.1). This implies that by searching for this dynamic signature we can obtain starting locations, a kind of data reduction of an entire disk image, to verify for filesystem metadata records. Up to this point, we have only proved that our method works for both NTFS and Ext4 [151, 172]. Whereas McCash [128] used the NTFS specific timestamp carving tool by Mueller [135] followed by a NTFS metadata parsing tool, our approach uses a generic timestamp carving tool followed by a specific filesystem metadata record parsing tool. This ideally makes the GMTC method extendable to various filesystems. Other than our precision and recall results of our tools, our greatest contribution for filesystem metadata record carving the genericity of our method.

Comparing the improvements in terms of precision and recall for filesystem metadata record carving of our Generic Metadata Time Carving (GMTC) work and previous work is difficult since there have been few studies into the subject. Furthermore, there is very little theory on which metrics to use to measure the quality of filesystem metadata carving, what is considered to be true positive hit, or even what the requirements of the test data are. A common measure of success used is comparing the novel tool with commercial tools. There is also the issue that different studies often examine different filesystems.

That being said, the work by Dewald and Seufert [55] often encountered over 7000 false positive inodes when searching for different signatures within the space of the inode table. Outside of the inode table, they often encountered millions or even billions of potential inodes. Given that they only accounted for 201269 searchable inodes in the inode table, it would imply that many more false positives were encountered across entire 16 GB Ubuntu image. Their recall for recovering inodes from the inode table appeared to range from approximately 36% to 100% depending upon the signatures used (with a varying number of false positive inodes encountered).

Plum and Dewald [163] used a similar approach but was searching for APFS filsystem metadata records with the purpose of extracting full pictorial files. Using disk images that utilized 1000 random file actions (add file, delete file, add folder, delete folder, etc.) their tool had an average 42.6% recovery rate, as compared to photorec which had an average of 8.1%. Comparing metadata based file recovery results to our record recovery results is difficult, since finding a metadata data record does not imply that the file will be recoverable.

As the previous scientific works for filesystem metadata record carving did not use the same metrics, methodologies, or data to measure the success of our their tools, it is difficult to compare our results to theirs. We however stand by our use of the precision and recall, as Laurenson [113] had used for measuring the success of file carving. Nonetheless, we can still interpret some of the previous results in terms of precision and recall. The results of Dewald and Seufert [55] show that their method had the possibility of having high recall depending on the signature, but with seemingly variable reliability. This indicated that there was room for improvement for recall for at least Ext4 inode carving. While our original recall results (Paper 3 [149]) for carving MFT records from a reformatted NTFS image and inodes from a non-reformatted Ext4 was 100%, the limitation was that all of the records on disk likely contained timestamps that exactly matched. As our later results would show (Paper 4 [172]), the recall of utilizing the exact timestamp matching GMTC method only achieved record recovery recall of 8.8% and 41.6% for two different disk images that contained chaotic and realistic timestamps. In these scenarios, Dewald and Seufert's tools likely performed better than ours for Ext4 inode recovery. As for the matter of precision, our GMTC methods appear to be much more precise than Dewald and Seufert's. While they yielded several thousand false positive inodes in an the inode table and potentially millions elsewhere on the disk [55], we achieved next to no false positive records for all of our experiments (the exception being a false positive MFT record in Paper 3 [149]). As Paper 4 [172] would later explain, this is due to the strictness of our filesystem specific parsers, an act that Dewald and Seufert did not perform. The exact matching GMTC method runtime was also faster than Dewald and Seufert's, as the total time to examine a 2 GiB image was between about 13 and 21 seconds [151], and their tools for carving inodes took two and half minutes on a 5.4 GB image [55].

To further improve the recall of our GMTC method, we created a prefix-based potential timestamp carving algorithm to be utilized in the GMTC method. Since timestamps store information such as the month or year in the most significant bytes of a timestamp, matching the prefixes of timestamps is a kind of approximate matching to identify temporally similar timestamps. This means that the dynamic signature could handle filesystems chaotic actions. Our hypothesis was correct, as carving MFT records from our NTFS images in Paper 4 [172] went from 8.8% recall to 97.9% and 41.6% recall to 92.3%. An additional benefit to the prefix-based timestamp carving method was that the time complexity of the timestamp carving algorithm remained the same since we only added a constant number of computational steps. The downside of this method was that searching for timestamps using smaller prefixes could result in some magnitudes more of potential timestamp hits, which can greatly increase the filesystem specific parser times by magnitudes as well.

This work also noted that the filesystem specific parsers that ultimately control the precision of the GMTC method. Thus, we compared the time it takes the parsers to recovery records from an image with and without the preliminary prefix-based potential timestamp information. In Paper 4 [172] we encrypted a 59.5 GB Ext4 image, and when using the 8-byte timestamp locations the NTFS parser took 7 minutes to complete. Without the potential timestamp locations, the parser took

5.75 hours. Likewise, when using the 4-byte timestamp locations the Ext4 parser took 17 minutes to complete. Without the potential timestamp locations, the parser took 17.5 hours.

The last improvement we would like to mention was how our exact matching potential timestamp GMTC method compared with established digital forensics tools for recovering metadata records from reformatted images. When comparing approach approach with Encase, X-ways, EaseUS, and bulk_extractor, only the EaseUS and the bulk_extractor were able to recover all MFT records from partially overwritten MFT image. Furthermore, X-ways was unable to carve for file content and Encase failed to carve for any tiff files since it was missing the correct file signature whereas we could recover all files since we were performing metadata-based data recovery. None of the tools except for ours were able to identify inodes.

2.c. Assuming a filesystem is significantly damaged and the filesystem does not store a file's filename metadata inside its associated filesystem metadata record, how can we connect the filename metadata to its filesystem metadata record?

Full file recovery is not complete without also obtaining the name and index number of a file [38], and this is a problem we treated in Paper 3 [151]. Since we only examined one filesystem that separated their Filename Content and their Metadata Content, we cannot answer this question generally, but only for ExtX filesystems. As inodes do not store their inode number or filename in their inode, but inside a directory entry, we needed to have the Ext4 parser use data within the potential inodes we encountered to try to learn what the filename and inode number pairs were. When checking offsets to potential timestamps, we gave special treatment to inodes that appeared to be directory entries, and were not deleted. This is because we could use the extents of the inode to read its directory entry blocks, and from this we obtain the inode number for the directory, as well as pairs of inode numbers and filenames that are stored in the directory. This allowed us multiple ways of guessing an inode's inode number

The first way to guess an inode number (assuming a default block group size) is to use the inode number of the dictionary we examined, its offset from the beginning of the disk image, and the offset of a different inode with an unknown inode number. If it appears that the inode and the dictionary's inode are in the same block group, then the offset from the dictionary inode to the other inode divided by the default size of an inode (256), plus the dictionary's inode number, is the new inode number estimate.[1] This provides a way to locally estimate what inode numbers are.

[1]Inode number estimate must be a natural number.

The second way to guess inode numbers uses the information that we record from exploring the dictionary entries. Once we encounter an inode and assign it a number using the previous method, add semi-unique information about the inodes to our recorded dictionary entries, this being its file version number and creation time. Then when coming across an inode anywhere on a disk image with the same file version number and creation time as an inode we have recorded, we can assign that inode a filename and number. Our approaches differ from Dewald and Seufert's approach that attempts to reconstruct the directory tree in order to connect an inode to its filename and inode number [55].

Results from Paper 3 [151] showed that when using our method for connecting inodes to their filename information, the precision for assigning the correct name to its inode was 100% when the Ext4 partition was not reformatted with a different filesystem, and at maximum 28.9% when the partition was reformatted.

From our research and preliminary tests, similar methods of connecting Filename Content and Metadata Content might also only be applicable to Ext2 and Ext3.

5.3 Limitations and Future Work

When attempting to answer our research questions, we were capable of coming up with new research questions, or noted that we did not sufficiently answer aspects of our research questions. For each listed limitation we provide a response.

The limitations of our approximate string matching research are primarily about technical limitations of our approximate string matching algorithm, *cedas*, designed and implemented in Paper 2 [171].

- The constrained edit distance algorithm, *cedas*, implemented in Paper 2 [174] only supports ASCII.

  In many digital forensics contexts, this would be unacceptable. However, the purpose of the paper was to test the algorithm we created on an ASCII dataset, so at the present time it was unnecessary to support other kinds of encodings. Possible future work related to this topic could be the implementation of either fixed-length or variable-length encoding support for *cedas*.

- Paper 2 [174] averages algorithm runtimes, but neglects to mention how many trials were conducted.

- The *cedas* algorithm does not support entirely generic edit operation constraints, such as accounting for order of edit operations or limiting the number of edit operation matches that consecutively occur [161].

A major difference between directly calculating the edit distance and using the edit distance as a metric for approximate string matching is that the theory for constrained edit distances using dynamic programming has already been developed [152], and no strong theory exists for implementing the constrained edit distance a string matching nondeterministic finite automaton (NFA). In our research, we established that some kinds of edit constraints are possible to implement into an NFA by way of using rows to represent kinds of edit operation combinations and by limiting or allowing certain edit operation transitions to exist. Future research could look at how these ideas can be extended to more complex edit constraints, and also identify how digital forenics may benefit from such an algorithm.

- The *cedas algorithm* does not currently support a combination of edit operation constraints that has a total edit distance greater than 2.

  The difficulty in creating such an algorithm is that we had to handcraft all of the transitions operations between all 10 rows of the automaton. This requires knowing which row of the automaton is connecting to the row being inspected, whether we are using its updated value or its previous value, and explicitly controlling which edit operation simulation is being performed. Also, as the edit distance grows, the NFA for *cedas* will "fan-out", creating many more rows and necessary transitions to treat. Future research could try generalize the *cedas* algorithm to support greater edit distances, where the automaton is automatically constructed and not "hard-coded". However, the application of such an algorithm to digital forensics may be limited since applying high edit distance searches will produce a significant amount of false positives.

- *cedas* is sub-optimal in terms of time complexity.

  The time complexity of simulating the NFA, $O(k^3n)$, where we emphasize the effect of the number of allowed errors k. Furthermore, the algorithm is based upon Shift-AND [209], rather than the faster Shift-OR [13]. By modifying *cedas* as a Shift-OR implementation, we could save a single computational step for each row of the algorithm. This enhancement would unlikely change how the algorithm operates in practical digital forensics. Other optimization methods, such as diagonalization [12], need to be assessed if they are applicable to the *cedas* algorithm.

The limitations of our filesystem metadata carving research, Paper 3 [151] and Paper 4 [172], are both technical and methodological in nature.

- The Generic Metadata Time Carving methodology has only been tested on two different filesystems, NTFS and Ext4.

  The development of the file parsing tools for takes an extraordinary amount of time, and we would be happy for other researchers to begin developing them. Although there are extremely useful resources for becoming familiar with a filesystem [62, 34], one must also be creative in thinking about how a parser can try to validate if a potential timestamp is actually part of a filesystem metadata record. For instance, if a metadata record has no signature, what aspects of the record could be useful to use when you know the potential location of its timestamp? Furthermore, if File Name information is separate from Metadata information, how can you try to connect those items? Future work needs to be done with respect to creating filesystem specific parsers for other filesystems, including APFS, FAT, and ReFS. Transitioning the tools to Ext2 and 3 will take far less effort.

- The filesystem parsers don't account for MFT records containing alternate datastreams or processing of other attributes.

  The limitation here is two-fold. For one, it would be beneficial to obtain as much information about an MFT record as possible. The second point, is that by overlooking alternate data streams, we are overlooking a source of potential data recovery. While this is not "research" per se, the parsers should be updated to understand as much information as possible about the metadata records being analyzed.

- The timestamp carver is not completely generic if one must guess the size of the timestamp.

  This is an item that we plan to research. The purpose of the original paper [151] was more or less to test the plausibility of the GMTC method, and the follow-up paper testing timestamp prefix carving [172] used the same parameters to remain consistent. The question we ask ourselves, is are there any parameters for timestamp carving that allow for 8, 4, or 2 timestamps to be carved for simultaneously?

We have also identified some future work that may be done that is unrelated to our limitations.

- The method for connecting inodes to their filename content needs to be improved. In our experiments from Paper 3, we were only able to connect about 28.9% of the inodes to their filename and inode number when the filesystem was damaged. So either by coming up with new methods to calculate inode number of carved inodes, or by improving the reading of directory entries, we have the potential to increase our ability to connect this information. Furthermore, what are the effects of an inode table that is fragmented?

- The potential timestamp carving algorithm and the filesystem specific parsers can be made faster. Because the potential timestamp carver performs linear search and is not dependent on data immediately before or after the area under examination, it would be possible to implement a parallel processing version of the algorithm. A work by Bayne [15] showed that when applying multi-pattern string matching and searching for 5 keywords simultaneously, he could search for the keywords nearly 7 times as when applying parallel processing on a CPU over standard processing on a CPU. Utilizing a GPU for the potential string matching algorithm would also speed up the search time significantly. In the same study by Bayne, he was able to increase the speed of searching for 5 keywords by about 5 times when switching from a CPU to a GPU. Parallel processing could potentially be used on filesystem specific parsers as well, as the NTFS parsing is done linearly through the disk. Although, the Ext4 parser would not benefit much from parallelism, as the order in which information is obtained matters. Another way to increase the speed of the parsers is to implement them into Cython,[2] which Python-like scripts may be compiled to C++.

- What are the effects of performing nonlinear search for potential timestamp carving, such as binary search? In the event that such searches are less effective than linear search, do they have a use case? This could have some potential in improving the time complexity of potential time carving from effectively $O(T)$ to something possibly sublinear.

- In most cases, writing the byte offsets to potential timestamps into a text file is tolerable, but in some instances this text file can become massive (greater than 10 GB). How can we store the potential timestamp on disk in a more space efficient manner?

- We do not know how the GMTC method works when coming across a virtual image within a partition. In Paper 4 about prefix timestamp carving [172], we came across several "boot.sdi, boot.sdi" files, of which were essentially small partitions, and we were able to read out the MFT records on these files. As of now, we are uncertain if we would obtain similar results on a virtual machine. Furthermore, how would the results for connecting inode

[2]https://cython.org/

and filename information be affected when coming across a virtual machine on an ExtX filesystem (assuming both the virtual machine and parent disk image are using ExtX)? According to Soltow [194], the ability to even find host data within a virtual machine storage file is dependent upon its data provisioning method, as as some methods zero out the allocated space for the VM file when the file is created, and others only have the VM zero out data when a file within the VM is being allocated.

- How would we go about developing a standard method for measuring the success of filesystem metadata carving?

  In Paper 3 [151], our method for measuring the success of our tools was similar to that of Plum and Dewald's, where we compared the *file id* of a discovered inode to our ground truth information about the dataset. In our method, we use precision and recall whereas Plum and Dewald use a recovery percentage of the actual file content that was able to be recovered. Then there is the question of how to measure the success of the tool on a non-synthetic image, such as we conducted in Paper 4 [172], where we measured the number of *Condition Positives* within spaces on the disk where metadata record information and their location is easy to recover. Rather than comparing inode or MFT numbers, we used a location-based data recovery method to verify if we found a record at a location where a record truly existed. This approach was similar to Dewald and Seufert's [55]. If filesystem metadata record carving is to be continued to be researched, a consistent methodology and datasets need to be established.

5.4 Final Thoughts

In the grand scheme of things, this research was concerned with exact and approximate string matching methodologies in order to improve the issues caused by the digital forensics backlog. The focus of the research was not to assist investigators by researching techniques to decrease the processing time of string matching based algorithms, but to improve the precision and recall of such tools so that investigators can review the results of their tools more efficiently. We have had some success in doing so.

The results of Papers 1 [171] and 2 [174] developed a theory and algorithm to have the capability of improving approximate string matching. The results showed the tool was able to obtain previously unreachable precision-recall trade-offs, capable of maximizing recall without minimizing precision. However, our empirical results showed this improvement in recall to be rather small in our case. Utilizing bit-parallelism, we ensured that the algorithm runs relatively quickly. A general form of the algorithm would run in $O(k^3n)$ where the edit distance k is unlikely to be higher than 2, and n represents the length of the text being read. This is not as fast as pre-existing approximate string matching methods which often run in near linear time [42, 100, 136, 139], but this appears to be the cost of accounting for a high degree of approximation parameters.

The results of Papers 3 [151] and 4 [172] developed a different kind of string matching process for identifying a dynamic signature to help recover filesystem metadata records over a variety of filesystems, the Generic Metadata Time Carving (GMTC) methodology. The primary advantage of our methodology compared to past filesystem metadata record carving approaches [55, 135, 163] is its genericity, in that the method can be applied to various filesystems. Thus far, we have shown that our GMTC method may also achieve higher precision and recall for identifying records from NTFS and Ext4 than past work, but more work needs to be performed in order to have standardized tests to compare different metadata carving methods. The GMTC method is also practical for digital forensics, as our longest running experiments indicated that we can process a 476 GB image in less than three hours. The string matching component effectively runs in linear time, and the runtimes for the filesystem specific parsers vary dependent upon the size of their input data and filesystem requirements. We do however emphasize that there is room for plenty of optimization for the filesystem specific parsers.

Future work that can assist in the study of the application of string matching methods to the digital forensics backlog are the developments of large and realistic datasets (especially disk images), standardized metrics to evaluate filesystem metadata record carving, standard methods to evaluate filesystem metadata record carving, and the development of (GMTC) filesystem specific parsers for filesystems other than Ext4 and NTFS.

Part II

Publications

Chapter 6

Paper 1: On Application of Constrained Edit Distance Algorithms to Cryptanalysis and Digital Forensics

Porter, Kyle, and Slobodan Petrovic. "On Application of Constrained Edit Distance Algorithms to Cryptanalysis and Digital Forensics." NISK Journal (2017): 112-123.

Abstract

This paper examines constrained edit distance algorithms and their applications to cryptanalysis and digital forensics. The constrained edit distance is an extension of the edit distance where the user may place constraints on the ways some string X is transformed into some string Y when calculating the minimum distance between strings, thereby potentially producing a different distance. If the user has a priori knowledge of which edit operations are probable or possible, then the constrained edit distance will report more accurate results than the unconstrained edit distance. We look at how constrained edit distance algorithms are implemented, how the distance can be integrated into approximate string matching algorithms, and how constraints are implemented into distance and search algorithms. Furthermore, we examine algorithms of this type which can be applied to cryptanalysis and digital forensics. Lastly, we make a small observation regarding how the Nondeterministic Finite Automaton for approximate string matching relates to Oommen's constrained edit distance formalisms, and show how we may apply basic constraints

to this automaton.

6.1 Introduction

Identifying the similarity between strings and approximate matching of similar strings or sequences is an important problem in computer science, and this applies to information security as well. Such problems are found in cryptanalysis where the distance between unequal binary sequences can be calculated [86][192], digital forensics for approximate keyword search [133], and intrusion detection for approximately matching attack signatures [41]. The computational complexity and accuracy of algorithms of these varieties is dependent on the distance used to quantify similarity. A commonly used distance is the Levenshtein distance [115], frequently referred to as the edit distance, where it is defined as the minimum number of elementary edit operations required to transform one sequence or string X into some string Y where the elementary edit operations are the insertion of a character into X, the deletion of a character from X, and the substitution of a character in X for a character in Y. While the edit distance is an effective measure of similarity in the stringological sense, it oftentimes produces far too many false positives in practical settings to the point where the application of these algorithms are simply ineffective. An extension of the edit distance, the constrained edit distance [152], combats the problem of excessive false positives by taking into account a priori knowledge about the probability or impossibility of the number and type of edit operations which are performed in the application being examined. This distance allows the user to specify arbitrary constraints regarding the number and type of edit operations that may be performed to transform some string X into Y, thereby producing a different distance than the unconstrained edit distance. In some cases where the unconstrained edit distance between strings is small, the constrained edit distance between strings can be increased or even become infinite if the transformation is impossible given some constraints.

In this paper, we study a selection of constrained edit distance algorithms which have been applied to cryptanalytic and digital forensic contexts. The algorithms chosen intend to showcase the full functionality and possibilities of the state of the art. We provide background information for the theoretical bases for the algorithms presented, an explanation of the algorithms themselves and methods to implement constraints, and examples of their applications. Lastly, we provide a novel observation on the relationship between the constrained edit distance theory established by Oommen in [152] and modern approximate string matching algorithms.

6.2 Background Theory and Algorithmic Methods

6.2.1 Levenshtein distance and Constrained Edit Distance

We begin by formally defining strings and aspects of determining similarity. We synonymously refer to a pattern, string, or sequence as defined by some arbitrary $X = x_1 x_2 \dots x_N$ where finite integer N is the length of X, and x_i for $1 \leq i \leq N$ are characters from some finite alphabet Σ. Furthermore, let X_i represent the prefix composed of the first i characters of X. Let $D(X, Y)$ correspond to the general Levenshtein distance, or edit distance, between strings X and Y where it is defined as the minimum number of elementary edit operations required to transform X into Y [115]. For many approximate matching algorithms, search string X matches some contiguous substring Y in the searched text Z if $D(X, Y)$ is less than or equal to some threshold k.

The following formalisms in this subsection were first defined by Oommen [152]. To express the edit operations formally, we need to introduce the null symbol ϕ which is used for representing the deletion and insertion operations. Let $\widetilde{\Sigma} = \Sigma \cup \phi$ and $a, b \in \widetilde{\Sigma}$ where the cost function $d(a, b)$ is mapped to a positive real number and defines the cost of the following elementary edit operations:

1. $d(x_i, \phi)$ is the cost of deleting character x_i from string X.
2. $d(\phi, y_j)$ is the cost of inserting character y_j into string X.
3. $d(x_i, y_j)$ is the cost of substituting x_i for y_j in string X.

For the computation of the edit distance, we shall assume that $d(x_i, y_j) = 1$ for all $x_i, y_j \in \widetilde{\Sigma}$ except in the trivial case when $x_i = y_j$ for which $d(x_i, y_j) = 0$. We shall call such a substitution, where $x_i = y_j$, a trivial edit operation. We shall also assume that the substitution of the null symbol by the null symbol is not permitted. The purpose of altering the cost is having the ability to weigh certain operations or characters more heavily based on a priori statistical knowledge of possible edit operations, and in this fashion two transformations between X and Y requiring the same number of edit operations do not necessarily produce the same distances.

Using the provided notation we have the tools to effectively express all possible transformations from string X to Y. We illustrate such a transformation through an example. Let $X =$ “secure” and $Y =$ “scared”, and let X' and Y' be transformation pair strings over $\widetilde{\Sigma}$. The pair X' and Y' describe the elementary edit operations to transform X into Y using ϕ to represent insertion and deletion operations. One possible transformation may be represented in the following way:

$$X' = \text{secure}\phi$$
$$Y' = \text{s}\phi\text{cared}$$

In this example, the character “e” is deleted from secure, the character “u” is substituted with an “a”, the character “d” is inserted into the word secure, and all other substitutions do not change the characters. The sum of edit costs can quickly be calculated via $\Sigma_{j=1}^{|X'|} d(x'_j, y'_j)$. This is just one of many possible transformations from X to Y.

The constrained edit distance is defined as the minimum sum of edit operations to transform string X into Y such that the transformations obey the given constraints, where the constraints may be arbitrarily complex as long as they are defined in terms of the number and type of edit operations [152]. For example, one may place constraints on the maximum number of allowed insertions i, deletions e, or substitutions s. Other types of constraints can limit the maximum number of times an edit operation may be performed consecutively, or limit the type or combination of edit operations performed. We examine some properties of transforming strings to recognize the functionality of applying constraints. According to Oommen [152][153], let $\Gamma(X, Y)$ be the set of all possible transformation pairs (X', Y') for transforming string X into Y via elementary edit operations. The cardinality of $\Gamma(X, Y)$ is given by the following expression:

$$|\Gamma(X, Y)| = \sum_{m=max(0,|Y|-|X|)}^{|Y|} \frac{(|X| + m)!}{m!(|Y| - m)!(|X| - |Y| + m)!} \tag{6.1}$$

By adding constraints, we eliminate some elements of $\Gamma(X, Y)$ since there are limitations on possible transformation pairs (X', Y'). Let the constrained set of transformations of $\Gamma(X, Y)$ be denoted as $\Gamma_T(X, Y) \subset \Gamma(X, Y)$. The set of elements $\alpha \subset \Gamma(X, Y)$ which have the minimum sum of edit costs are transformations that correspond to the edit distance between X and Y. The set of elements $\beta \subset \Gamma_T(X, Y)$ which have the minimum sum of edit costs for all possible transformations under constraints T correspond to the constrained edit distance between X and Y. Since $\Gamma_T(X, Y) \subset \Gamma(X, Y)$, β potentially contains fewer elements than the unconstrained case α if the sums of edit costs are equal (fewer possible transformations), or α and β may now correspond to different edit cost summations.

When setting constraints appropriately, we can eliminate transformations on X which we know are unlikely or impossible to occur within an application. This also changes which strings Y are considered to be similar under the unconstrained edit distance. For example, let the constraints be a maximum of one deletion and one insertion when calculating the edit distance between strings X and Y. $\Gamma_T(X, Y)$

is empty for any strings Y such that $|Y| < |X| - 1$ or $|Y| > |X| + 1$. This quality is useful for search, for if we have a priori knowledge of which types of errors may occur on some string X, then we can apply constraints that align to these probabilities, which then in turn can lead to some constrained edit distances between strings that require unlikely transformations to become increased or even infinite.

6.2.2 Constrained Edit Distance and Dynamic Programming

Common methods of implementing the constrained edit distance are either through dynamic programming or automata theoretic methods. Dynamic programming methods have been shown to be considerably more flexible than finite automata implementations, but the methods utilizing automata can conduct simpler fast approximate string matching. Calculating the constrained edit distance utilizing dynamic programming is based on the methods for calculating the unconstrained edit distance, first described by Wagner and Fisher in 1974 [205].

Oommen developed a dynamic programming method for calculating the constrained edit distance with constraints defined in the number and type of edit operations that may be performed to transform some string X into string Y [152]. The essence of this algorithm is dependent on the recursive function $W(i, e, s)$ for the constrained edit distance associated with editing prefixes X_{e+s} to Y_{i+s} subjected to the constraint that exactly i insertions, e deletions, and s substitutions are performed.

Theorem 1. *[152] For any two strings X and Y, let $W(i, e, s)$ be defined by:*

$$\begin{aligned} W(i,e,s) = \mathit{min}[&\{W(i-1,e,s) + d(\phi, y_{i+s})\} \\ &\{W(i,e-1,s) + d(x_{e+s}, \phi)\} \\ &\{W(i,e,s-1) + d(x_{e+s}, y_{i+s})\}] \end{aligned}$$

For all feasible triples (i, e, s), and $W(0, 0, 0) = 0$.

Using the inherent constraints of the relationship between possible insertions, deletions, and substitutions all feasible values of i, e, s are used to fill out the entire $W(\cdot, \cdot, \cdot)$ array. An example of an inherent constraint is if string X is being transformed into an equal length string Y and one insertion operation must take place; then one deletion operation must take place as well. Theorem 1 describes how to calculate the values of $W(i, e, s)$, and in Theorem 2 we see the relationship between the constrained edit distance $D_T(X, Y)$ and the $W(\cdot, \cdot, \cdot)$ array for some constraints T expressed in terms of the number of insertions i.

Theorem 2. *[152] The quantity $D_T(X, Y)$ is related to the elements of the array $W(i, e, s)$ as follows:*

$$D_T(X,Y) = \mathit{min}_{i \in T} W(i, |X| - |Y| + i, |Y| - i) \tag{6.2}$$

Essentially, from the possible i, e, and s constraints the minimum number of necessary non-trivial edit operations is returned. We give an example here using the words "secure" and "scared". After inputting these strings into an algorithm for computing $W(\cdot,\cdot,\cdot)$, some transformations and their corresponding edit costs are given in Table 6.1.

Table 6.1: Transformations between "secure" and scared" given various constraints

Constraints (i, e, s)	**Transformation**	**Sum of Edit Operation Costs**
(0,0,6)	secure scared	5
(1,1,5)	secureϕ sϕcared	3
(2,2,4)	secuϕreϕ sϕcϕared	4
(3,3,3)	secuϕre$\phi\phi$ sϕcϕarϕed	6
$\vdots$	$\vdots$	$\vdots$

If our constraint was that the transformation must contain more than one insertion, then the constrained edit distance between "secure" and "scared" would be 4. It is important to note that the number of substitutions is for general substitutions, but the cost is only applied if the letters being substituted differ.

Applying complex constraints requires modification of the recursive function $W(i, e, s)$, as we shall see later in the paper. However, if the constraints are relatively simple, such as requiring a precise number of specific edit operations, we simply do not consider values $W(i, e, s)$ which do not fit the constraints. Calculating $W(\cdot,\cdot,\cdot)$ and a corresponding constrained edit distance $D_T(X, Y)$ have $O(|X||Y|min(|X|, |Y|))$ time complexity. The flexibility and practicality of the algorithm are given by the facts that the user can modify the cost function operations, apply more complex constraints, and can produce the optimal edit transformation by backtracking through $W(\cdot,\cdot,\cdot)$ in $O(max(|X|, |Y|))$ time [152].

6.2.3 Automata Theory and Approximate Matching

In order to discuss search algorithms utilizing the constrained edit distance, we first need to discuss bit-parallel implementations of the Nondeterministic Finite Automaton (NFA) for approximate string matching. We first go over some necessary components of formal language theory.

Any finite automaton A may be defined by its set of states Q, the set representing the alphabet of characters Σ, the set of initial states $I \subseteq Q$, the set of terminal states $F \subseteq Q$, and arrows from state to respective state representing the transition relationships between the states. Each arrow is associated with some set of characters from Σ representing which characters may trigger a transition from one state to the next. Upon beginning to read in some string of characters, the automaton is active in its initial states. For each character input from the string, the active states may transfer to other states if the input character is included in the set of characters associated with the transition. If the sequence of contiguous characters input from a string into an automaton allows the traversal from an initial state to a terminal state, then the string is considered to be a match.

Figure 6.1 shows a common automaton for approximate string matching. This automaton is nondeterministic, meaning that any number of the states in Q may be active simultaneously. Furthermore, such nondeterministic automata allow for ϵ-transitions, where transitions are made without needing a prerequisite character. The initial state is represented by the node with a bolded arrow pointing to it, and is always active as indicated by the self-loop. The terminal states are represented by the double-circled nodes, the horizontal transitions represent exact character matches, vertical transitions represent insertions, solid diagonal transitions represent substitutions, and dashed diagonal transitions represent deletions via ϵ-transitions.

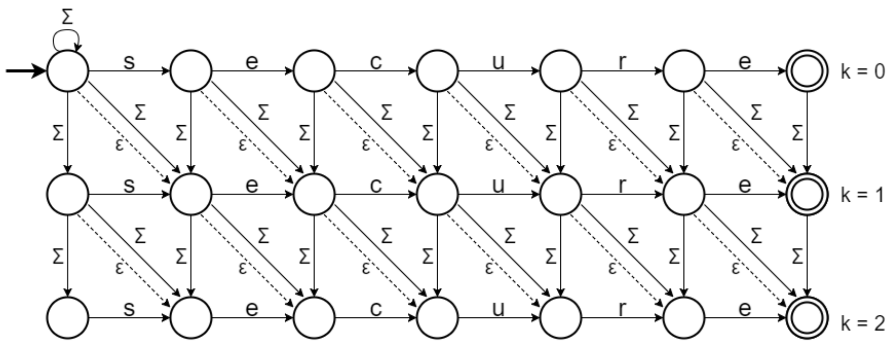


Figure 6.1: NFA for approximate string matching. Matches the pattern "secure" allowing two edit operations.

For an approximate search with an edit distance threshold k, this automaton has $k + 1$ rows. The first row in the automaton represents a machine that performs exact matching of the search pattern against the searched input text Y, the second row is a machine that matches the search pattern in Y with one edit operation performed on the pattern, etc. The primary advantage of these theoretical machines is that they can determine potential errors in the pattern simultaneously, where for every input character each row checks for potential matches, insertions, deletions, and substitutions against each position of the pattern. Constraints can be applied to this type of automaton by modifying the transitions or counting potential edit operations, as we shall see in Section 6.3.

Due to the fact that these automata are nondeterministic, they must be simulated in practical applications. An efficient form of simulation is via bit-parallelism, wherein bit-vectors represent each row of the automaton and are updated by way of basic bitwise operations which correspond to the automaton's transition relations. The bitwise operations update all the bits of a bit-vector at once, and therefore update the states of a row of an automaton simultaneously. This parallelism reduces the number of operations a search algorithm performs at most by w bits in the computer word (32 or 64 bits) containing the bit-vector if the length of the pattern is less than or equal to w [65].

Here we introduce some fundamentals of bit-parallelism based algorithms, see for example [142]. Each row of the automaton for pattern X is represented as a binary vector of length $|X|$, and we create a table of Boolean vectors $B[t_j]$ of the same size, called *bit-masks*, as representations for incoming characters t_j for comparison. These bit-masks represent the positions of a character within the search pattern. For example, Table 6.2 gives the bit-masks for the pattern "secure", where any character not present in the pattern is represented by the * symbol.

Table 6.2: Bit-mask table for the word "secure"

Character t_j	**Bit-mask** $B[t_j]$
c	000100
e	100010
r	010000
s	000001
u	001000
*	000000

We examine the bit-parallel construction of the NFA from Figure 6.1 to understand the unconstrained matching case, as the constrained cases are extensions of this algorithm. Each row of the automaton is represented by a binary vector R_i, $0 \leq i \leq k$ where k is the total number of allowed errors (non-trivial edit operations), and the goal is to update each row R_i as defined by the automaton. A clarification needs to be made regarding the direction of the automaton in Figure 6.1 versus their simulated counterparts. By convention, automaton graphs typically traverse from left to right; however the vectors simulating these automata typically traverse from right to left, see [142]. The NFA from Figure 6.1 is simulated by Algorithm 4, where the initial state of each row is $R_i = 0^{|X|-i}1^i$:

Algorithm 4: [142] Bit-parallel simulation of the NFA for approximate string matching.

```
0: $R'_0 \leftarrow ((R_0 << 1)\ \&\ B[t_j])|\ 0^{|X-1|}1$
0: $R'_i \leftarrow ((R_i << 1)\ \&\ B[t_j])\ |\ R_{i-1}\ |\ (R_{i-1} << 1)\ |\ (R'_{i-1} << 1)$
```

The first line of this algorithm represents a character match with no errors taken place. The subsequent rows take in account for i errors, where each of the components is bitwise OR'd with different transition relationships. Explicitly from left to right, the first expression represents exact character matches, the second represents an insertion, the third a substitution, and the fourth a deletion. A match is accounted for if any of the rows have the $|X|$th bit from the left side equal to 1, the bits representing the terminal states. This algorithm was first implemented by Wu and Manber and runs in $O(k\lceil |X|/w\rceil |Y|)$ time [209], but more modern algorithms have made improvements to reduce the time complexity to $O(\lceil(|X|-k)(k+1)/w\rceil |Y|)$ (see for example [100]).

6.3 Applications of Constrained Edit Distance Algorithms

This section is split into subsections for applications of constrained edit distance algorithms implemented through dynamic programming and bit-parallel NFA simulation.

6.3.1 Dynamic Programming Applications

Perhaps the most extensively used application of the constrained edit distance has been in cryptanalytic generalized correlation attacks against stream ciphers with irregularly clocked linear feedback shift registers (LFSRs), where the algorithms being used are based on the $W(i, e, s)$ function from Theorem 1 with modifications. For a thorough review of correlation attacks see Siegenthaler [192]. Here we describe the essentials necessary to show how the constrained edit distance is applied to generalized correlation attacks. Correlation attacks are attacks on non-linear combination stream ciphers where the goal is to use the correlation between the LFSR R_j and the ciphertext to reduce the number of total guesses of the key significantly. The Hamming distance is used to quantify the similarity between the LFSR output and the ciphertext for standard correlation attacks, but when the sequences are of different lengths then the Hamming distance cannot be used. For stream ciphers employing irregularly clocked LFSRs, this is the case, and the edit distance is an appropriate distance.

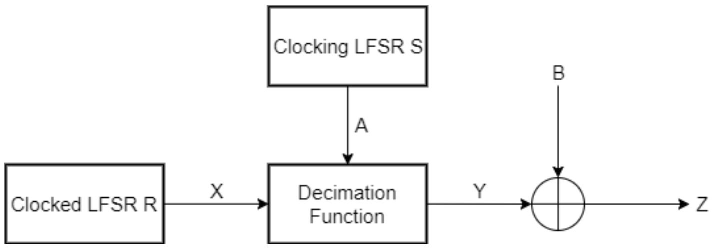


Figure 6.2: Clock-controlled stream cipher model from Golić and Mihaljević [86]

Work by Golić and Mihaljević [86] introduced the concept of a generalized correlation attack, and described the procedure to attack a simple clock controlled stream cipher using the constrained edit distance. Figure 6.2 shows this model, where a clocked LFSR R produces a binary sequence X, and a clocking LFSR S produces sequence A which is used as input with X into a decimation function that outputs the sequence Y. The bits in sequence Y are given by the bit characters $y_n = x_{f(n)}$ where $f(n) = n + \Sigma_{j=1}^{n} a_j$ for $n = 0, 1, 2, \ldots$, which is just X with deletions. This decimated sequence Y is then XOR'd with noise B, such as plaintext, producing the ciphertext Z. In terms of string editing, X may be transformed into Z through a number of deletions and substitutions, where the maximum number of consecutive deletions is 1. Golić and Mihaljević generalized the constrained edit distance solution for applying constraints of a maximum of E consecutive deletions.

Conceptually, the constrained edit distance algorithm applied in this attack is computed by means of dynamic programming. However, only deletions and substitutions are permitted in the function $W(\cdot,\cdot)$ and the recursion only calculates edit distances for which the maximum number of consecutive deletions is E. The prefixes of the strings X and Z being examined are X_{e+s} and Z_s where $|X| = M$ and $|Z| = N$.

Theorem 3. *[86] The partial constrained edit distance $W(e, s)$ satisfies the recursion where d_0 is the cost of deletion*

$$\begin{aligned} W(e,s) =& \mathit{min}\{W(e-e_1, s-1) + e_1 d_0 + d(x_{e+s-e_1}, z_s) : \\ & \{0, e - \mathit{min}\{|X| - |Z|, sE\}\} \leq e_1 \leq \mathit{min}\{e, E\}\} \end{aligned}$$

for $1 \leq s \leq N$, $0 \leq e \leq \mathit{min}\{M - N, (s+1)E\}$, *and, for* $s = 0$ *and* $0 \leq e \leq \mathit{min}\{M - N, E\}$, $W(e, 0) = ed_0$.

The constraint of E is performed in the recursion by determining the minimum distance between prefixes considering up to E deletions in Z. Even more advanced constraints can be implemented in this scenario by further modifying the recursion function as shown by Petrović and Golić [161]. In addition to having a maximum consecutive number of deletions E or insertions I, they simultaneously applied constraints in which between consecutive runs of substitutions there can be at most one consecutive run of deletions and/or at most one consecutive run of insertions. Similar constructions were used to extend these results to arbitrary combining functions without memory [84], and then to alternating step generators [85].

These methods and algorithms have been applied to the field of digital forensic search as well. In [159], the constrained edit distance with constraints on maximum length consecutive runs of deletions and insertions has been used for a digital forensic search preselection phase, in which large fragments of the total search space have their constrained edit distance calculated between the fragment and a search pattern. When the distance returned was beneath a prespecified threshold, the fragment was considered to be worth investigating. By using the constrained edit distance rather than the unconstrained edit distance, a significant data reduction of data needing closer inspection was obtained. A similar data reduction approach was applied to a preselection phase of selecting relevant Snort signature rules given an input string utilizing only a constraint on the maximum consecutive runs of deletions [160].

6.3.2 NFA Simulation Applications

Constrained edit distance applications utilizing NFA simulation are typically used in approximate string matching scenarios. This distance was introduced into approximate search by Chitrakar and Petrović [40] and applied the algorithm for spam filtering with a priori knowledge of the probabilities of errors which may be introduced into spam keywords. The constraint in this case sets a maximum number of possible *indels*, defined as the total sum of insertions and deletions. The method of implementation uses the NFA for approximate matching and is an extension of Algorithm 4, where it associates an integer counter to every state that represents how many insertion or deletion transitions can still be performed on the substring being analyzed. If a state is traversed to via an insertion or deletion, then that state's counter is reduced by one, and when a state is reached for which the counter is 0 then the algorithm no longer permits insertion or deletion transitions from that state. A similar algorithm which sets constraints on the maximum number of allowed insertions, deletions, and substitutions using counters was used for approximate search in intrusion detection in which the probability of edit transformations on attack signatures was previously known [41]. Both methods saw reductions in false positive matches compared to the unconstrained cases under the specified conditions.

A question regarding constrained edit distance NFA algorithms is how we may apply arbitrary constraints, such as a maximum consecutive run of a particular type of edit operation, so that we may apply more flexible constraints to approximate search. Such developments could lead to faster generalized correlation attack cryptanalysis. We discuss this in the next section.

6.4 Discussion

6.4.1 Oommen's Formalisms Applied to Approximate Matching NFA

We would like to examine other ways we may implement constraints to the NFA for approximate matching. A novel observation we make in this paper is describing this NFA in terms of Oommen's string editing formalisms. The automaton allows for the set of (X', Y') transformation pairs for all strings X and Y over $\widetilde{\Sigma}$ such that the sum of edit operations is less than or equal to k. The transformation pairs X' and Y' can be obtained by traversing the automaton. Horizontal transitions are edits considered by (x_i, y_j) pairs where $x_i = y_j$, vertical transitions are considered by (ϕ, y_j) pairs, ϵ-transitions are considered by (x_i, ϕ) pairs, and diagonal transitions are considered by (x_i, y_j) pairs for $x_i \neq y_j$. We give an example for strings $X = secure$ and $Y = secare$. Figure 6.3 shows the possible transformations from X to Y for $k \leq 2$, which can be traced on the automaton in the following ways.

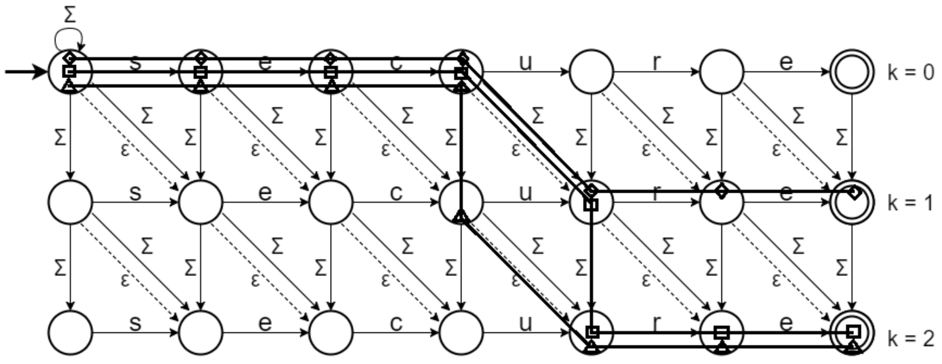


Figure 6.3: NFA for approximate matching with possible transformations between "secure' and "secare".

The line with diamond dots corresponds to the transformation pair $(secure, secare)$, the line with square dots corresponds to $(secu\phi re, sec\phi are)$, and the line with triangular dots corresponds to $(sec\phi ure, seca\phi re)$. It is intuitive to see that by disabling transitions in the automaton we are then by definition applying constraints to $\Gamma(X,Y)$ where $D(X,Y) \leq 2$, but the problem is how to translate more complex constraints as seen performed in the dynamic programming method of Oommen into transitions in an NFA. Challenges which come about are that Oommen's $W(\cdot,\cdot,\cdot)$ array is a large memory matrix of the possible ways to transform one prefix into another, whereas algorithms for simulating the NFA for approximate matching are typically memoryless. Furthermore, the nondeterministic nature of NFA makes it difficult to see at which stage in the transformation certain edit operations took place since how a string was transformed can be ambiguous. The implementation of complex constraints into the NFA for approximate matching are considered to be open problems.

6.4.2 The Automata Processor

While NFA simulation via bit-parallelism is quite fast, it is limited by the fact that the length of the search pattern must be less than or equal to the size of a computer word w for the Levenshtein approximate search algorithms to run optimally. Recent hardware developments such as the Automata Processor (AP) [56], which is a native hardware implementation for nondeterministic finite automata, allows for approximately 1.5 million NFA states and thousands of NFAs to be used in parallel. Most importantly, all of these states can process an input symbol and access successor states in a single clock cycle [206]. This hardware architecture has already been used for intrusion detection. An application called Fast-SNAP [185], used AP to scan network data streams for 4312 Snort signature rules simultaneously, for which the throughput was 10.3 Gbps. These results highlight the importance of implementing constraints into approximate matching NFA utilizing automata theoretic methods.

6.5 Summary

In this paper, we discussed the methods for implementing the constrained edit distance, how different constraints can be implemented, and looked at how constrained edit distance algorithms have been applied to digital forensics and cryptanalysis.

The constrained edit distance computation may be implemented as string distance algorithms where they are built on a dynamic programming algorithm by Oommen [152]. Such algorithms implement their constraints by simply choosing which exact edit operations took place in the transformation from string X to Y or more complicated constraints may be implemented by modifying the recursive algorithm to only consider transformations with some maximum consecutive runs of specific edit operations. These algorithms have been applied to cryptanalysis, being used in generalized correlation attacks against stream ciphers with irregularly clocked

LFSRs [84][85][86]. Additionally, the constrained edit distance may be used in data reduction for preselection phases of pattern searching in intrusion detection rule databases [160] and digital forensic search spaces [159]. Primary advantages of this dynamic programming method are the abilities to set flexible constraints, and to reconstruct the sequence of edit operations.

Alternatively, the constrained edit distance may be implemented inside algorithms which simulate the nondeterministic finite automaton for approximate string matching. The algorithms we examined simulate this automaton by utilizing bit-parallel approaches [65], and implement their constraints by applying integer counters to each active automaton state representing the maximum number of edit-operations the substring under analysis may still apply. Algorithms of this variety have been used in spam detection [40] and intrusion detection [41]. Additionally, new hardware such as the Automata Processor [56] allows for direct implementation of NFA, which can make searching even faster, and has already been applied to intrusion detection [185].

Lastly, we made a novel observation that the NFA for approximate matching using the Levenshtein distance can be described in terms of Oommen's string editing formalisms. By this definition, it is clear to see that a constrained edit distance may be implemented into this NFA by eliminating or modifying transitions. Translating complex constraints, such as maximum consecutive runs of specific edit operations, which have been performed with Oommen's algorithm into transition relations in an NFA remains to be an open problem.

Chapter 7

Paper 2: Obtaining Precision-Recall Trade-Offs in Fuzzy Searches of Large Email Corpora

Porter, Kyle, and Slobodan Petrovic. "Obtaining Precision-Recall Trade-Offs in Fuzzy Searches of Large Email Corpora." IFIP International Conference on Digital Forensics. Springer, Cham, 2018.

Abstract

Fuzzy search is often used in digital forensic investigations to find words that are stringologically similar to a chosen keyword. However, a common complaint is the high rate of false positives in big data environments. This chapter describes the design and implementation of `cedas`, a novel constrained edit distance approximate string matching algorithm that provides complete control over the types and numbers of elementary edit operations considered in approximate matches. The unique flexibility of `cedas` facilitates fine-tuned control of precision-recall trade-offs. Specifically, searches can be constrained to the union of matches resulting from any exact edit combination of insertion, deletion and substitution op-

G. Peterson and S. Shenoi (Eds.): Advances in Digital Forensics XIV, IFIP AICT 532, pp. 67-85, 2018.
https://doi.org/10.1007/978-3-319-99277-8_5

erations performed on the search term. The flexibility is leveraged in experiments involving fuzzy searches of an inverted index of the Enron corpus, a large English email dataset, which reveal the specific edit operation constraints that should be applied to achieve valuable precision-recall trade-offs. The constraints that produce relatively high combinations of precision and recall are identified, along with the combinations of edit operations that cause precision to drop sharply and the combination of edit operation constraints that maximize recall without sacrificing precision substantially. These edit operation constraints are potentially valuable during the middle stages of a digital forensic investigation because precision has greater value in the early stages of an investigation while recall becomes more valuable in the later stages.

Keywords: Email forensics, approximate string matching, finite automata

7.1 Introduction

Keyword search has been a staple in digital forensics since its beginnings, and a number of forensic tools support fuzzy search (or approximate string matching) algorithms that match text against keywords with typographical errors or keywords that are stringologically similar. These algorithms may be used to search inverted indexes, where every approximate match is linked to a list of documents that contain a match.

Great discretion must be used when utilizing these tools to search large datasets because strings that match (approximately) may be similar in a stringologial sense, but are completely unrelated in terms of their semantics. Even exact keyword matching produces an undesirable number of false positive documents to sift through, where maybe 80%-90% of the returned document hits could be irrelevant [23]. Nevertheless, the ability to detect slight textual aberrations is highly desirable in digital forensic investigations. For example, in the 2008 Casey Anthony case, in which Ms. Anthony was convicted and ultimately acquitted of murdering her daughter, investigators missed a Google search for a misspelling of the word “suffocation,” which was written as “suffication” [2].

Digital forensic tools such as dtSearch [58] and Intella [102] incorporate methods for controlling the “fuzziness” of searches. While the tools use proprietary techniques, it appears that they utilize the edit distance [115] in their fuzzy searches. The edit distance – or Levenshtein distance – is defined as the minimum number of elementary edit operations that can transform a string X to a string Y, where the elementary edit operations are defined as the insertion of a character, deletion of a character and substitution of a character in string X. However, precise control of the fuzziness of searches is often limited. In fact, it may not be clear what modifying the fuzziness of a search actually does other than the results “looking” more fuzzy. For example, some tools allow fuzziness to be expressed using a value between 0 to 10, without clarifying exactly what the values represent.

The research described in this chapter has two contributions. The first is the design and implementation of a novel constrained edit distance approximate search `cedas` algorithm, which provides complete control over the types and numbers of elementary edit operations considered in approximate matches. The flexibility of search, which is unique to `cedas`, allows for fine-tuned control of precision-recall trade-offs. Specifically, searches can be constrained to the union of matches resulting from any exact edit operation combination of insertions, deletions and substitutions performed on the search term.

The second contribution, which is a consequence of the first, is an experimental demonstration of which edit operation constraints should be applied to achieve valuable precision-recall trade-offs in fuzzy searches of an inverted index of the Enron Corpus [47], a large English email dataset. Precision-recall trade-offs with relatively high precision are valuable because fuzzy searches typically have high rates of false positives and increasing recall is simply obtained by conducting fuzzy searches with higher edit distance thresholds. The experiments that were performed identified the constraints that produce relatively high combinations of precision and recall, the combinations of edit operations that cause precision to drop sharply and the combination of edit operation constraints that maximize recall without sacrificing precision substantially. These edit operation constraints appear to be valuable during the middle stages of an investigation because precision has greater value in the early stages of an investigation whereas recall becomes more valuable later in an investigation [119].

7.2 Background

This section discusses the underlying theory and algorithms.

7.2.1 Approximate String Matching Automata

A common method for performing approximate string matching, as implemented by the popular agrep suite [209], is to use a nondeterministic finite automaton (NFA) for approximate matching. Since `cedas` implements an extension of this automaton, it is useful to discuss some key components of automata theory.

A finite automaton is a machine that takes a string of characters X as input and determines whether or not the input contains a match for some desired string Y.

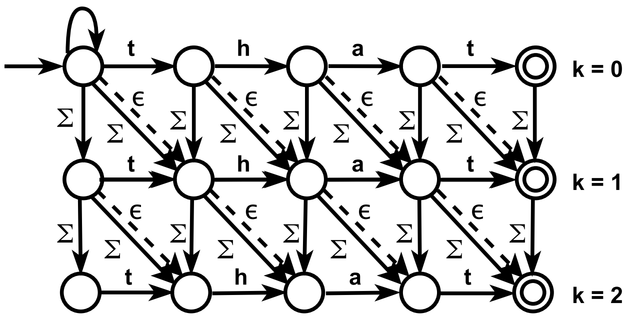


Figure 7.1: NFA matching the pattern "that" (allowing two edit operations).

An automaton comprises a set of states Q that can be connected to each other via arrows called transitions, where each transition is associated with a character or a set of characters from some alphabet Σ. The set of initial states $I \subseteq Q$ comprise the states that are active before reading the first character. States that are active check the transitions originating from themselves when a new character is being read; if a transition includes the character being read, then the state pointed to by the arrow becomes active. The set of states $F \subseteq Q$ correspond to the terminal states; if any of these states become active, then a match has occurred. The set of strings that result in a match are considered to be accepted by the automaton; this set is the language L recognized by the automaton.

Figure 7.1 shows the nondeterministic finite automaton for approximate matching A_L, where the nondeterminism implies that any number of states may be active simultaneously. The initial state of A_L is the node with a bold arrow pointing to it; it is always active as indicated by the self-loop. The terminal states are the double-circled nodes. Horizontal arrows denote exact character matches. Diagonal arrows denote character substitutions and vertical arrows denote character insertions, where both transitions consume a character in Σ. Since A_L is a nondeterministic finite automaton, it permits ϵ-transitions, where transitions are made without consuming a character. Dashed diagonal arrows express ϵ-transitions that correspond to character deletions. For approximate search with an edit distance threshold of k, the automaton has $k + 1$ rows.

The automaton A_L is very effective at pattern matching because it checks for potential errors in a search pattern simultaneously. For every character consumed by the automaton, each row checks for potential matches, insertions, deletions and substitutions against every position in the pattern.

For common English text, it is suggested that the edit distance threshold for approximate string matching algorithms should be limited to one, and in most cases should never exceed two [60]. This suggestion is well founded because about 80% of the misspellings in English text are due to a single edit operation[50].

Let $L_{k=1}$ and $L_{k=2}$ be the languages accepted by automaton A_L with thresholds $k = 1$ and $k = 2$, respectively. The nondeterministic finite automaton AT described in this section allows for different degrees of fuzziness that enable the exploration of the entire space between $L_{k=1}$ and $L_{k=2}$ in terms of the exact combinations of elementary edit operations applied to the search keyword. This automaton accepts the languages L_T, where $L_{k=1} \subseteq L_T \subseteq L_{k=2}$.

The automaton A_T is constructed in the following manner. The automaton that accepts $L_{k=2}$ can be viewed as the union of the languages accepted by each of its rows. For example, for an edit distance threshold of $k = 2$, the first row accepts the language comprising matches that have no edit operations performed on the keyword, the second row accepts the language of matches with one edit operation performed on the keyword and the third row accepts the language of matches with two edit operations performed on the keyword. The union of these subsets is a cover of $L_{k=2}$. An alternative cover of $L_{k=2}$ is the union of all the languages accepted by the automata for a specific number of insertions i, deletions e and substitutions s performed in a match such that $i + e + s \leq k$.

The following lemma proves the equivalence of the covers.

Lemma. Let L_k be the language accepted by an automaton such that k elementary edit operations are performed on a specified pattern. Let $L_{k=n}$ be the language accepted by the nondeterministic finite automaton for approximate matching with edit distance threshold n be equivalent to its cover $C_\alpha = \cup_{k=0}^{k=n} L_k$. Furthermore, let $L_{(i,e,s)}$ be equivalent to the language accepted by an automaton such that exactly i insertions, e deletions and s substitutions have been performed on a specified pattern. Let $C_\beta = \cup_{i,e,s:0 \leq i+e+s \leq n} L_{(i,e,s)}$. $C_\alpha = C_\beta$.

Proof. For all $x \in L_k$, there exists $L_{(i,e,s)}$ such that $x \in L_{(i,e,s)}$, where $i+e+s = k$. Therefore, $C_\alpha \subset C_\beta$. For all $x \in L_{(i,e,s)}$ such that $i+e+s = k$, $x \in L_k$. Thus $C_\beta \subset C_\alpha$. $\square$

By constraining the possible edit operations between the rows of the automaton A_T, each row of the automaton can correspond to a specific combination of i insertions, e deletions and s substitutions such that $i + e + s \leq k$ for edit distance threshold k instead of each row corresponding to some number of edit operations. This construction enables the accepted language L_T to be controlled by allowing terminal states $f \in F$ to remain in F or removing them from F. Specifically, some $L_{(i,e,s)}$ can be chosen to be not included in $C_\beta = \cup_{i,e,s:0 \leq i+e+s \leq n} L_{(i,e,s)}$.

7.2.2 NFA Definition

The constrained edit distance between two strings X and Y is the minimum number of edit operations required to transform X to Y given that the transformation obeys some pre-specified constraints T [152]. In general, constraints may be defined arbitrarily as long as they consider the numbers and types of edit operations. Let (i, e, s) be an element of T, the set of edit operations that constrain a transformation from string X to string Y , where (i, e, s) is an exact combination of edit operations. A_T may perform approximate searches where matches are constrained to the allowed edit operation combinations in T. For example, the search may be constrained to approximate matches derived from the edit operation combinations $(0, 0, 0)$, $(1, 0, 1)$, and $(0, 2, 0)$. The corresponding accepted language is $L_{(0,0,0)} \cup L_{(1,0,1)} \cup L_{(0,2,0)}$.

Figure 7.2 shows the constrained edit distance nondeterministic finite automaton A_T. It uses the same symbol conventions as the nondeterministic finite automaton in Figure 7.1, except that substitutions and insertions are expressed by diagonal and vertical transitions, respectively, where the transitions may go up or down. In order to ensure that each row $R_{(i,e,s)}$ of the automaton A_T corresponds to the accepted language $L_{(i,e,s)}$, it is necessary to engage the notion of a partially ordered set of multisets, which describes the edit operation transpositions that connect each row. The following definitions [80] are required:

Definition. Let X be a set of elements. Then, a multiset M drawn from set X is expressed by a function count M or C_M defined as $C_M : X \rightarrow N$ where N is the set of non-negative integers. For each $x \in X$, $C_M(x)$ is the characteristic value of x in M, which indicates the number of occurrences of elements x in M. A multiset M is a set if $C_M(x) = 0$ or 1 for all $x \in X$.

Definition. Let M_1 and M_2 be multisets selected from a set X. Then, M_1 is a submultiset of M_2 ($M_1 \subseteq M_2$) if $C_{M_1}(x) \leq C_{M_2}(x)$ for all $x \in X$. M_1 is a proper submultiset of M_2 ($M_1 \subset M_2$) if $C_{M_1}(x) \leq C_{M_2}(x)$ for all $x \in X$ and there exists at least one $x \in X$ such that $C_{M_1}(x) < C_{M_2}(x)$.

The set of multisets considered here comprises the elements (i, e, s), which implies that the multiset contains i insertions, e deletions and s substitutions. The cardinality of the multisets is no greater than the edit distance threshold k. The partial ordering of this set of multisets is the binary relation $\sim$, where for multisets M_1 and M_2, $M_1 \sim M_2$ means that M_1 is related to M_2 via $M_1 \subset M_2$.

Figure 7.3 presents the partially ordered multiset diagram D. Diagram D models the edit operation transitions between the rows of automaton A_T , where each multiset element (i, e, s) corresponds to row $R_{(i,e,s)}$ and each row of D corres-

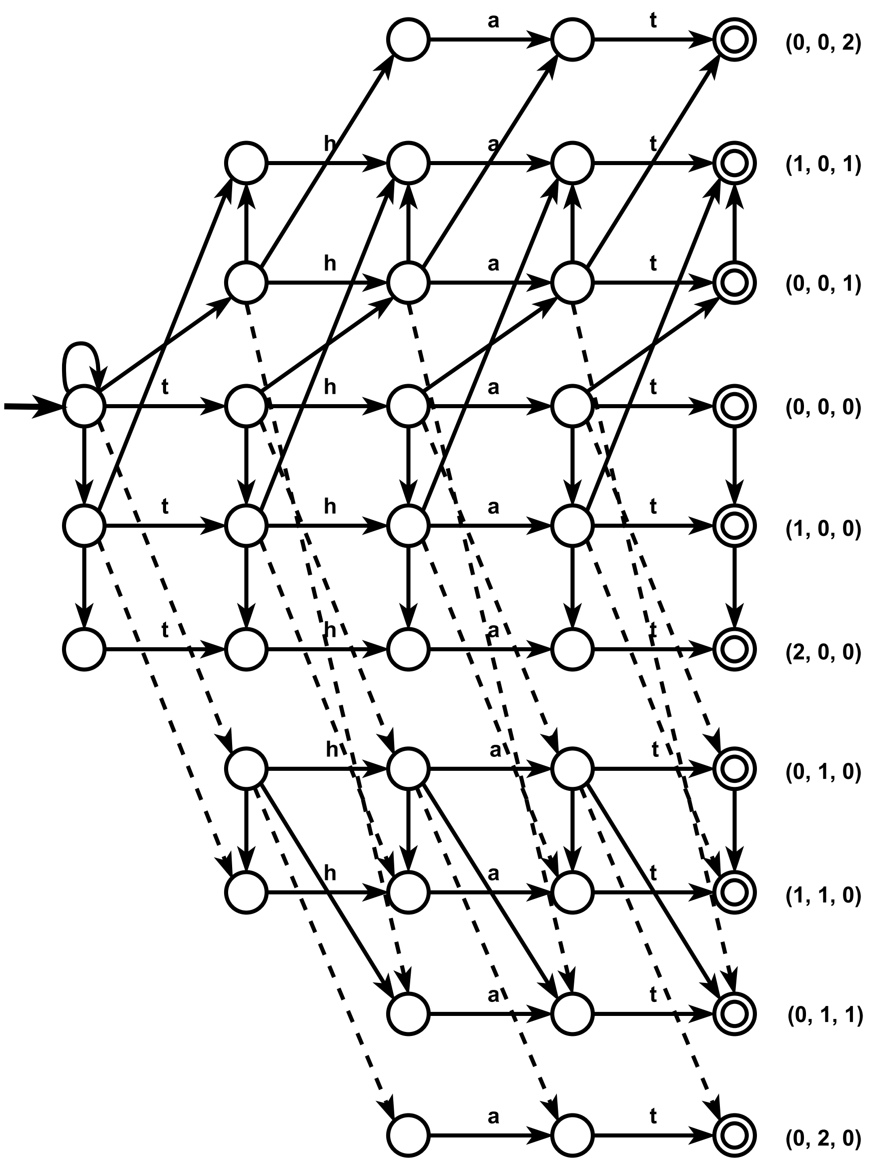


Figure 7.2: NFA matching the pattern “that”.

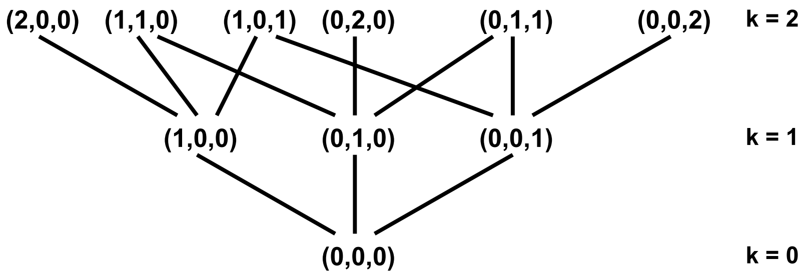


Figure 7.3: Partially ordered multisets (i, e, s)

ponds to a sum of edit operations. As seen in D, every $R_{(i,e,s)}$ has a specific edit operation transition sent to it from a specific row. In this way, each row $R_{(i,e,s)}$ can determine which (and how many) elementary edit operations are being considered in a match due to the partial ordering. For example, $R_{(1,0,0)}$ only has insertion transitions going to it from $R_{(0,0,0)}$, and $R_{(1,1,0)}$ only has deletion transitions going to it from $R_{(1,0,0)}$ (where one insertion has already taken place) and it only has insertion transitions going to it from $R_{(0,1,0)}$ (where one deletion has already taken place).

Finally, since the automaton is nondeterministic, it cannot be implemented directly using a von Neumann architecture.

7.2.3 Bit-Parallel Implementation

Bit-parallelism allows for an efficient simulation of a nondeterministic finite automaton. The method uses bit-vectors to represent each row of the automaton, where the vectors are updated via basic logical bitwise operations that correspond to transition relations of the automaton. Because bitwise operations update every bit in the bit-vector simultaneously, it updates the states in the row of the automaton simultaneously. If the lengths of the bit-vectors are not greater than the number of bits w in a computer word, then the parallelism reduces the maximum number of operations performed by a search algorithm by w [65].

In the bit-parallel nondeterministic finite automaton simulation, each row of the automaton for the search pattern X is expressed as a binary vector of length $|X|+1$; this also requires the input characters to be expressed as vectors of the same size. Input characters t_j are handled by bit-masks. Thus, a table of bit-masks $B[t_j]$ is created, where each bitmask represents the positions of the character t_j in pattern X. Table 7.1 shows the bit-masks when X = "that." Characters that are not present in the pattern are expressed using the "*" symbol.

Table 7.1: Bit-masks for the word “that”.

Character t_j	**Bit-mask** $B[t_j]$
a	01000
h	00100
t	10010
*	00000

Algorithm 5 presents the bit-parallel simulation of automaton A_T. The algorithm is an extension of the simulation of the nondeterministic finite automaton for approximate string matching using the unconstrained edit distance; this was first implemented by Wu and Manber [209]. Therefore, the components of the A_T simulation are similar, the primary modifications being the transition relationships between rows and the number of rows. The rows of the automaton are denoted by bitvectors $R_{(i,e,s)}$ and their updated values are denoted by $R'_{(i,e,s)}$. $R_(i,e,s)^B$ represents the Boolean values for whether or not row $R_{(i,e,s)}$ reports a match, which occurs when $R_{(i,e,s)}$ has a terminal state. For some i, e, and s, the value $R^B_{(i,e,s)}$ is true when $(i,e,s) \in T$

Each row is updated by first checking if the input character is an exact match for the row by computing $((R'_{(i,e,s)} << 1) \, \& \, B[t])$. This value is then bitwise ORed with potential transition relationships. $R_{(i,e,s)}$ checks for insertions, $(R'_{(i,e,s)} << 1)$ checks for deletions and $(R_{(i,e,s)} << 1)$ checks for substitutions from a row $R_{(i,e,s)}$. As mentioned above, incoming transitions for a row $R_{(i,e,s)}$ may be determined by checking against the incoming relations to the multiset (i,e,s) in diagram D in Figure 7.3. After updating all the rows, matches are checked by bitwise ANDing each of the allowed rows and determining if any bit representing the terminal states is set to one.

7.2.4 Evaluation

The additional flexibility in specifying fuzziness comes at the cost of time and space. The time and space complexities of `cedas` can be specified in terms of the number of multisets in diagram D for an edit distance threshold k. This number is equal to the number of rows in Algorithm 5, which is $f(k) = \frac{1}{6}(k+1)(k+2)(k+3)$. Therefore, for keyword searches shorter than 64 characters on an x64 architecture, the worst-case space complexity is cubic in k and the worst-case time complexity is $O(k^3 n)$, where n is the length of the text searched. Obviously, this implies that the algorithm should not be applied to time sensitive tasks with massive throughput such as intrusion detection, or applied to a live search of

Algorithm 5: NFA update algorithm

```
Initialize all rows $R'$ to 0 except $R'(0,0,0) \leftarrow$ 0x00000001,
  $R'(0,1,0) \leftarrow$ 0x00000002, and $R'(0,4,0) \leftarrow$ 0x00000004;
for each input character $t$ do
    $R_{(0,0,0)} \leftarrow R'_{(0,0,0)}$;
    $R'_{(0,0,0)} \leftarrow ((R'_{(0,0,0)} << 1)\ \&\ B[t])\ |\ \texttt{0x00000001}$;
    $R_{(1,0,0)} \leftarrow R'_{(1,0,0)}$;
    $R'_{(1,0,0)} \leftarrow ((R'_{(1,0,0)} << 1)\ \&\ B[t]])\ |\ R_{(0,0,0)}$;
    $R_{(0,1,0)} \leftarrow R'_{(0,1,0)}$;
    $R'_{(0,1,0)} \leftarrow ((R'_{(0,1,0)} << 1)\ \&\ B[t]])\ |\ (R'_{(0,0,0)} << 1)$;
    $R_{(0,0,1)} \leftarrow R'_{(0,0,1)}$;
    $R'_{(0,0,1)} \leftarrow ((R'_{(0,0,1)} << 1)\ \&\ B[t]])\ |\ (R_{(0,0,0)} << 1)$;
    $R_{(0,1,1)} \leftarrow R'_{(0,1,1)}$;
    $R'_{(0,1,1)} \leftarrow ((R'_{(0,1,1)} << 1)\ \&\ B[t])\ |\ (R_{(0,1,0)} << 1)\ |$
                $(R'_{(0,0,1)} << 1)$;
    $R_{(1,0,1)} \leftarrow R'_{(1,0,1)}$;
    $R'_{(1,0,1)} \leftarrow ((R'_{(1,0,1)} << 1)\ \&\ B[t])\ |\ (R_{(1,0,0)} << 1)\ |\ R_{(0,0,1)}$;
    $R_{(1,1,0)} \leftarrow R'_{(1,1,0)}$;
    $R'_{(1,1,0)} \leftarrow ((R'_{(1,1,0)} << 1)\ \&\ B[t])\ |\ R_{(0,1,0)}\ |\ (R'_{(1,0,0)} << 1)$;
    $R_{(2,0,0)} \leftarrow R'_{(2,0,0)}$;
    $R'_{(2,0,0)} \leftarrow ((R'_{(2,0,0)} << 1)\ \&\ B[t])\ |\ R_{(1,0,0)}$;
    $R_{(0,2,0)} \leftarrow R'_{(0,2,0)}$;
    $R'_{(0,2,0)} \leftarrow ((R'_{(0,2,0)} << 1)\ \&\ B[t])\ |\ (R'_{(0,1,0)} << 1)$;
    $R_{(0,0,2)} \leftarrow R'_{(0,0,2)}$;
    $R'_{(0,0,2)} \leftarrow ((R'_{(0,0,2)} << 1)\ \&\ B[t])\ |\ (R_{(0,0,1)} << 1)$;
    if $(((R'_{(0,0,0)}\&R^B_{(0,0,0)})|(R'_{(0,0,1)}\&R^B_{(0,0,1)})|(R'_{(0,1,0)}\&R^B_{(0,1,0)})|$
          $(R'_{(1,0,0)}\&R^B_{(1,0,0)})|(R'_{(0,1,1)}\&R^B_{(0,1,1)})|(R'_{(1,0,1)}\&R^B_{(1,0,1)})|$
          $(R'_{(1,1,0)}\&R^B_{(1,1,0)})|(R'_{(2,0,0)}\&R^B_{(2,0,0)})|(R'_{(0,2,0)}\&R^B_{(0,2,0)})|$
          $(R'_{(0,0,2)}\&R^B_{(0,0,2)}))\&(0x00000001 << n-1))$ then
        Match is Found;
    end
end
```

massive forensic data with high edit distance thresholds k. However, this is sufficient for specifying the fuzziness of approximate searches over an index of emails, because the values of k should be low and the index acts as a data reduction mechanism. The experimentation described in the next section demonstrates that the algorithm runs approximately six times slower than `agrep` with an edit distance threshold of $k = 2$.

7.3 Experiment Methodology

The flexibility of `cedas` was evaluated by specifying fuzziness in the context of an investigation of the Enron email dataset [47] and assessing the effectiveness of the constraints. The email messages were converted to the mbox format to simplify processing. Only the contents of the email bodies were examined.

In order to search the data, an inverted index of the emails was created using the `mkid` program [70], which yielded a database of index tokens. After deduplicating the tokens in the index, all the tokens were output to a single text file that was searched using `cedas`. Note that the choice of indexing algorithm affects the list of tokens because different algorithms may interpret delimiters differently and, therefore, would affect any search. The list of tokens used in the experiments totaled 460,800 unique words.

Twenty-eight different keywords related to the 2001 Enron scandal were used. This list of keywords was not compiled by a digital forensic investigator, so the choice of keywords could be improved. Keywords were chosen that were relevant to the case, but would not obviously produce an overwhelming number of false positives. Furthermore, no keyword that contained less than six characters was chosen.

Unconstrained fuzzy searching with an edit distance threshold of $k = 2$ for small keywords produces massive lists of words (often exceeding 10,000 words) that have to be analyzed manually. This can be viewed as a limitation of the proposed approach. Table 7.2 shows the list of search keywords. Many of the words were taken from Rodger Lepinsky's webpage on data science and the Enron corpus [114].

A case-insensitive fuzzy search for each keyword was conducted on the list of index tokens, and each search was done under 64 different constraints. A match occurred if a keyword was found as a substring of an index token with the allowed tolerance of edit operations as specified by the constraints in terms of $(i, e, s) \in T$. All the approximate matches found in the index were returned in a list. The elements (i, e, s) that were possibly not included in T were those in which the sum $i + e + s$ was equal to the edit distance threshold $k = 2$. Specifically, the automaton accepted $L_{(0,0,0)}, L_{(1,0,0)}, L_{(0,1,0)}, L_{(0,0,1)}$, but the inclusion of all possible

Table 7.2: Keyword list

Word Length	Keywords
6	Cuiaba
7	BlueDog, BobWest, corrupt, illegal, launder, Sarzyna, scandal
8	bankrupt, Backbone, Fishtail, Margaux1, Shutdown, subpoena, Velocity, unlawful
9	collusion, Whitewing, Yosemite
10	Catalytica, conspiracy, KennethLay, litigation, reputation, suspicious
>10	ArthurAndersen, illegitimate, talkingpoints

combinations of languages $L_{(i,e,s)}$ such that $i + e + s = 2$ was allowed.

The effectiveness of a search for each constraint on each keyword was measured in terms of the precision and recall derived from the list of returned approximate matches. To understand the overall effectiveness of each constraint, the average precision and recall results for all the keywords under each constraint were computed as harmonic means. The harmonic mean was chosen because the arithmetic mean produced overly optimistic results for fuzzy searches under the chosen constraints.

7.3.1 Interpreting Match Results

Precision and recall have been used by researchers to gauge the effectiveness of approximate string matching algorithms [27] [178]. Precision is the proportion of retrieved items that are relevant whereas recall is the proportion of total relevant items retrieved [90]. As recall increases, precision tends to decrease. Precision and recall are expressed as:

$$\text{Precision} = \frac{|(\text{retrieved items}) \cap (\text{relevant items})|}{|(\text{retrieved items})|} \tag{7.1}$$

$$\text{Recall} = \frac{|(\text{retrieved items}) \cap (\text{relevant items})|}{|(\text{relevant items})|} \tag{7.2}$$

The precision and recall metrics are useful, but they are not perfect because the notion of relevance is subjective. Relevant terms are defined as being variations of the original term (e.g., obtaining "litigating" when searching for "litigation"), closely related to the original term in a semantic sense (e.g., obtaining "legalese" when searching for "illegal"), or misspellings of the original term. However, if a keyword is a substring of the examined index token and is clearly unrelated, then it is not considered relevant. For example, if the search term is "audit" and a hit is obtained for "AudiTalk," then the hit is not relevant. For this reason, the classification of relevant versus non-relevant hits for approximate hits is always a manual process.

Another shortcoming of the metrics is that it is not possible to compute the true precision and recall for every keyword; this is because the number of relevant words for each keyword in the Enron dataset is unknown. Therefore, a compromise was employed: the number of total relevant items was set to be equal to the items identified for each keyword for unconstrained approximate matching at the edit distance threshold $k = 2$. This implies that unconstrained approximate matching with $k = 2$ yields 100% recall, which is not necessarily true.

Finally, it is important to note that approximate string matching results are highly dependent on the specific data being matched and the keywords being used [49]. Therefore, utilizing `cedas` on an inverted index that was not derived from an English email corpus may produce different results.

7.4 Experimental Results

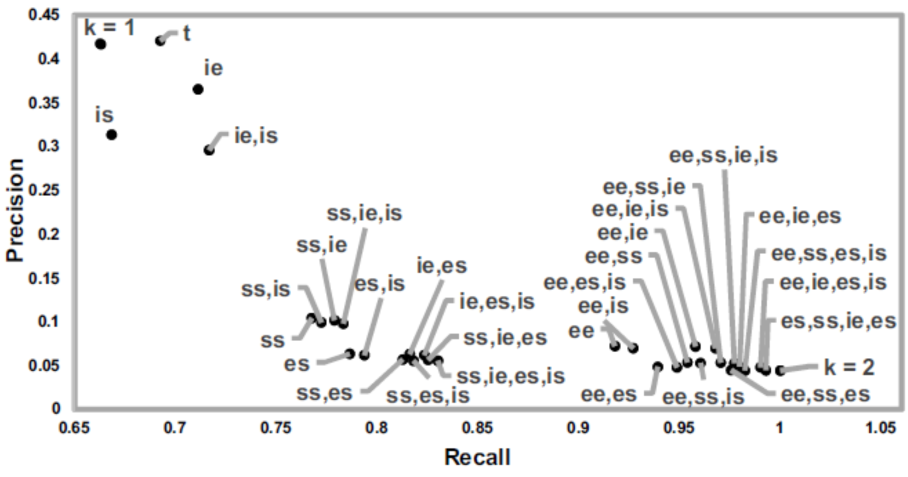
This section describes the experimental results.

7.4.1 Precision and Recall

The results in this section reflect the effectiveness of each set of constraints T in terms of precision and recall, and identify the constraints that produce valuable results for investigating an English email corpus. Figure 7.4 shows the precision-recall trade-off curve for various constraints. Data points labeled $k = 1$ and $k = 2$ represent the results for unconstrained fuzzy searches with edit distance thresholds set to one and two, respectively.

As expected, the application of constraints to fuzzy searches of the Enron inverted index resulted in higher recall than an unconstrained fuzzy search with an edit distance threshold of $k = 1$, and better precision than an unconstrained fuzzy search with an edit distance threshold of $k = 2$. However, the primary interest is in the precision-recall trade-offs that are useful in an investigation. As mentioned in the introduction, a common complaint is the number of false positives produced by a fuzzy search, and the fact that precision is valued more than recall early in an investigation [119]. This implies that the constraints that produce results with relatively high precision are most useful in an investigation, because increased recall is easily obtained by increasing the edit distance threshold.

What is immediately apparent from the data is that there are several distinct clusters of data points, where each cluster is associated with different edit operation com-



$(1,1,0) = \text{ie}$; $(1,0,1) = \text{is}$; $(0,1,1) = \text{es}$; $(2,0,0) = \text{ii}$; $(0,2,0) = \text{ee}$; $(0,0,2) = \text{ss}$

Figure 7.4: Precision-recall trade-off curve applying various constraints

binations. The cluster with the highest precision comprises all the data points near data point $k = 1$, where $(0,1,1), (0,0,2), (0,2,0) \notin T$. This means that matches under these constraints did not include edit operations with exactly two substitutions, a substitution and deletion, or two deletions performed on the keyword. It follows that, in order to preserve the precision of a fuzzy search to be near the unconstrained case of an edit distance threshold of $k = 1$, it is necessary to constrain the fuzzy search to edit operations that do not include the previously mentioned edit operation combinations. Furthermore, the data points in the cluster mostly show a marked improvement in recall. Because of the relatively high precision and recall, it can be posited that the constraints in this cluster are useful in the middle stages of an investigation.

Data points in this cluster that include a single insertion and deletion (ie) have very good precision-recall tradeoffs. To ensure that the results of a fuzzy search with these constraints are simply not due to the transposition edit operations for which adjacent characters may be swapped, the same tests were conducted using the `nrgrep` [139] algorithm to perform an unconstrained fuzzy search allowing transpositions with an edit distance threshold of $k = 1$. These results are represented by data point t and it can be seen that ie and t do not yield the same results.

7.4.2 Execution Time

The speed of `cedas` was evaluated by timing the unconstrained fuzzy search with an edit distance threshold of $k = 2$ for every keyword in the Enron inverted index.

Table 7.3: Average execution times for `agrep` and `cedas`.

`agrep`	`cedas`
0.0477142857 s	0.2795714286 s

The average of these results was computed and compared with the average of the results obtained using `agrep`.

The results in Table 7.3 demonstrate that cedas executed nearly six times slower than `agrep` for an edit distance threshold $k = 2$. However, it should be noted that the `cedas` implementation was not optimized; therefore, it has the potential to run faster than measured.

7.4.3 Analysis and Suggestions

The gap in precision between the higher and lower data clusters is potentially shaped by the statistics of the English language. Additional experimentation is necessary to confirm this observation, but it is clearly easy to transform words to other words using many substitutions or deletions. By limiting the application of deletion and substitution edit operations, the structure of the original word is preserved. For this reason, if somewhat high precision is needed, it is appropriate to use a fuzzy search that does not include edit operations involving two deletions, two substitutions, or a single deletion and a single substitution.

To maximize the recall in the fuzzy search results without sacrificing precision as seen when applying many of the edit operation combinations, it is suggested that the set of constraints T should contain (0,0,0), (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), (2,0,0); for the sake of brevity, the language accepted by this automaton is denoted by $L_{-(ee,ss,es)}$. The precision and recall of this language correspond to the data point ie, is. If more precision is needed with nearly as much recall, then the set of constraints T should contain (0,0,0), (0,0,0), (1,0,0), (0,1,0), (0,0,1), (1,1,0), (2,0,0) (whose precision and recall correspond to data point ie). Ultimately, `cedas` users should choose constraints based on the precision recall trade-offs they are willing to tolerate.

7.5 Related Work

Fuzzy search algorithms are implemented in digital forensic tools such as dtSearch [58] and Intella [102]. However, the variables for setting the tolerated fuzziness in these tools do not always correlate directly with the edit distance thresholds. Whether or not these tools employ constrained edit distance algorithms is unknown because their techniques are proprietary. Nevertheless, they appear to be combining the edit distance measure with other types of distance measures, natural language processing and/or information retrieval techniques.

The `agrep` [208] and `nrgrep` [139] open-source tools may be used for fuzzy searches in digital forensic investigations. The tools incorporate various approximate matching algorithms, including edit-distance-based approximate matching, prefix matching, regular expression matching and other options. They can be considered to represent the cutting edge of bit-parallel nondeterministic finite automaton implementations for approximate matching in terms of speed and utility. However, the primary advantage of `cedas` compared with the edit distance matching algorithms used by the tools is its flexibility in constraining edit operations. The `agrep` tool does not implement constraints; as a result, specifying fuzziness in terms of edit distance operations is limited to setting the edit distance threshold. The `nrgrep` tool is more flexible in that it allows a user to set an edit distance threshold, use transpositions and define a subset of the edit operations used in a search. The last feature essentially constrains edit operations, thereby producing a subset of possible edit operation constraints as in the case of `cedas`. However, this type of matching cannot return results equivalent to $L_{-(ee,ss,es)}$.

Other researchers have also proposed constrained edit distance search algorithms. For example, Chitrakar and Petrovic [40] [41] have specified constrained edit distance search algorithms that employ row-based bit-parallelism. One algorithm specifies edit operation constraints in terms of the maximum number of allowed indels [40] (sum of insertions and deletions). A second algorithm expresses edit operation constraints in terms of the maximum allowed number of insertions, deletions and substitutions permitted in a match [41]. These algorithms also engage subsets of possible constraints permitted by `cedas`, but they cannot specify constraints that yield the language $L_{-(ee,ss,es)}$. Experiments by Chitrakar and Petrovic reveal that their algorithms are nearly as fast as `agrep`.

7.6 Conclusion

This chapter has presented `cedas`, a novel constrained edit distance fuzzy search algorithm that performs approximate searches where the possible transformations on the search terms are constrained to any set of edit operation combinations with exactly i insertions, e deletions and s substitutions. The algorithm is a bit-parallel simulation of a nondeterministic finite automaton, in which the rows of the automaton are defined not by the number of elementary edit operations considered, but by the numbers and types of edit operations. This flexibility in defining edit operation constraints for approximate search is unique to `cedas`.

Experiments employed the `cedas` algorithm to perform constrained edit distance fuzzy searches for a list of keywords in an inverted index of the Enron email corpus. The average precision and recall results of searches applying various edit operation combination constraints identified the constraints that were the most valuable for fuzzy searches of an English email dataset. Because a common complaint against fuzzy search is its large number of false positives, edit operation constraints that yield high precision would be valuable in digital forensic investigations.

The experiments revealed that, in order to avoid the precision drop commonly seen in unconstrained fuzzy search at an edit distance threshold of $k = 2$, it is necessary to constrain fuzzy search to not include any matches involving two deletions, two substitutions, or a substitution and deletion. Fuzzy searches with an edit distance threshold of two and whose constraints did not include the previously mentioned edit operation combinations produced relatively high combinations of precision and recall, where the precision is somewhat reduced with an unconstrained edit distance threshold of $k = 1$ while also improving recall. To maximize the recall of a fuzzy search without significant precision reduction, the combination of edit operations (i, e, s) should be constrained to (0,0,0), (1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1) and (2,0,0). These findings should be useful in the middle stages of an investigation because precision has greater value in the early stages and recall becomes more valuable later in an investigation [120].

The flexibility of `cedas` comes at a cost. The worst-case space complexity of the algorithm is cubic in k and the worst-case time complexity is $O(k^3 n)$ for searching keywords of length less than 64 characters on an x64 architecture, where k is the edit distance threshold and n is the length of the searched text. During the experiments, it was discovered that the average time taken to perform an unconstrained approximate search with edit distance threshold $k = 2$ on the inverted index of the Enron dataset and return the list of approximate matches increased from about 0.0477 seconds with `agrep` to about 0.2796 seconds with `cedas`. It is important to note that the `cedas` implementation has not been optimized. In fact, the space and time requirements can be reduced by dynamically generating the rows of the automaton that are necessary instead of simply removing terminal states from specific rows.

The implementation of cedas in a hardware architecture targeted for nondeterministic finite automata (e.g., Automata Processor [56]) could potentially run in linear time. Tracy et al. [198] have shown that the nondeterministic finite automaton for approximate matching (Figure 7.1) runs in worst-case linear time on the Automata Processor, where the hardware could maximally handle a nondeterministic finite automaton with a search pattern length of 2,730 characters with an edit distance threshold $k = 4$. The fastest bit-parallel nondeterministic finite automaton simulations of the same type of automaton require that search patterns do not exceed about 30 characters to preserve optimal results with a worst-case time complexity of $O(\lceil(m-k)(k+1)/w\rceil n)$ [100], where m is the length of the search pattern, k is the edit distance threshold and n is the length of the input. Finally, other improvements, such as those employed by other search tools, can also be made to `cedas`; these include prefix matching and character-specific fuzziness. Finally, other improvements, such as those employed by other search tools, can also be made to cedas; these include prefix matching and character-specific fuzziness.

Acknowledgments

This research was supported by the Research Council of Norway Program IKT-PLUSS under the R&D Project: “Ars Forensica – Computational Forensics for Large-Scale Fraud Detection, Crime Investigation and Prevention” (Grant Agreement 248094/O70).

Chapter 8

Paper 3: Generic Metadata Time Carving

Nordvik, R., Porter, K., Toolan, F., Axelsson, S., & Franke, K. (2020). Generic Metadata Time Carving. Forensic Science International: Digital Investigation, 33, 301005

Abstract

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

8.1 Introduction

File carving is a technique which identifies and extracts files from unallocated areas based on signatures found within the file content, and not by using file system metadata [74]. While extremely useful, file carving has a few challenges. First, investigators need to decide which file type to carve. To decrease the file carving time, investigators often select the file types they assume could be pertinent for the criminal case. For instance, by carving for typical image files in cases related to sexual abuse of children, the investigator limits the ability to identify other file types. Furthermore, not all files have a signature, and will not be found by using file carving. Some carving techniques will carve based on the assumption that files have contiguous blocks, which will fail when trying to carve a fragmented file [74].

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

We only recover file or directory metadata from `NTFS` and `Ext4` to demonstrate the usability of the novel approach, but the approach can be extended to recover metadata from other file systems. In order to achieve a realistic scenario we re-format the volume with another file system, effectively "damaging" the previous file system. The tools developed in this paper are prototypes, and the main target group are file system experts with the competence to manually assess file system structures. The tools and the disk images can be downloaded for review at [150].

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

Our approach focuses on metadata structures found in `MFT` entries or inodes carved from unallocated space. There will always be a risk that the blocks (clusters) pointed to by the discovered metadata structures may be overwritten by new or existing allocated files, but this may be identified by examining the allocated file system bitmap for allocated blocks, and by comparing metadata information with the content of the recovered file. Most file systems have some sort of bitmap system which has bits representing each block (cluster), and allocated blocks have their corresponding bit set [34, p. 311].

Our new approach is suited for recovery of metadata and file content from storage

File System	Co-located timestamps	Granularity
NTFS [34]	4	64 bit - ns intervals since 1.1.1601
ReFS [149]	4	64 bit - ns intervals since 1.1.1601
APFS [95]	4 (5)	64 bits - ns since 1970
HFS+ [10]	4	32 bits - s since 1904
BTRFS [26]	3 (4)	64 bits - s since 1970 + 32 bits (ns)
ExFAT [91]	3	32 bits + UTC offset
FAT [34]	3	16 bits for time (except accessed), 16 bits for day
UFS1 [34]	3	32 bits - s since 1970 + 32 bits (ns)
UFS2 [34]	4	64 bits (ns) since 1970
Ext2/3 [34]	4	32 bits - s since 1970
Ext4 [55]	4	34 bits - s since 1970 + 30 bits (ns) [89]

Table 8.1: File Systems with timestamps co-located within metadata structures

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

Detailed file system structures are described by Carrier [34]. Even though his book does not include details about `Ext4`, it contains most of the basic information from `Ext2` and `Ext3`. Dewald and Seufert [55] include more details about `Ext4`.

8.1.1 Assumptions

Currently, most file systems will include at least 3 contiguous timestamps. Linux file systems normally use the `MAC` (Modified, Accessed and Changed) timestamps [34, p. 297], for instance `Ext2` and `Ext3` use the contiguous `atime` (accessed), `ctime` (inode changed), `mtime` (data modified) and `dtime` (deleted) [34, p. 298]. `Ext4` also contains the same contiguous timestamps, but adds the `crtime` (creation) in the end of the inode [63]. `NTFS` and `ReFS` use 4 contiguous timestamps (Creation, Modified, `MFT` modified, and Accessed) in multiple attributes [34, 149]. Table 8.1 shows a few file systems with closely co-located timestamps.

8.1.2 Objectives

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

We aim to identify file or directory metadata structures from different file systems based on a common identifier. In our case, equal and closely co-located timestamps, which will allow for a generic approach for metadata carving. We identify the potential timestamps by using a simple string matching algorithm, and then we interpret the semantics[1] of the expected file system metadata in order to significantly reduce the number of false positives.

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

8.1.3 Novelty of the new approach

Existing techniques for metadata carving do not use timestamps as a common identifier (dynamic signature) for different file systems. Even Dewald and Seufert [55] describe that there is no magic signature for inodes in the `Ext4` file system, and they depend on semantics from `Ext4` in order to identify the locations of the inode metadata structures. However, metadata structures can easily be found by using string pattern matching based on equality, but unfortunately with a large number of false positives which have similar properties. Thus, obtaining high recall and low precision for finding file system metadata entries. We also do not depend on a start date or an end date to identify the timestamps. This datetime agnostic nature of our approach allows the support of any file system that has closely co-located timestamps. While we do not depend on other semantics in order to identify the locations of these potential timestamps, we do utilize semantic parsers to validate and reduce the number of false positive hits of file system metadata structures significantly.

8.1.4 Importance for Digital Forensics

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

8.1.5 Organization of this paper

We have introduced the objectives and the novel generic metadata time carving approach in the Introduction section. In the Related work and contributions sec-

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

tion we discuss the current state of the art related to file and metadata carving. In the Method section we describe the carving algorithms and the methods we used for the experiments, and in the Results section we describe our results using precision and recall. Then we discuss our results in the Discussion section, and we conclude in the Conclusion and further work section.

8.2 Related work and contributions

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

Muller [135] introduced the idea of searching for `NTFS` timestamps as a string in unallocated space, since each timestamp is 64 bits and represents the number of nanosecond intervals since 1.1.1601. He also describes that the timestamps are in groups of 4 contiguous timestamps for each group. He created an `EnScript` (plugin for `EnCase`) that searched for the `NTFS` timestamps and bookmarked them. The `EnScript` uses a `grep` search for a particular date range, and it has an option for checking the next 8 bytes in order to only include hits that are followed by another valid timestamp. Use of the consecutive timestamps search ideally reduces the number of false positives, but based on the comments on this blog post it appears as though the consecutive timestamps search does not work correctly.

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

McCash [128] based his work on the EnScript from Muller [135], and adds the idea of using this information to detect MFT records and their attributes to extract the data content. He also describes that the script can be used to identify directory indexes and Registry key nodes.

8.2.1 Metadata carving

Dewald and Seufert [55] consider the case in which the `Ext4` superblock or group descriptor table is corrupted or overwritten, and they use either metadata mode or content mode for parsing file systems or metadata carving respectively. In content mode their solution is to carve for inodes, which potentially provides the metadata necessary to extract the file content. However, the filename and inode number is not recovered in content mode. Since inodes have no magic bytes (except for extent headers in `Ext4`), they describe that they carve for them using pattern matching and analysis of the metadata. They conclude that their approach can reconstruct files from `Ext4` despite not knowing about the specific structure of the file system. They do, however, describe that they need multiple `Ext4` parameters in metadata mode for file system parsing. They describe that these can either be given by the user, or estimated based on the file system size.

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

Plum and Dewald [163] describe carving for APFS container superblocks, volume superblocks, or inode carving. APFS uses multiple container superblocks, and each of them may contain a reference to the previous container superblock. Within each container superblock they find volume superblocks, which describe specific volumes. These can be used to parse a specific volume and recover files from previous states of the file system. They further describe that inodes do not have a specific signature, but they can be carved using a combinations of the object type and subtype inode fields. These inodes can be used to potentially recover files with the connected metadata.

Their approach is similar to the work of Dewald and Seufert [55], but it differs by depending on specific semantics from APFS. Our generic approach will also work for the APFS inodes, since each inode has a set of contiguous timestamps. However, we have not implemented a semantic parser for APFS.

Work by Garfinkel [76] describes the Bulk_Extractor tool which parses a large stream of data, using multiple threads, for feature extractions (URLs, e-mail addresses, Google search terms, Exif data, etc), which utilizes optimistic decompression before extracting the features. The features are detected based on rules which consider local context, which improve precision and recall. The features extracted do not need to be found within file entries. As part of the result, histograms of extracted features are created.

8.2.2 Evaluating recovered files

Casey et al. [38] describe forensic processes such as authentication, classification and evaluation of recovered files. The problem is that different recovery tools do not use the same names for the same thing. They suggest to use Potentially

Recovered before the authentication is performed. The authentication process is necessary in order to decide if the file is Fully Recovered, Partly Recovered, only Name and Metadata recovered, or Name Recovered. The decision should be based on confidence level after testing or trying to falsify different scenarios or claims.

8.3 Method

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

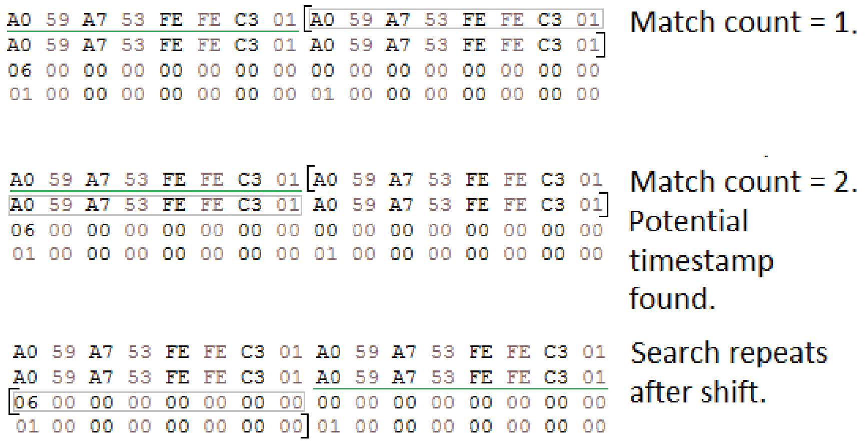
As a proof of concept, we have added support for the recovery of metadata based on the identified timestamps in the `Ext4` and `NTFS` file systems.

8.3.1 General Potential Timestamp Algorithm Description

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

The mechanics of the search algorithm are based on the *sliding window* approach as is often found in malware analysis. The search begins at $T[0]$, in which the first m bytes are taken to be a potential timestamp, which contains the values $T[0 : m)$. We then check if this m byte keyword is equivalent to every non-overlapping m bytes in $T[m : (m + k))$, and keep count of how many exact matches have occurred. Given that we are searching for timestamps where at least two of them per metadata structure are equivalent, if no matches are found, we would then advance our search position by m bytes to position $T[0 + m]$. The advancement of m bytes assumes that timestamps will always fall on a multiple of m, and we do plan in implementing an *exhaustive* search functionality which checks for timestamps on every one or two bytes. Such skip sizes were chosen to enhance the speed of search substantially, as the current solution for 8 byte timestamps will process a disk image 8 times faster than an exhaustive search alternative that checks for potential timestamps on every single byte. If one or more matches are found we advance our search position by k bytes to position $T[0 + k]$. In either case, the entire search procedure is repeated from our new search position. This process is repeated for the entirety of the data T, except last k threshold of bytes. The skip size of k was chosen as it is the minimum size to avoid multiple hits for the same metadata structure. Note, for our current implementation k must be a multiple of m, otherwise the bytes being searched will be misaligned with the actual disk image timestamps. Algorithm 6 provides the pseudocode of the basic potential timestamp carving algorithm.

We provide an illustration for further explanation. In Figure 8.1, the underlined bytes represent the potential timestamp keyword with $m = 8$, and the brackets represent the threshold of bytes, $k = 24$, being searched for matches.


Match count = 1.
Match count = 2. Potential timestamp found.
Search repeats after shift.

```
A0 59 A7 53 FE FE C3 01 [A0 59 A7 53 FE FE C3 01
A0 59 A7 53 FE FE C3 01  A0 59 A7 53 FE FE C3 01]
06 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00

A0 59 A7 53 FE FE C3 01 [A0 59 A7 53 FE FE C3 01
A0 59 A7 53 FE FE C3 01  A0 59 A7 53 FE FE C3 01]
06 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
01 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00

 A0 59 A7 53 FE FE C3 01  A0 59 A7 53 FE FE C3 01
 A0 59 A7 53 FE FE C3 01  A0 59 A7 53 FE FE C3 01
[06 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 01 00 00 00 00 00 00 00] 01 00 00 00 00 00 00 00
```

Figure 8.1: Visual representation of the search procedure where three matching time stamps are searched for. The underlined byte sequence represents the current byte sequence being tested as a possible timestamp. The subsequent bytes in brackets represent the search threshold for checking matches. The bytes in grey boxes represent checks for matching byte sequences. In the second row, after a second match is found, we advance the search procedure ahead by k bytes, where the process is repeated.

This general search by itself likely produces a large number of false positives, thus

Algorithm 6: Basic Potential Timestamp Carving Algorithm.

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

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

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

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

8.3.2 Practical Potential Timestamp Program Details

The general timestamp carving algorithm was implemented in C++, which we refer to as *cPTS*, and is supported by a number of libraries. Since disk images under analysis are likely greater than memory, we use the cross-platform mio memory mapping library[2]. This allows us to read in 1 GB of memory at a time, and read the image as a series of arrays. However, once the potential time stamp carver arrives within the last 4096 bytes of the gigabyte in memory, we load a new gigabyte into memory from the search point relative to the disk, as to handle directory entries that are spread across segments. For converting datetime formats into decimal form, we used the Date library[3]. The program outputs a text file list of all the potential timestamp locations (in byte offset) that were found.

[2]https://github.com/mandreyel/mio

[3]https://github.com/HowardHinnant/date

8.3.3 Semantic parsers

Identifying closely co-located potential timestamps based on equality will provide generic results, but will contain many false positives due to its genericity. Therefore, parsers which utilize the semantics of the expected metadata structures of specific file systems were developed for more accurate automation. Our Python 3 parsers accept the timestamp locations from the generic timestamp carver and the disk image as input, as seen in Figure 8.2. Our experiments utilize this process.

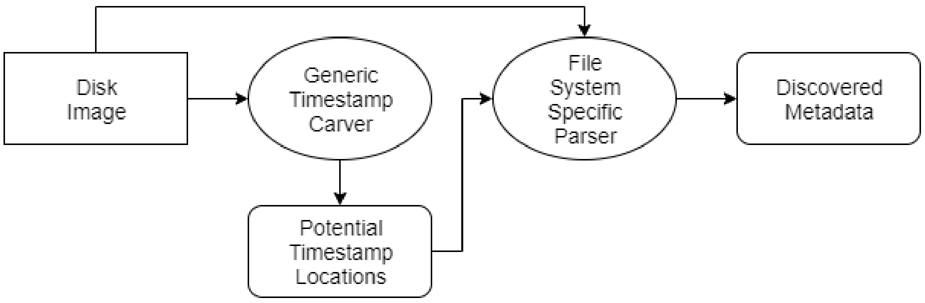


Figure 8.2: Diagram for system deployment, used in our experiments.

NTFS semantic parser

The `NTFS` script assesses if the potential timestamp is within the Standard Information Attribute (SIA), or a Filename Attribute (FNA). To reduce the number of false positives, we exclude any potential timestamps from before the year 1970 and years beyond 2100. The script outputs metadata information contained in the SIA, FNA, and Data Attribute if possible. When attempting to parse out a full MFT entry, we start with the identification of the SIA. Once we identify the location of the SIA header, we use the length of the SIA to see what the header of the next attribute is, ultimately searching for the Data Attribute. If the next attribute header is not the Data attribute, and the first byte of the attribute type is less than 0x80, we read the length of the attribute and perform another skip down to the next attribute. Though, if the next attribute is an FNA, we will output its metadata information, and add the byte location of its first timestamp to a list of future timestamps to avoid. This is done so we do not get redundant FNA outputs. Encountering at least one FNA is required to read out a potential Data Attribute we encounter. This is repeated until we find the Data attribute, or abandoned if we identify an attribute type where its first byte is greater than 0x80 or if we have searched more than the length of the potential MFT entry. A limitation of this work is that we do not currently perform MFT entry searching starting from identified File Name Attributes, where their timestamps are more likely to be reliably[4] found due to their relatively unchanging nature compared to Standard Information Attributes [44]. Another limitation is that we currently do not support Alternate Data Streams. Relevant MFT entry information is output into the file `NTFSResults.txt`, and if the file is resident, we also include the resident file encoded in ASCII.

Ext4 semantic parser

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

Like the `NTFS` parser, we use the potential timestamps as anchor points and test for various semantics at local offsets. But now, we also verify information found in likely directory entries. For `Ext4`, we test all possible offsets backwards for file flags of interest: 0x04 for directories, 0x08 for regular files, and 0x0A for symbolic links. For `Ext4` inodes not using extents, we ensure the relative position of bytes 36-39 of the inode are unused. For inodes using extents, we check that the relative position of its extent header magic number is equal to 0xF30A. We then conduct additional tests to increase the likelihood of having discovered an inode, such as using some of Dewald and Seufert's [55] timestamp consistency tests, that the size of the file appropriately fits the sector count, and that the size of the file cannot exceed the size of the disk image. All inodes found to be sufficiently valid have their likely starting points added to a list.

The great difficulty in performing full file extraction in `Ext4` is connecting inodes and their directory entries when not relying on superblocks, block group descriptor tables, or inode bitmaps. Such connections will need to be made if these metadata structures are irrecoverable. The inode contains the majority of the metadata for the file, but its associated directory entry contains the filename and the inode number. The task was then to connect inodes based on their physical position to their actual inode number. We pursued solving this problem solely relying on information we can find locally within or around a validated inode.

Our solution revolves around using the verified inodes of directories that have not been deleted, as this gives us ground truth information about inode numbers and filenames, including the inode number of the directory itself. Verification is performed by following the directory's extent or direct block pointer to its first directory entry and checking if bytes 4-6 are 0x0C0001 (the length of the entry and

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

the first byte of the length of the name). For `Ext4`, we perform two passes over the disk image. The first pass collects information on valid inodes, creating a dictionary of inode numbers and filenames found in all validated directories, as well as a so-called synchronization list. This synchronization list is a recording of the first validated inode of a directory found per block group, wherein we record the inode's location and inode number. The second pass uses the inode dictionary and the synchronization list to make inode number estimations of validated inodes, outputting the inode number alongside its likely filename.

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

$$e = dn + ((vl - dl)/256) \tag{8.1}$$

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

The second way of estimating inode numbers uses the previous estimations, and the inode dictionary. The first time we make an estimate for a particular inode number, we update its entry in the inode dictionary by adding in the inode's file version number and created time as parameters. If the inode number did not exist in the dictionary prior to the inode estimation, we simply create an entry with these parameters. Once the entry has been updated or created, it cannot change. The file version number and the created time should be relatively unique per inode, and so when parsing future inodes we check if we have already recorded its file version number and creation time in the inode dictionary, and write out the associated recorded inode number and filename.

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

8.3.4 Experimental setup

We used an external USB thumb drive and wiped the partition[5] using the tool `dc3dd v. 7.2.646` [54] in `macOS Mojave v. 10.14` (`Linux` could also be used).

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

Listing 8.1: Wiping USB thumb drive.

8.3.5 Experiment - NTFS reformatted with exFAT

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

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

We measured the false positive and false negative rates by comparing the carved metadata results with the filenames we found in `ntfsbase.dd`. A false positive is a hit location not found within metadata describing a file or directory, while a false negative is a set of timestamps not identified as a hit, but which is located within metadata describing a file or directory. Finally, we calculated the precision and recall [158] of the methods implemented.

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

Listing 8.2: Command to find possible NTFS timestamps.

We used a timestamp size of 8 bytes, a search threshold of 24 bytes to search for equivalent timestamps, where at least 3 timestamps are equal. The output from Listing 8.2 was saved to the file cPTS.txt.

[5]We wiped only the partition, shown in Listing 8.1, because `MacOS` gave a resource busy message when trying to wipe the complete raw disk

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

Listing 8.3: Command to identify if the hits are SIA or FNA in a MFT entry, and outputs information from these attributes and from the DATA attribute if possible.

The next step was to assess if each hit was part of a standard information attribute (SIA) or a file name attribute (FNA). Then the script in Listing 8.3 identifies the Data attribute and shows the resident data or the non-resident data runs.

Additionally, we tested if `X-Ways`, `EnCase`, and `EaseUS Data Recovery Wizard` (previously named `Recuva`) were able to recover the previous `NTFS` partition or to find unallocated `MFT` entries using the same forensic image.

8.3.6 Experiment - Previous Ext4 reformatted with NTFS

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

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

Listing 8.4: Command to find possible Ext4 timestamps.

We used a timestamp size of 4 bytes, a search threshold of 12 bytes to search for equal timestamps, where at least 2 timestamps are equal. The output from Listing 8.4 was saved to the file cPTS.txt.

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

Listing 8.5: Command to parse Ext4 inodes.

In Listing 8.5 we start at byte offset 0 in the image, the block size is 4096 bytes, blocks per group are estimated to block size $* 8 = 32768$.

8.3.7 Limitations

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

The output results of the prototype should be assessed by file system experts (or expert systems) in order to assess if a file (name, metadata and content) can be fully or potentially recovered. This is called authentication, and it includes an evaluation of the classifier, the results, and finally a confident decision [38].

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

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

8.4 Results

The `cPTS` command took 13 seconds to run on a 2 GiB byte `dd` image file. The `ntfsParser.py` took less than 1 second, while the `ext4parser.py` took 8 seconds. This is faster than the runtime performance of Dewald and Seufert's [55] tools, however they additionally exported the file content automatically.

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

Table 8.2: Precision and Recall for finding MFT records in ntfsexfat.dd

8.4.1 NTFS metadata carving

For each discovered MFT entry, we know the SIA is associated with the FNA because of the distance between them is less than 1024 bytes, which could visually be verified or falsified by interpreting the byte location for the SIA and FNA timestamp hits. The Data attribute belongs to the FNA because we skipped to the next attribute until we found the unnamed Data attribute, which we found within the next 1024 bytes. This means we have the name, metadata and the content, and since we have the data runs, we know this record is non-resident and potentially recoverable. In order to test if the original content can be connected to the metadata, we need to extract the content and perform hypothesis testing. Extraction of the content based on known data runs is described by Carrier [34].

In Table 8.2 we focus on files/directories and Standard Information Attributes (SIAs). We know each base MFT entry has one SIA. Since we have 79 files (50 files and the 29 system generated files and directories) in our experiment, we know there must be 79 SIAs in the MFT table. We also know there must be 79 SIAs in $LogFile[6] and 4 SIAs in $MFTMirr [34, p. 303]. This gives a total of 162 SIAs. We found all 162 SIAs, and only one of the hits was a false positive. We did not have any false negatives for SIAs. Our computation of precision and recall is shown below.

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

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

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

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

Hits from $LogFile

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

8.4.2 Ext4 metadata carving

We were able to recover all inode metadata entries that were not overwritten by `NTFS`, but reformatting `Ext4` with NTFS in our experiment wiped approximately 20257 inodes from the inode table. When using flex groups, the inode tables are all located continuously in the first block group [55], instead of being divided into its corresponding block group. Our current observations show that our approach supports both types, and does not depend on information from the block group descriptors. For each extent found in an inode, an extraction of the file content is easily performed by extracting from the extent start block as well as the number of blocks contained in this extent. A deleted inode will normally have the extents zeroed out, making the inode connection to the file content infeasible. However, since we find hits for duplicated inodes in different physical locations, we may be able to recover the file content by using extents found in these duplicates. We did not extract the files from the overwritten `Ext4` image due to their quantity, and thus are only discovering potentially recovered files.

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

For the original and reformatted images, we conducted two precision and recall experiments. The first calculates the precision of attributing our recorded or estimated inode number to the true inode number of the inode, and we also calculate the recall of finding our known files with correct attribution. For calculating recall, if at least one inode per inode number was found and we correctly estimated or recorded its inode number, it counted as a true positive, where the duplicate inodes (with respect to its true inode number) were removed. For these precision and recall calculations, we only considered inodes we extracted that contained their true inode number as an attribute. Table 8.3 shows our results for the non-reformatted

	TP	FP	Precision	Recall
Recorded inode matches True inode	77481	0	1	1
Estimated inode matches True inode	27336	50145	0.3528	1
Est or Rec inode Matches True inode	77481	0	1	1

Table 8.3: Precision and Recall for finding and attributing iNode numbers for known files in expExt4Attr.dd

TP	FP	Precision
77675	0	1

Table 8.4: Precision of iNode classification for non-reformatted image.

version of the `dd` file. A similar process was done for the reformatted `dd` file, as seen in Table 8.5, except rather than calculating the recall we simply record the number of discovered inodes per method of inode number estimation. This was done as we cannot be certain how many false negatives were due to our methods or to the file system reformatting (we cannot find something that is overwritten). Note that since at least 20257 of 25050 inodes were wiped from the table, and we recover 5755 inodes, that we are potentially recovering at least 963 previously overwritten inodes.

The second experiment calculates the precision of our method to correctly classify inodes, whether they were from known files or not. False positives in this case are inode hits that contain junk information. Table 8.4 shows the precision from the non-reformatted experiment, and Table 8.6 shows the precision of the reformatted experiment. In both experiments, we obtained 100% precision. We cannot calculate the recall in this case, since we cannot know how many inodes (from the inode tables and copies throughout the disk) exist on the image.

8.4.3 Commercial tools

The results of the tool testing described in this section are shown in Table 8.7.

NTFS reformatted with exFAT

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

	TP	FP	Precision	Files Found
Recorded inode matches True inode	15544	41692	0.2716	4848
Estimated inode matches True inode	7091	50145	0.1239	5755
Est or Rec inode Matches True inode	16553	40683	0.2892	5755

Table 8.5: Precision and Files Found for finding and attributing iNode numbers for known files in Ext4AttrNowNTFS.dd

TP	FP	Precision
57427	0	1

Table 8.6: Precision of inode classification for reformatted image.

X-Ways also has a function that should be able to scan for lost partitions, but this feature was not able to detect the previous partition. We searched for MFT entries, but X-Ways did not find the MFT entries from the previous partition.

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

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

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

We tested the storage device we performed the experiment on with EaseUS Data Recovery Wizard v11.15, and it was able to identify all the 50 files and their content. Since this tool is closed source, we assume they performed a partition recovery by using the VBR backup. For partition recovery they only showed the size, the date created and the path. It also carved for files, but the carved files

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

Table 8.7: Tool testing - Carve for metadata from previous file system when reformatted with another file system.

did not include the metadata. Lastly, we found the tool could not correctly carve the fragmented files or the text files.

Ext4 reformatted with NTFS

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

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

Additional testing

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

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

8.5 Discussion

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

8.5.1 Discussion related to NTFS

For the timestamp hits where we found NTFS MFT entries with a resident data attribute, we can reliably connect the metadata and the file content [38]. This is because the resident data is found within the MFT entry. Normal file carving will not find small ASCII text files that are resident in MFT entries, because these files have no signature within their content.

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

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

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

It is important that the investigator is knowledgeable about the file systems found when using our approach. First of all, our approach uses the data runs found within the NTFS metadata, and the first data run for the MFT entry is relative to the start of the file system, and we need to use the correct cluster size used by the previous NTFS file system. Furthermore, subsequent data runs are relative to the previous run [34, p. 258].

Since we are proposing a generic approach, we cannot automate the extraction of files without considering the specific context, which requires context based recovery tools or manual expert assessment. For instance, the storage media could have first had an NTFS file system, and then been reformatted with NTFS or another file system. Then of course the context is different, and must be taken into consideration.

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

8.5.2 Discussion related to Ext4

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

8.5.3 Addressing Our Statistics and Current Challenges

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

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

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

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

8.6 Conclusion and further work

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

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

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

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

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

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

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

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

8.7 Acknowledgement

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project "Ars Forensica - Computational Forensics for Large-scale Fraud Detection, Crime Investigation & Prevention", grant agreement 248094/O70.

Chapter 9

Paper 4: Timestamp Prefix Carving for Filesystem Metadata Extraction

Porter, K., Nordvik, R., Toolan, F., & Axelsson, S. (2021). Timestamp prefix carving for filesystem metadata extraction. Forensic Science International: Digital Investigation, 38, 301266.

Abstract

While file carving is a popular and effective method for extracting file content from unallocated space in a forensic image, it can be time consuming to carve for the wide variety of possible file signatures. Furthermore, file carving does not connect the discovered file to its filesystem metadata. These limitations of file carving are the advantages of *Generic Metadata Time Carving*, in which filesystem metadata is searched for by first finding repeated co-located timestamps using a potential timestamp carving algorithm. The potential metadata is verified by a filesystem specific parser, and the pointer within the metadata to the file data may allow for full file recovery. Currently, a limitation of the Generic Metadata Time Carving method is that it will only find metadata records that have multiple equivalent timestamps, thus missing metadata records and files with differing, but very similar, timestamps. Therefore, in order to improve the recall of the Generic Metadata Time Carving methodology, we have designed and implemented a prefix matching potential timestamp carving algorithm. We apply our experiments to realistic NTFS and Ext4 forensic images, in which we compare the precision and recall results for differing prefix lengths. Our results indicate that using prefix-based

potential timestamp carving can yield significantly greater recall for extracting filesystem metadata records, with little to no reduction in precision as compared to the original exact potential timestamp carving method.

9.1 Introduction

File carving is an established digital forensics method for extracting files that cannot be found using the filesystem, and while extremely useful it is not without its faults. When applying popular file carving tools such as Scalpel [181], one must attempt to search for all possible file signatures that are relevant to the case, which not only makes the search process more time consuming,[1] but more importantly, the file signature database may be incomplete. Omitted file signatures means missing files when carving.[2] File carving also does not have an automated method for connecting filesystem metadata to the discovered file (assuming the metadata still exists on disk) [55, 149], and has difficulty dealing with fragmentation [74].

These limitations of file carving are the advantages of *Generic Metadata Time Carving* (GMTC) by Nordvik et al. [151]. GMTC is a metadata carving method that uses a simple string matching algorithm, a *potential timestamp carver*, to search for equivalent and closely co-located byte sequences in order to find potential filesystem metadata record timestamps. After the locations of the potential timestamps are found, a filesystem specific parser either accepts or rejects the surrounding content as filesystem metadata. The filesystem metadata record may then be used to retrieve the associated file whether or not the file is fragmented, where the ability to do so is dependent on the filesystem and its policy for deleted files. This technique thus connects filesystem metadata to the file data, enabling at least *metadata and content recovery* [38]. Furthermore, the method does not depend on file signatures, since timestamps are essentially a dynamic signature for all filetypes.

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

Generic Metadata Time Carving has several limitations as well. The largest of

[1]While Scalpel uses a modified version of the single-pattern string matching algorithm Boyer-Moore [29] (as of 2018 [17]) and causes header-footer matching to run in $O(sn)$ time where s is the number of header-footer signatures and n is the length of the data being processed, Zha and Sahni [212] created *FastScalpel* which uses the multi-pattern string matching algorithm Aho-Corasick [5] that performs header-footer matching in linear time.

[2]In the case of general data carving methods (for example, the Bulk Extractor [76]), omitted regular expressions may mean missed data such as credit card numbers, social security numbers, etc.

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

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

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

We perform a location-based data recovery evaluation to test the performance of our tools' abilities to carve for filesystem metadata records, wherein we measure their precision and recall for extracting records from specific files or regions of the disk (such as from the $MFT, $LogFile, and the inode table). Note that we are running our tools on the entire disk images, as the tools are intended to be used, but we only calculate precision and recall for identifying filesystem metadata record hits for specific regions of disk, on a particular partition. We would have

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

liked to obtain the precision and recall of our tools' ability to carve for filesystem metadata records across an entire disk image or partition, but since we did not create the test images we have no way of knowing the ground truth information regarding the locations of all filesystem metadata records on any partition. We additionally determine the files containing any hits for file system metadata records found outside these test spaces.

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

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

9.2 Related Work

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

In Figure 9.1, we show an abstraction of a disk image with two partitions, which also shows a simplified abstraction of filesystems. The image is a representation of how critical filesystem data such as the MFT table or superblocks keeps track of the filesystem metadata records. For details on traditional file carving, see the Forensics Wiki [68].

9.2.1 Metadata Carving

One of the first works that scientifically studied metadata carving was done by Dewald and Seufert [55]. They exclusively carve for inodes in Ext4, where they intentionally made their images' superblocks and group descriptor tables unus-

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

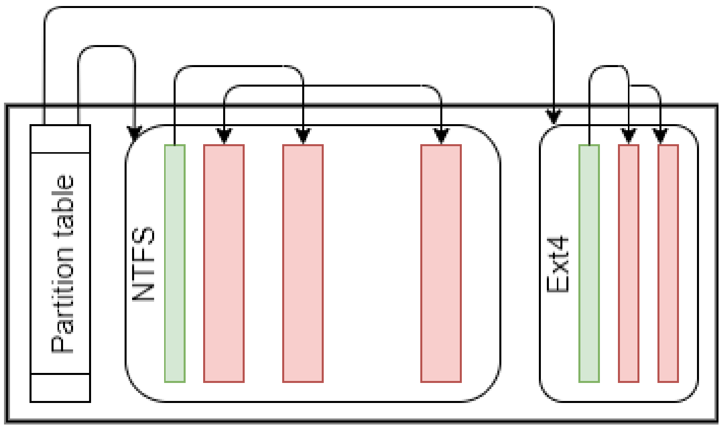


Figure 9.1: An abstraction of a simple disk image, partitions, and filesystems. The large encompassing rectangle is the entire disk image, the furthest left rectangle with internal lines is the partition table that points to the partitions, and the other rectangles with rounded corners are partitions. Each partition has a filesystem, where the green rectangles represent filesystem critical data structures such as the $MFT record (and its mirror), superblock, or group descriptor table. These help keep track of the filesystem records (for example, inodes or MFT records), which are represented by the red rectangles. *Generic Metadata Time Carving* [151], and our work, attempts to find the red blocks without help from the green blocks. For a more complete picture of the general filesystem structure, see the work by Carrier [34].

able. Their method of byte-wise search uses search patterns that are expected to be found in inodes such as file flags, extent header magic numbers, and tests the relationships between the timestamps ensuring their validity. In their experiments, they tested a variety of combinations of the inode attributes to search for, where the effectiveness of the combination of patterns is dependent on the case. Similar work was performed by Plum and Dewald [163], where they carved for nodes using different combinations of object type and subtype.

Our work extends the work by Nordvik et al. [151], wherein they created the Generic Metadata Time Carving method. The method considers every non-overlapping m length sequence of bytes as a search keyword, where this keyword is checked for equivalency against every non-overlapping m length sequence of bytes in a k length search window directly following the keyword. If the keyword matches one or more of the sequences of bytes in the search window, then the location of the potential timestamp is recorded. One notable aspect of their byte-wise search approach to timestamps is that it does not require the user to set a minimum or maximum date they are searching for, only a guess as to what the length of the timestamp is and the filesystem that is suspected. For instance, it is required that the length m of the timestamp must be defined as either 4 or 8. After a list of potential timestamp locations are compiled, another scan over the disk image is performed that is filesystem specific. Using the list of potential timestamp locations, they directly access each location on the image and check if specific byte offsets relative to the timestamp location fit the profile of the metadata record being searched for. Examples of such expected features are Standard Information Attribute (SIA) or Filename Attribute (FNA) flags for NTFS, or the extent header magic number for Ext4. Once the metadata record is identified as a positive hit, the metadata information can easily be read out, including resident files for NTFS records, dataruns from NTFS Data Attributes, and extents and block pointers from Ext4.

The potential timestamp carver by Nordvik et al. [151] works generally, but to date, their filesystem specific parsers only support NTFS and Ext4. For their experiments, they created disk images with a known set of files for each filesystem, where each filesystem was then damaged by reformatting the image with a different filesystem. Their results for the NTFS experiment achieved greater than 99% precision in identifying MFT records and full recall in retrieving files known to the original filesystem. For Ext4 they obtained 100% precision in identifying inodes, but only retrieved about 23% of the inodes known to the original filesystem. Their experiments however only included metadata records that had multiple timestamps that were exactly the same, thus our work intends to apply a prefix matching version of their approach to more realistic datasets.

The first notable reference to byte-wise timestamp carving appears to be from McCash [128], wherein he used an EnScript from Mueller [135] to discover MFT records, indexes, and registry keys. The timestamp carving methodology essentially allows the user to input a date or range of possible dates, the EnScript converts the dates into their NTFS byte format, and the possible byte sequences are then searched for. To improve the precision of the tool, there is an option to search for contiguous potential timestamps.

9.2.2 Related Methods of Data Retrieval

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

One tool that focuses on Ext4 file recovery via non-traditional means is Ext4Magic [126]. The basics of the tool is that it uses journal blocks with an old but functional deleted inode. The inode will hopefully point to datablocks which have not been reused for a different file.

Bulk Extractor by Garfinkel [76] gathers relevant forensic features such as email addresses, phone numbers, credit card numbers, and more by parsing through a disk image in a single scan block by block. A benefit of the tool is that it truly is filesystem agnostic, and can analyze different parts of the disk in parallel.

9.3 Methodology

In this section we first describe our prefix-based potential timestamp carving algorithm and how it fits into Nordvik et al.'s [151] Generic Metadata Time Carving workflow, and then describe our experimental and evaluative methodologies.

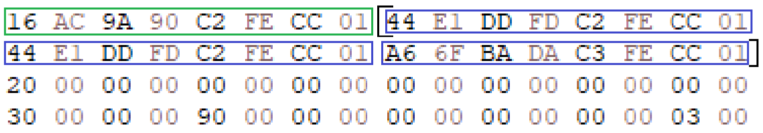
9.3.1 Prefix-Based Potential Timestamp Carving Algorithm

The prefix-based potential timestamp carving algorithm, like the exact matching version by Nordvik et al. [151], is used to identify the byte offset locations of potential filesystem metadata record timestamps from across an entire disk image. The algorithm outputs the offsets from the beginning of the image to a text file. Our prefix-based potential timestamp carving algorithm adds unique elements to the timestamp carving algorithm and source code from Nordvik et al. [151]. We briefly review the original algorithm, as understanding their work is imperative for understanding our own contributions. Their basic assumption is that timestamps within filesystem metadata records are typically co-located close to each other, and often two or more timestamps are identical.[5]

The search procedure for these algorithms is based on the *sliding window* approach, where we have some byte-stream T representing the forensic image being searched, and we let m be the length of a timestamp. Potentially, almost every non-overlapping m bytes of the forensic image is tested as a *candidate timestamp*, and used as a *keyword*. This test requires a user defined value k, which is the length of bytes for a search window following each candidate timestamp. If the candidate timestamp passes the test, then it is considered to be a *potential timestamp*. The search begins at $T[0]$, with the first candidate timestamp being $T[0 : m - 1]$. This candidate timestamp is then compared to each non-overlapping m byte sequence within the k length window for equivalency. We refer to this process as the *timestamp equivalency test*, and these byte sequences as *test sequences*. If the number of matches is greater than or equal to the threshold $h - 1$, h being the number of required matching timestamps within a metadata record set by the user, then the position of the candidate timestamp (now potential timestamp) is recorded, the search skips ahead by k, and repeats the search procedure. The definitions given in this paragraph and the timestamp equivalency test are illustrated in Figure 9.2. If

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

the candidate timestamp is not found to be a potential timestamp, then the search only skips ahead by m, and the search procedure repeats. The search continues until the last k bytes of T. Full details of the algorithm can be found in Nordvik et al. [151].



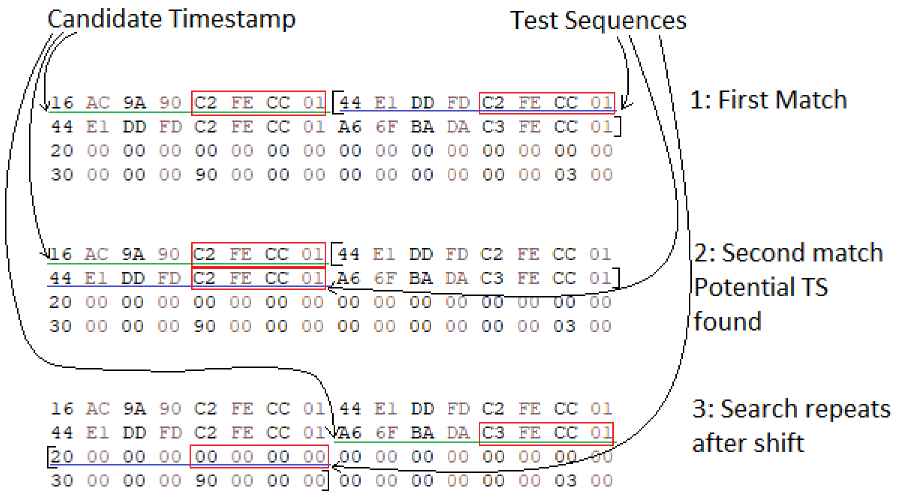
```
16 AC 9A 90 C2 FE CC 01 [44 E1 DD FD C2 FE CC 01
44 E1 DD FD C2 FE CC 01 A6 6F BA DA C3 FE CC 01]
20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
30 00 00 00 90 00 00 00 00 00 00 00 00 00 03 00
```

Figure 9.2: For 8 byte timestamps, the *candidate timestamp* is highlighted with green, and the *test sequences* are highlighted in blue. The search window is indicated by the brackets. The *timestamp equivalency test* simply checks how many times the candidate timestamp matches the test sequences. If the number of matches is greater than or equal to the threshold $h - 1$, where h is the number of required matching timestamps within a metadata record set by the user, the candidate timestamp is deemed a *potential timestamp*.

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

An example of such a search is shown in Figure 8.12. The prefix of the timestamp (the most significant bytes) is least likely to change when a timestamp is updated, and typically holds information regarding the timestamp's month and year. We argue this form of prefix matching is more suitable as an approximation metric than other popular metrics such as the Hamming [93] or edit distance [115], since they do not consider the order in which the matching errors occurred. Furthermore, the method for prefix matching is algorithmically simple.

For testing if a candidate timestamp is a potential timestamp, the original algorithm by Nordvik et al. [151] converted the candidate timestamp byte sequence and the test sequences to their big-endian forms for using them as `unsigned long longs`, and we do this as well. For most operating systems and filesystems, timestamps are recorded in little-endian, but utilizing them in a program in big-endian is generally more useful. In order to test whether the candidate timestamp and the test sequences have an equivalent prefix of length p (that the p most significant bytes of the timestamps are the same) we need only XOR the timestamps in their big-endian form, and shift the resulting value to the right by $8 * (m - p)$ bits. The shift to the right removes the $m - p$ least significant bytes from the result of the XOR, and if the remaining value is 0, then the prefixes must match. If the prefixes match, the count of matching timestamps for the candidate timestamp is


Candidate Timestamp
Test Sequences
1: First Match
2: Second match Potential TS found
3: Search repeats after shift

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

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

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

Figure 9.3: Hex dump with highlights to illustrate the timestamp prefix matching search procedure. The byte sequence underlined in green represents the current candidate timestamp, and those underlined with blue are test sequences. The brackets represent the candidate timestamp's search window. The red boxes represent the little-endian prefixes that are being compared for equivalency. The first two examples show matches, despite the fact the candidate timestamp does not equal the subsequent ones. If three matching timestamps are required ($h = 3$), the third example shows the advancement of the search by k bytes, and begins to repeat the entire procedure.

increased by 1. Algorithm 7 explicitly describes this process. The prefix matching algorithm is only a component within the prefix-based potential timestamp carving algorithm (see Algorithm 13), which is shown in Appendix B.

We add one extra condition that must be met when applying prefix-based timestamp carving. The original method requires that candidate timestamps cannot be sequences of repeated bytes, as this filters out very common byte sequences such as 0x0000 and 0xFFFF. These common byte sequences would otherwise generate many false positive timestamp matches. We keep this condition, but add that a candidate timestamp's most significant bytes cannot be 0. This is due to the fact that there are many byte sequences which in big-endian start with non-zero values, but end with zeros, which will cause our algorithm to identify many potential timestamps where the candidate timestamp is a non-zero byte sequence, but the prefixes being tested are not. We consider this to be a fair assumption to make, as most timestamps' most significant bytes are non-zero. Both conditions can be found in Algorithm 13 within B.

Since the modification of the potential timestamp carving algorithm by Nordvik

Algorithm 7: Prefix matching algorithm.

```
Input: Big-endian unsigned long long forms of candidate
       timestamp $x$ and test sequence $y$
Output: Number of matches for the candidate timestamp either increases
        or stays the same
$m$ # Length of timestamp;
$p$ # Size of prefix;
$xorResult = x \bigoplus y$;
if $((xorResult >> (8*(m-p))) == 0)$ then
    $matchCount$ += 1;
end
```

et al. [151] primarily only changes the timestamp equivalency test by a constant number of steps, it implies that the time complexity of the prefix-based potential timestamp carving algorithm must remain the same. While Nordvik et al. showed that a general potential timestamp carver would run in nonlinear time (their worst case scenario omits the repeated byte timestamp check and conversion from a string of characters to a 64-bit data type), the implementation of the potential timestamp algorithm effectively runs in linear time, dependent on the length $|T|$ of the disk image. This is because length m is limited to a choice of 4 and 8 (timestamps are stored in an unsigned long long type), and because the search window length k in most practical situations would never exceed the size of a block or cluster.

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

9.3.2 Generic Metadata Time Carving and the Filesystem Specific Parsers

The potential timestamp carving algorithm described in the previous section is only the first step of the Generic Metadata Timestamp Carving (GMTC) methodology. The produced list of potential timestamp locations in general will be extremely large, and thus referring to many "false positive timestamps" identified across the full disk image. In the GMTC method, the list of potential timestamp locations is then fed into a filesystem specific parser, and the parser checks the offsets on the disk image given by the potential timestamp list in an attempt to verify if the potential timestamp is contained within a filesystem metadata record. The output of the parser is a .csv and .txt file containing data from the suspected records. More or less, a filesystem specific parser acts as a filter. We briefly describe the strict verification tests performed by the NTFS and Ext4 parsers. The majority of stated parser tests were in the original work by Nordvik et al. [151]

The NTFS parser first checks if the date of a potential timestamp falls within the year range of 1970 and 2100. If so, it then checks all possible offsets behind the potential timestamp location for Standard Information Attribute (SIA) or Filename Attribute (FNA) attribute header flags. If one of these flags are found, we make an assumption of where the attribute begins. We can use the identified attribute lengths to navigate from one attribute to the next, where we require to start from the SIA, then hop to an FNA, and then if possible the Data attribute. Furthermore, the MFT attributes encountered must be in numerical order (as given by their header flags). Any attributes encountered for an assumed record must occur within a 1024 byte space. The parser also only reports an MFT record if the identified filetype extracted from an FNA is one of four possible types. The accepted filetypes are: "File", "Directory", "Index View", or "Directory and Index View". More information regarding MFT data structures can be found in the work by Carrier [34].

The Ext4 parser is more complicated due to the fact that inodes are small, have far less features to strictly identify than MFT records, and that the parser attempts to connect the inodes to a filename and inode number. Both the inode number and filename are in a different data structure than the inode. The first step for validating potential timestamps is to check the possible offsets from the potential timestamp to the filetype nibble at the start of the inode. We only allow for three different types: "Regular Files", "Directories", and "Symbolic Links". The offset to one of these values dictates our guess to where the inode begins. If the extent flag is set, and the offset to the extent header magic number is 0xF30A, or if there is no extent flag and the offsets from the beginning of the inode 0x24 to 0x27 are 0, we continue our validation tests. The total size of the file is checked to see if it corresponds to the total amount of blocks it is occupying, if the total size of the file is less than the size of the image, and if the relationships between the timestamps are valid. For instance, we check if the deleted value is not 0, then it must be greater or equal to both the modified and created time. The steps so far are the validation checks done in the preprocessing phase, as we need to gather information to try to connect inodes to their filename and inode numbers. If the inode passes the initial preprocessing and validity checks, it is fully processed. The timestamps are checked to ensure they fall within the years 2000 and 2020 (the deletion timestamp being the exception). For more information about Ext4 data structures and inodes, see [62].

9.3.3 Experimental Methodology

We use our prefix-based Generic Metadata Time Carving (GMTC) method on three realistic forensic images. We first apply our novel prefix-based potential timestamp carving algorithm on the images, where the output of the algorithm creates a text file list of the locations of the potential timestamps in byte offsets from the beginning of the image. This list is then input into one of the two pre-existing but modified filesystem specific parsers (NTFS or Ext4), where the output of the parser is a .csv and .txt file with data from the discovered metadata records. We identified a few bugs in the original filesystem specific parser scripts by Nordvik et al. [151], so we updated them so that we may achieve more complete and accurate results.

The three images being tested are a 1 GB NTFS image, one 59.5 GB Ext4 image from a real device, and one 476 GB synthetic NTFS image. The small NTFS image is from NIST's Deleted File Recovery page (DFR-13) [145], the Ext4 image was extracted by the authors from a real Samsung S8 mobile phone, and the large NTFS image is the "Lone Wolf" forensic image, available from Digital Corpora [134]. Notably, these images are not guaranteed to have at least two equivalent timestamps per metadata record, unlike the work by Nordvik et al. [151].

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

We clarify some items regarding our testing and evaluation methods. The prefix-based GMTC methodology (the potential timestamp carving followed by the filesystem specific parser) is applied to the entire disk image for our tests. Thus we obtain potential timestamp locations and filesystem metadata records from across entire disk images. However, since we have no ground truth information regarding the number and location of all file system metadata records across the disks, we cannot evaluate the precision and recall of our tools across an entire disk. Thus, we limit our precision and recall evaluations to specific areas or files on the disk, where we can easily retrieve the number and location of records. Examples of such files or regions of disk include the MFT table or inode table for a particular partition. This is explained more in depth in the next subsection.

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

9.3.4 Precision-Recall Location-Based Data Recovery Evaluation

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

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

We refer to these ground truth offsets to MFT records as *Condition Positives*. Formally, a Condition Positive is the knowledge that at address A, there exists a filesystem metadata record. It is *conditioned* on the fact that we are limiting our precision-recall evaluations to the regions of the disk occupied by a specific file or ranges of disk space. By running our prefix-based GMTC method on the entire disk image, we obtain a large set of byte offset locations that our tools detect as the locations of filesystem metadata records.[6] We refer to the offsets identified by our tools as *Test Positives*. One can think of our filesystem specific parsers similar to

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

that of a classifier when they filter potential timestamp locations, which declares at address A we detect a filesystem metadata record of minimum length L. The value L for inodes and MFT records is 256 bytes (the minimum length of an MFT record includes the record header of length 0x38, Standard Information Attribute of length 0x60, and a minimum length Filename Attribute of 0x68). A Test Negative is simply that our tools do not detect a filesystem metadata record at address A.

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

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

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

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

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

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

9.3.5 Specifics of NTFS Experiments

The 1 GB NTFS image is the 13th test case (dfr-13-ntfs.dd) from NIST's Deleted File Recovery page [145]. This test case has performed random filesystem activity, so the timestamps of the MFT entries are rarely all equal. When running our prefix-based timestamp carving algorithm we set the length of the timestamps $m = 8$, the search window $k = 24$, and the required number of matching timestamps to $h = 3$ (the same parameters used by Nordvik et al. [151]). We carved for timestamps for all possible prefixes p, from 1 to 8.

For this image, we only performed the location-based data recovery evaluation on the $MFT of the partition starting at sector 128, as we did not identify any MFT records in the $LogFile. We verified the lack of full MFT records in the $LogFile by running the LogFileParser by Schicht[7] on the file. Transactions in the LogFile where the Redo Operation or Undo Operation has the status of "InitializeFileRecordSegment", and the other Redo or Undo Operation has the status of "Noop" indicates that the transaction contains an entire MFT record [48]. We found no such transactions.

The experiment using the 476 GB Lone Wolf forensic image (available from Digital Corpora [134]) focuses on the "Basic Data Partition" for the precision and recall evaluations, the largest partition on disk. Our timestamp carving experiments for the Lone Wolf image use the same parameters as the DFR-13 image.

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

[7]https://github.com/jschicht/LogFileParser

9.3.6 Specifics of Ext4 Experiments

The Ext4 experiment uses a dump of a Samsung S8 mobile phone running Android, where we specifically focus on the "SYSTEM" partition's inode table for the precision and recall calculations. The User partition was encrypted, and SYSTEM partition was the second largest partition on the image. The image was created by first flashing the recovery partition using the TWRP Recovery image [199], and then using an ADB bridge executing a combination of netcat and dd commands in order to acquire the raw image. The recovery image method is described in detail by Son et al. [195] and Vidas et al. [204]. We ran our prefix-based timestamp carving algorithm on the image with the same parameters as those used by Nordvik et al. [151], where the length of the timestamps m was set to 4, the search window $k = 12$, and the required number of matching timestamps to $h = 2$. We carved for timestamps for all possible prefixes p, from 1 to 4.

The Ext4 parser also requires a few additional parameters, which are assumptions that assist in attempting to connect inodes to their filename and inode number. Using the Sleuthkit, we obtained the blocksize of 4096 bytes (which is the default blocksize [62]), and the byte offset of 225968128 to the partition. Thus, the parser only examines the disk from this offset onward.

9.3.7 Computer Specifications

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

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

While running the tools we did not activate any other resource demanding processes. However, it is always possible that the OS performed additional scheduled tasks. We used the tool DiskMark[8] v2.2 to measure the read/write speed.

9.4 Results

Overall, our results show that by reducing the size of the prefix p of a timestamp in the timestamp equivalency test, a much higher recall for filesystem metadata record

[8]https://inchwest.com/diskmark/

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

9.4.1 Small NTFS Image

The results for carving MFT records from the $MFT from the 1 GB NTFS image's partition beginning at sector 128, as seen in Table 9.1, show that applying prefix matching of timestamps greatly increases the recall, and appears to maintain the exact matching Generic Metadata Time Carving method's 100% precision. The exact matching GMTC ($p = 8$) only obtained 8.8% recall for finding MFT records, whereas decreasing p to 3 and less achieved a 97.9% recall. The number of Test Positives identified over the entire partition for different values of p is shown in Table 9.2. The true positives account for most of the Test Positives found over the entire partition, but three had gone unaccounted for. It transpired they were the $MFTMirr, $LogFile, and $Volume records found in the $MFTMirr file.

This increase in recall was not without its trade-offs, as seen in Table 9.3. Upon decreasing p from 8 to 4, the number of identified potential timestamp locations increased by three magnitudes. While this did not appear to unduly influence the timestamp carving algorithm, the time required for the filesystem parser increased more than 100 fold.

9.4.2 Ext4 Samsung S8 Image

The results for carving inodes from the Ext4 image's SYSTEM partition's inode table are shown in Table 9.4. Like the small NTFS image, we achieved 100% precision in identifying inodes, where the recall increased for carving inodes from the inode table as the prefix length value of p decreased. However, the increase in recall was quite minor, only increasing by about 3%. We discuss our theories of why the precision and recall were so high for the Ext4 experiment in the Discussion section.

Table 9.5 shows the Test Positive counts of detected inodes found over the entire partition, with respect to the prefix length p being used. All test positive hits that were not discovered in the inode table were discovered in the Journal where, when

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

Table 9.1: Precision and recall for carving MFT records from the $MFT from the 1 GB NTFS image's partition beginning at sector 128 with $p = 1, 2, \ldots, 8$. The $MFT had 239 Condition Positives.

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

Table 9.2: Test Positive count over entire partition from the 1 GB NTFS image, where $p = 1, 2, \ldots, 8$.

the timestamp prefix length $p = 1$, we detected 1924 inodes.

In terms of computational performance, Table 9.6 exposed trends regarding the timestamp carving program when working with large files. Larger prefixes p caused the timestamp carver to take longer to complete, but not by too much. Unlike the small NTFS image experiment, the number of potential timestamp locations only increased by about one magnitude going from $p = 4$ to $p = 1$. The time required to run the filesystem parser appears to have an approximately linear relationship between the number of potential timestamp carving locations, as the time required to run at $p = 1$ is about 10 times as slow as using a prefix size of $p = 4$.

9.4.3 Large NTFS Image

The results for carving MFT records from the $MFT and $LogFile from the Lone-Wolf image's Basic Data Partition are seen in Table 9.7 and Table 9.8 respectively. Again, we achieved 100% precision in identifying MFT records, both for the $MFT and $LogFile (we encountered no false positives with respect to our Condition Positive lists). The recall results reflect previous trends. Using exact matching timestamp carving we only achieved 41.6% recall for carving MFT records from the $MFT, and allowing for smaller timestamp prefix matching caused increasingly higher recall. The point of diminishing returns appeared to have oc-

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

Table 9.3: Generic Metadata Time Carving performance for the entire 1 GB NTFS image with $p = 1, 2, \ldots, 8$. PTS stands for "Potential Timestamp".

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

Table 9.4: Precision and recall for carving inodes from the inode table from the SYSTEM partition in the 59.5 GB Ext4 Samsung S8 image with $p = 1, 2, 3, 4$. The inode table had 7436 Condition Positives.

curred at $p = 2$, where about 97.2% recall was achieved. The recall results are quite different for carving MFT records from the $LogFile, as the recall hovered around 87% despite the value of p.

Table 9.9 shows the total number of test positives found over the entire partition, and after filtering out the Test Positive hits found in the $MFT and $LogFile, there were still a large number of hits left unaccounted for. When discussing where these hits were found on the partition, we focus on the results for $p = 1$, since each value of p larger than this should be a subset of the $p = 1$ results. In total, there were 91157 test positive hits that were yet to be accounted for. Using the dictionary we created that contained the allocated cluster ranges of all known files on the partition, we were able to discover where these potential MFT records were coming from, as seen in Table 9.10. The $MFTMirr contained the usual records of $MFT, $MFTMirr, $LogFile, $Volume. Four different boot.sdi files (with the filenames "boot.sdi, boot.sdi") each contained 42 filesystem metadata record hits, where a boot.sdi file is essentially a small partition of its own with completely irrelevant MFT records. It is used as a Ramdisk which can be shown with the *bcdedit* com-

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

Table 9.5: Test Positive count over entire SYSTEM partition from the Ext4 Samsung S8 image, where $p = 1, 2, 3, 4$.

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

Table 9.6: Generic Metadata Time Carving performance for the entire 59.5 GB Samsung S8 image with $p = 1, 2, 3, 4$. PTS stands for "Potential Timestamp", m for minutes, and s for seconds. Note, the Ext4 parser skips the first approximately 210 MB.

mand [4]. Lastly, we have the two Volume Shadow Copies[9] that contained 90985 detected MFT records.

In terms of timing performance, the large NTFS experiment mostly behaved as expected (see Table 9.11). Like in the Ext4 timestamp carving experiment, the run-times for all values of p were similar, but experiments with lower values of p took less time. What was rather surprising was the relatively small increase in potential timestamp locations that were found by $p = 1$ versus $p = 8$, given the size of the image. The increase was only by a factor of about 4.77, quite a deal less than the increase of magnitudes we saw before. A possible reason for this is that the Lone Wolf image is a synthetic image that was only being used for some months. Stranger still, were the parser times over all possible values of p. Given the previous results, we should have seen parser times drastically increase as p decreased. This did not happen, as seen by the fact that the parsing time for $p = 1$ was on average less than most other values of p, and this is despite the fact that the experiment for $p = 1$ had about 46 million more potential timestamps to check than the $p = 8$ experiment. Since both runs of the parser produced such similar

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

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

Table 9.7: Precision and recall for carving MFT records from the $MFT of the Basic Data Partition from the 476 GB LoneWolf NTFS image with $p = 1, 2, \dots, 8$. The $MFT had 142960 Condition Positives.

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

Table 9.8: Precision and recall for carving MFT records from the $LogFile of the Basic Data Partition from the 476 GB LoneWolf NTFS image with $p = 1, 2, \dots, 8$. The $LogFile had 2604 Condition Positives.

results, at the moment we can only guess that some aspect of the parser script handles things inefficiently. A major difference between the NTFS parser and the Ext4 parser is that the Ext4 parser uses a Python memory mapping library[10] to handle the parsing of large files, while the NTFS parser has handcrafted code to handle large files.

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

[10]https://docs.python.org/3/library/mmap.html

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

Table 9.9: Test Positive count over entire Basic Data Partition from the large LoneWolf NTFS image, where $p = 1, 2, \ldots, 8$.

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

Table 9.10: Files containing the remaining Test Positives not found in the $MFT or $LogFile of the Basic Data Partition, where $p = 1$. The number associated to each file indicates how many MFT records were found in that particular file.

9.5 Discussion

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

Analysis: Small NTFS Image

The small image from NIST [145] purposefully created chaotic actions on the system, thus creating MFT records with erratic timestamps. The missed MFT records from the MFT table when $p = 1$ were the $MFT, as the Standard Information Attribute timestamps were 0, and 4 other records that did not contain File Name Attributes. The NTFS parser requires a File Name Attribute to be present.

The only other interesting item to note is explaining why the number of potential timestamp locations jumped drastically from $p = 5$ to $p = 4$. The *dfr-13-ntfs.dd* image fills sectors not occupied by an MFT entry with repeated byte sequences of either 0x2A or 0x5A, and the beginning of each sector has a message describing how the sector is or is not used. The combination of this message and the repeated byte sequences creates a large occurrence of valid potential timestamps. Such a situation would be unusual for more realistic images.

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

Table 9.11: Generic Metadata Time Carving performance for the entire 476 GB NTFS image with $p = 1, 2, \ldots, 8$. PTS stands for "Potential Timestamp", hr for hours, m for minutes, and s for seconds.

9.5.1 Analysis: Ext4 Samsung S8 Image

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

The 432 false negative inodes from the inode table were entirely comprised of Symbolic Links. While the GMTC method by Nordvik et. al. [151] and our work is said to consider symbolic links (and will catch some), the Ext4 parser always assumes that at offset 0x28 from the start of the inode will be direct blocks or the start of the extents. However, this is an incorrect assumption, as a symbolic link will be stored at this offset if the string is less than 60 bytes long [62].[11]

9.5.2 Analysis: Large NTFS Image

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

The high recall of at least 86.4% of MFT records found in the $LogFile can be attributed to the fact that full MFT records only occur in $LogFile transactions with "InitializeFileRecordSegment" operations, which means a new file is being created [188]. When a new file is created, all the timestamps for the Standard Information

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

Attribute (SIA) are updated [108]. This implies all the timestamps for the MFT records should be the same or nearly the same. Decreasing the timestamp prefix length p from 8 to 1 increased recall by less than 1%, which shows that some of the timestamps were indeed slightly different. The implications of these results is that the exact matching GMTC method will work well for carving MFT records from the $LogFile. However, as Nordvik et al. [151] previously observed, the majority of records recovered from the $LogFile contained no datarun information, where we only identified 11 records that did.

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

Of the 91157 Test Positive hits for MFT records that were found in neither the $MFT or the $LogFile (see Table 9.10), the 90985 hits in Volume Shadow Copies are the most interesting. This is because the Volume Shadow Copies are snapshots of previous states of the partition, and thus may either contain previous states of files and their MFT records, or they may contain MFT records that are now deleted. Furthermore, a large number of Test Positive hits outside the MFT table or LogFile, but within specific regions of disk, may indicate that Volume Shadow Copies even exist on a partition in the first place.

9.5.3 Limitations

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

The first issue is that we are applying the GMTC method to data that does not fit its more typical use-case. The GMTC method should be applied if the filesystem is damaged or otherwise inaccessible. We are applying the method to perfectly working images, and undamaged filesystems. The reason for this is to understand what the prefix-based GMTC method can *potentially* recover.

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

The last large limitation of our work is our experiments' unsatisfying explanation of the unflagging 100% precision. However, by looking at our results, we can see that low timestamp prefix lengths do indeed produce many more false positive potential timestamps. For example, reducing the prefix length p from 4 to 1 in the Ext4 experiment increased the number of potential timestamps from approximately 11 million to 98 million. For the Large NTFS experiment, reducing p from 8 to 1 increased the number of potential timestamps from about 12 million to 58 million. The effect of prefix length p on the number of potential timestamps identified for the large NTFS and Ext4 images is shown in Figures 9.4 and 9.5 respectively, where we also show the number of Condition Positives to illustrate that the count of Condition Positives is only a fraction of the the number of false positive timestamps we may be encountering. According to our filesystem specific parsing experiments, it would seem that the filesystem specific parsers are extremely strict since we encountered no false positive records despite checking for millions of more offsets on the disk image. Likewise, it appears that the roll of the potential timestamp carver is to control the total number of byte offsets that a filesystem specific parser must verify when looking through a disk image for filesystem metadata records (affecting recall), and that the filesystem specific parser ultimately controls the precision of the GMTC method.

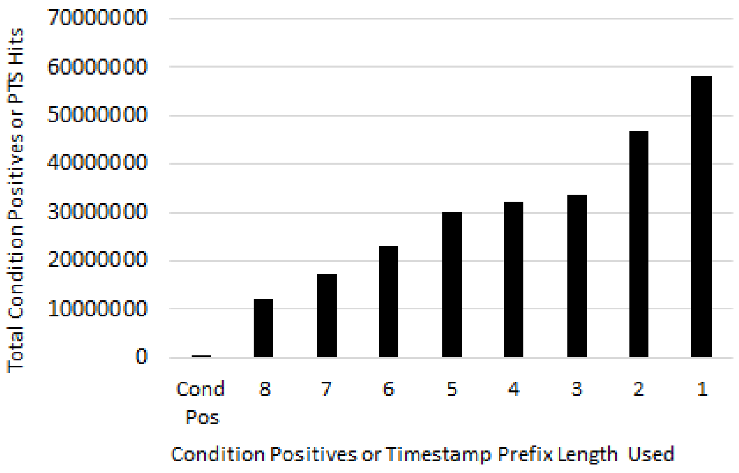


Figure 9.4: Histogram comparing the number of Condition Positives we account for on the Basic Data Partition of the 476 GB Lone Wolf image and the number of potential timestamp (PTS) locations identified after carving for all possible prefix lengths.

We wanted to observe these suspected rolls of the prefix-based potential timestamp

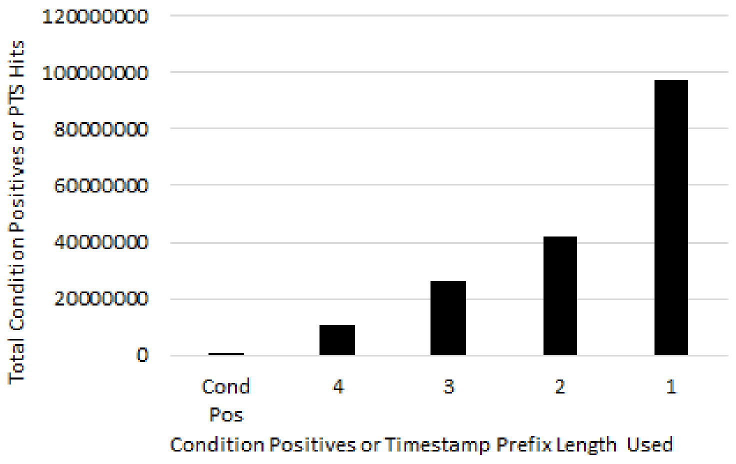


Figure 9.5: Histogram comparing the number of Condition Positives we account for on the SYSTEM partition of the 59.5 GB Samsung S8 image and the number of potential timestamp (PTS) locations identified after carving for all possible prefix lengths.

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

We encrypted the 59.5 GB Ext4 image with Kleopatra[12] and carved for potential timestamps on the encrypted image using the same parameters as our previous experiments, but only searched for timestamps based on a prefix size of 1 byte. Then both filesystem specific parsers were ran on the encrypted image using their respective potential timestamp locations from the potential timestamp carving. Next, we ran the NTFS and Ext4 parsers over every byte of the encrypted image without using potential timestamp information, with the exception of the first and last 1024 bytes.

For searching for NTFS MFT records, we obtained no false positives for the

[12]https://www.openpgp.org/software/kleopatra/

GMTC experiment or the pure parser experiment. Carving for potential timestamps took approximately 17 minutes, and where 360990 potential timestamps were discovered. Applying the NTFS specific parser on the image with the potential timestamp results took about 7 minutes to run. Applying the NTFS parser directly on the encrypted image took 5.75 hours.

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

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

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

9.5.4 Revisiting Research Questions

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

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

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

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

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

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

How does the original Generic Metadata Time Carving method compare with our prefix matching implementation?

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

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

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

In terms of time and space complexity, the time complexity of the prefix matching algorithm is the same as the exact matching algorithm. Our results show that the timestamp carving times are close to constant for all values of p, but carving with lower values of p will take slightly less time. A limitation of our work is that we did not perform extensive tests on the original GMTC algorithm, thus making statements on the speed of our algorithm versus the original mostly theoretical. Where the prefix-based GMTC method performs worse than the original method, is the space required for the potential timestamp locations produced by the potential timestamp carver, and consequentially the time required by the filesystem specific parsers. For example, the exact timestamp carving on the Ext4 image identified nearly 11 million potential timestamps and the parser took about 46 seconds to run, but carving for timestamps with a prefix length of 1 byte on the same image identified nearly 98 million potential timestamps and the parser took about 7.5 minutes to run. As there were only 7436 inodes in the Ext4 partition's inode table, the grand majority of the potential timestamps are false positive timestamps.

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

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

In terms of functionality, the prefix-based GMTC method appears practical for carving filesystem metadata records as our experiments acheived recall of approximately 90% or greater. The exact matching GMTC can be practical if time is of primary concern, or one wishes to carve for records in specific files such as $LogFile (where the existence of dataruns is rare), but for many files or regions of disk this will risk missing many filesystem metadata records.

As filesystem metadata records are highly structured (and often sparse) data, and our filesystem specific parsers run many verification tests, we can understand why our parsers filtered out all of the tested false positive potential timestamps. Further investigations in our Limitations subsection showed that even when running our tools on the encrypted Ext4 image, that we encountered no false positives. The implication is that while potential timestamp carving will allow for greater recall, what ultimately controls the precision are the filesystem specific parsers, and that the precision measured in all cases was 100%.

However, we also showed in the Limitations subsection that by running the parser without potential timestamp information on the disk images took a significantly longer time than the GMTC method. For example, the prefix-based GMTC method took about 35 minutes to fully run on the encrypted Ext4 image when searching for inodes, whereas running the Ext4 parser alone took about 17.5 hours. Applying the prefix-based GMTC method to the encrypted image took 24 minutes to search for MFT records, while running the NTFS parser directly on the image took 5.75 hours.

In terms of time and space both the exact matching and prefix matching GMTC methods are practical. The time taken to carve out potential timestamps on the 476 GB NTFS image was on average just over two hours. Furthermore, the carving time is not much affected by the change in the prefix length p, as was predicted by the fact that the time complexity of the prefix-based timestamp carving method is the same as the exact timestamp carving method. In general, it appears that as we allow for smaller prefixes in timestamp carving, the time it takes for the filesystem specific parsers to complete increases. The exception to this rule is the NTFS parser for large files, but we believe this to be more of an implementation issue than indicative of a general trend. Even so, the longest time it took for the NTFS parser to scan the Lone Wolf image was about 46 minutes. Thus, the longest time for total analysis of the 476 GB NTFS image was about 2 hours and 48 minutes (as seen in Table 9.11).

In summary, there is a performance trade-off that exists for prefix-based Generic Metadata Time Carving, where lowering the prefix parameter p may significantly increase recall, slightly reduces timestamp carving time, but can also significantly increase the filesystem specific parser time due to having the need to validate more potential timestamps.

9.6 Conclusion and Further Work

In this work, we created and applied a timestamp prefix matching version of the Generic Metadata Time Carving (GMTC) method [151]. The GMTC method can be used to carve for filesystem metadata records from a forensic image without the use of the filesystem, and can potentially allow for full file recovery on a damaged or partially overwritten disk. The crux of our contribution was the prefix-based potential timestamp carving algorithm, that only compares the prefixes of length p as opposed to the entire timestamp. This is because stringologically similar timestamps in most cases should be temporally similar as well. We tested the prefix-based method on three realistic forensic images. Two of the images used NTFS, one of the images used Ext4, and they varied in size from 1 GB to 476 GB.

Our location-based data recovery experiments mostly support our hypotheses. First, we have shown that applying timestamp prefix matching to the GMTC method can produce significantly greater recall in carving filesystem metadata records than the exact timestamp matching version. Surprisingly, performing prefix-based timestamp matching did not appear to affect the precision for carving MFT records or inodes from our test data, as we obtained 100% precision for all of our experiments. Further examinations in the Limitations subsection shows that prefix-based potential timestamp carving will increase the number of potentially valid offsets to metadata timestamps, but it is ultimately the filtering done by the filesystem specific parsers that controls the precision. However, running the parsers on an image without prior potential timestamp information will take significantly longer than using a GMTC method. Using the prefix-based Generic Metadata Time Carving method, the potential timestamp carver essentially performs data reduction of possible MFT record or inode locations for the filesystem specific parsers to check. The method appears to be practical, as our longest experiment on a 476 GB image in total clocked in at about 2 hours and 48 minutes.

Interestingly, changing the size of the matching prefix for the timestamps does not affect the time taken to perform timestamp carving by much. This makes sense as the prefix-based potential timestamp carving algorithm only added a constant number of steps to the original algorithm, therefore producing an algorithm with the same time complexity. Our experiments showed that timestamp carving with lower values of p took slightly less time than experiments with larger p values. On the other hand, reducing the size of the timestamp prefix often greatly increased the time taken by the filesystem specific parsers to extract the metadata records. This is due to the fact that matching for timestamps that are approximately similar results in some magnitudes more of potential timestamps to consider, and thus causing some magnitudes more time to run the filesystem specific parsers. We noted an exception to this rule for the large NTFS image, but this may be due to its implementation and the fact it does not use Python memory mapping libraries as the Ext4 parser does.

Future work for the prefix-based timestamp carving algorithm would be to try to improve its efficiency. Since the algorithm ingests the disk image in a linear fashion, perhaps the efficiency could be improved by using parallel processing to analyze different parts of the disk simultaneously, much like Garfinkel's Bulk Extractor [76]. The filesystem specific parsers can also be further optimized.

Our work has shown there needs to be improvements made to the filesystem specific parsers as well, so that they can handle more possible variations to filesystem metadata records. For instance, the NTFS parser needs to be able to handle MFT records with non-resident attributes. For Ext4, there needs to be hard link support, and better support for symbolic links. Then in general, there is also a need for development of parsers of filesystems other than NTFS and Ext4.

9.7 Acknowledgement

The research leading to these results has received funding from the Research Council of Norway programme IKTPLUSS, under the R&D project "Ars Forensica - Computational Forensics for Large-scale Fraud Detection, Crime Investigation & Prevention", grant agreement 248094/O70.

Appendix A

Novel Algorithms and Further Details

Here we list our contributed novel algorithms as well as supporting information. Note that Algorithm 13 has its own chapter, Appendix B, to reflect how it was represented in the original publication [172].

A.1 NFA update algorithm for *cedas*

Algorithm 8 is an implementation of the NFA from Paper 2 [174], where the edit distance k is limited to 2. The parameters include all Boolean values $R^{B}_{(i,e,s)}$ set by the user, which represents which terminal states of the NFA are turned on or off (1 for on, 0 for off). This is what handles the edit operation constraints. The bit-mask table $B[t]$ is also an input parameter, where the table must first be preprocessed. Characters t are read from the text under examination. The values $B[t]$ in the table refer to the binary vectors representing characters being read by the algorithm. Note, we have changed the algorithm style from the original paper to match our other algorithms. Also note that this is an algorithm for the NFA simulations. The string matching implementation also outputs the approximately matched strings, and not only notes that a match is found.

A.2 GMTC Potential Timestamp Carving Algorithm

Algorithm 9 is the potential timestamp carving algorithm from our GMTC method [151]. Again, we keep the original formatting of the algorithm, but below we provide the functions that the algorithm references, *stringToDecimal* as Algorithm 10 and *checkRepeatBytes* as Algorithm 11.

Algorithm 8: NFA update algorithm for *cedas* (detailed)

```
Input: File $T$, reading ASCII characters $t$
Output: Notes that approximate matching for keyword $P$ is found.
Bool $R^B_{(i,e,s)}$ # User input edit operation constraints for up to $k = 2$;
$B$ # Bitmask table reflecting keyword $P$;
Initialize all rows $R'$ to 0 except:
$R'(0,0,0) \leftarrow$ 0x00000001;
$R'(0,1,0) \leftarrow$ 0x00000002;
$R'(0,4,0) \leftarrow$ 0x00000004;
for each input character $t \in T$ do
    $R_{(0,0,0)} \leftarrow R'_{(0,0,0)}$;
    $R'_{(0,0,0)} \leftarrow ((R'_{(0,0,0)} << 1)\ \&\ B[t])\ |\ \texttt{0x00000001}$;
    $R_{(1,0,0)} \leftarrow R'_{(1,0,0)}$;
    $R'_{(1,0,0)} \leftarrow ((R'_{(1,0,0)} << 1)\ \&\ B[t]])\ |\ R_{(0,0,0)}$;
    $R_{(0,1,0)} \leftarrow R'_{(0,1,0)}$;
    $R'_{(0,1,0)} \leftarrow ((R'_{(0,1,0)} << 1)\ \&\ B[t]])\ |\ (R'_{(0,0,0)} << 1)$;
    $R_{(0,0,1)} \leftarrow R'_{(0,0,1)}$;
    $R'_{(0,0,1)} \leftarrow ((R'_{(0,0,1)} << 1)\ \&\ B[t]])\ |\ (R_{(0,0,0)} << 1)$;
    $R_{(0,1,1)} \leftarrow R'_{(0,1,1)}$;
    $R'_{(0,1,1)} \leftarrow ((R'_{(0,1,1)} << 1)\ \&\ B[t])\ |\ (R_{(0,1,0)} << 1)\ |$
                $(R'_{(0,0,1)} << 1)$;
    $R_{(1,0,1)} \leftarrow R'_{(1,0,1)}$;
    $R'_{(1,0,1)} \leftarrow ((R'_{(1,0,1)} << 1)\ \&\ B[t])\ |\ (R_{(1,0,0)} << 1)\ |\ R_{(0,0,1)}$;
    $R_{(1,1,0)} \leftarrow R'_{(1,1,0)}$;
    $R'_{(1,1,0)} \leftarrow ((R'_{(1,1,0)} << 1)\ \&\ B[t])\ |\ R_{(0,1,0)}\ |\ (R'_{(1,0,0)} << 1)$;
    $R_{(2,0,0)} \leftarrow R'_{(2,0,0)}$;
    $R'_{(2,0,0)} \leftarrow ((R'_{(2,0,0)} << 1)\ \&\ B[t])\ |\ R_{(1,0,0)}$;
    $R_{(0,2,0)} \leftarrow R'_{(0,2,0)}$;
    $R'_{(0,2,0)} \leftarrow ((R'_{(0,2,0)} << 1)\ \&\ B[t])\ |\ (R'_{(0,1,0)} << 1)$;
    $R_{(0,0,2)} \leftarrow R'_{(0,0,2)}$;
    $R'_{(0,0,2)} \leftarrow ((R'_{(0,0,2)} << 1)\ \&\ B[t])\ |\ (R_{(0,0,1)} << 1)$;
    if $(((R'_{(0,0,0)}\&R^B_{(0,0,0)})|(R'_{(0,0,1)}\&R^B_{(0,0,1)})|(R'_{(0,1,0)}\&R^B_{(0,1,0)})|$
        $(R'_{(1,0,0)}\&R^B_{(1,0,0)})|(R'_{(0,1,1)}\&R^B_{(0,1,1)})|(R'_{(1,0,1)}\&R^B_{(1,0,1)})|$
        $(R'_{(1,1,0)}\&R^B_{(1,1,0)})|(R'_{(2,0,0)}\&R^B_{(2,0,0)})|(R'_{(0,2,0)}\&R^B_{(0,2,0)})|$
        $(R'_{(0,0,2)}\&R^B_{(0,0,2)}))\&(0x00000001 << n-1))$ then
        Match is Found;
    end
end
```

Algorithm 9: Basic Potential Timestamp Carving Algorithm (Detailed)

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

Algorithm 10: stringToDecimal Function

```
Input: Raw disk image $T$ as a byte array. Contains character array $S$
       starting at index $i$.
Output: $output$ value of $S$
$m$ # Length of timestamp;
$i$ # Starting index of potential timestamp;
$output = 0$ # output value;
if ($m = 8$) then
    $output$ = ((unsigned long long)(((unsigned char) T[i]))) | (((unsigned
      long long)(((unsigned char) T[i + 1]))) « 8) | (((unsigned long
      long)(((unsigned char) T[i + 2]))) « 16) | (((unsigned long
      long)(((unsigned char) T[i + 3]))) « 24) | (((unsigned long
      long)(((unsigned char) T[i + 4]))) « 32) | (((unsigned long
      long)(((unsigned char) T[i + 5]))) « 40) | (((unsigned long
      long)(((unsigned char) T[i + 6]))) « 48) | (((unsigned long
      long)(((unsigned char) T[i + 7]))) « 56);
end
if ($m = 4$) then
    $output$ = ((unsigned long long)(((unsigned char) T[i]))) | (((unsigned
      long long)(((unsigned char) T[i + 1]))) « 8) | (((unsigned long
      long)(((unsigned char) T[i + 2]))) « 16) | (((unsigned long
      long)(((unsigned char) T[i + 3]))) « 24);
end
if ($m = 2$) then
    $output$ = ((unsigned long long)(((unsigned char) T[i]))) | (((unsigned
      long long)(((unsigned char) T[i + 1]))) « 8);
end
return $output$;
```

Algorithm 11: checkRepeatBytes Function

```
Input: Raw disk image $T$ as a byte array. Contains character array $S$
       starting at index $i$.
Output: Boolean output of whether character array $S$ is a repeated series
        of bytes.
$m$ # Length of timestamp;
$i$ # Starting index of potential timestamp;
$repeat = true$ # Repeat Boolean is assumed true until proven otherwise.
if (m == 8) then
    while ($r = 1; r < 8; r$++) do
        $repeat$ = $repeat$ and (T[i] == T[i + r]);
    end
end
if (m == 4) then
    while ($r = 1; r < 4; r$++) do
        $repeat$ = $repeat$ and (T[i] == T[i + r]);
    end
end
if (m == 2) then
    while ($r = 1; r < 2; r$++) do
        $repeat$ = $repeat$ and (T[i] == T[i + r]);
    end
end
return $repeat$
```

A.3 Prefix-Based Timestamp Equivalency Test

This algorithm is the prefix-based timestamp equivalency test, which is at the heart of Algorithm 13 in Appendix B.

Algorithm 12: Prefix matching algorithm.

```
Input: Big-endian unsigned long long forms of candidate
       timestamp $x$ and test sequence $y$
Output: Number of matches for the candidate timestamp either increases
        or stays the same
$m$ # Length of timestamp;
$p$ # Size of prefix;
xorResult = $x \bigoplus y$;
if ((xorResult >> (8*(m-p))) == 0) then
    matchCount += 1;
end
```

Appendix B

Prefix-Based Potential Timestamp Carving Algorithm

Note, we do not include the memory mapping aspects to handle large files. For the full potential timestamp carving program, see:

https://github.com/ TimestampPrefixCarving/ Peer-Review/blob/main/main.cpp.

Algorithm 13: Prefix-based potential timestamp carving algorithm, using timestamp prefix matching for the timestamp equivalency test.

```
Input: Raw disk image $T$ as a byte array. Parameters:
$m$ # Length of timestamp;
$k$ # Length of search threshold;
$h$ # Threshold for number of required matching timestamps per record;
$p$ # Prefix Length;
Output: Potential timestamp offsets from beginning of image (in bytes) in txt file.
$i$ = 0 # Current byte offset from start of image;
while $(i < |T| - k)$ do
    candidateTS = $T[i : (i + m)]$;
    # Boolean holding status of repeated byte sequence check;
    repeat = True;
    for $b \leftarrow 1$ to $m - 1$ by 1 do
        $repeat = repeat \,\&\, (candidateTS[0] == candidateTS[b])$;
    end
    # Variable for holding value of a string of characters read little-endian and transformed into a numerical value;
    littleEndian = 0;
    if $m$ == 8 then
        $littleEndian = (candidateTS[0] << (8 * 0) |...| (candidateTS[7] << (8 * 7))$;
    else if $m$ == 4 then
        $littleEndian = (candidateTS[0] << (8 * 0)) |...| (candidateTS[3] << (8 * 3))$;
    #If candidate timestamp is not a repeated sequence of bytes, and the prefix value of $littleEndian$ is not 0;
    if $(!repeat \,\&\, ((littleEndian >> 8 * (m - p)) \,!= 0))$ then
        matchCount = 0;
        $j = i + m$;
        while $(j < i + m + k)$ do
            $testSequence = 0$;
            if $m$ == 8 then
                $testSequence = (T[j] << (8 * 0)) |...| (T[j + 7] << (8 * 7))$;
            else if $m$ == 4 then
                $testSequence = (T[j] << (8 * 0)) |...| (T[j + 3] << (8 * 3))$;
            #Our timestamp prefix matching equivalency check;
            $xorResult$ = littleEndian $\bigoplus$ $testSequence$;
            if $((xorResult >> (8 * (m - p))) == 0)$ then
                $matchCount += 1$;
            end
            $j += m$;
            if $(matchCount >= (h - 1))$ then
                #Print Byte Location $i$;
                $j = i + m + k$;
                $i$ += $(k - m)$;
            end
        end
    end
    i += m ;
end
```

Bibliography

[1] Daubert v. Merrell Dow Pharmaceuticals, 509 US 579, 1993.

[2] Associated Press: Casey Anthony detectives missed 'suffocation' search, 2012. https://www.usatoday.com/story/news/nation/2012/11/25/casey-anthony-suffocation-google/1725253/.

[3] ACCESSDATA. Forensic toolkit (ftk) user guide. https://accessdata.com/product-download/forensic-toolkit-ftk-version-7.1.0", Nov 2018.

[4] ACTIVE KILLDISK. How to... for killdisk software. https://www.killdisk.com/load-bootdisk-win10.htm, Jan 2021.

[5] AHO, A. V., AND CORASICK, M. J. Efficient string matching: an aid to bibliographic search. *Communications of the ACM 18*, 6 (1975), 333–340.

[6] AL-ZAIDY, R., FUNG, B. C., YOUSSEF, A. M., AND FORTIN, F. Mining criminal networks from unstructured text documents. *Digital Investigation 8*, 3-4 (2012), 147–160.

[7] ALAZAB, M., VENKATRAMAN, S., AND WATTERS, P. Effective digital forensic analysis of the ntfs disk image. *Ubiquitous Computing and Communication Journal 4*, 1 (2009), 551–558.

[8] ANDA, F., LE-KHAC, N.-A., AND SCANLON, M. Deepuage: improving underage age estimation accuracy to aid csem investigation. *Forensic Science International: Digital Investigation 32* (2020), 300921.

[9] ANDERSEN, S. Technical report: A preliminary process model for investigation, 2019.

[10] APPLE. HFS plus volume format. https://developer.apple.com/library/archive/technotes/tn/tn1150.html, 2004.

[11] ÅRNES, A. *Digital Forensics*. John Wiley & Sons, 2017.

[12] BAEZA-YATES, AND G. NAVARRO, R. Faster approximate string matching. *Algorithmica 23*, 2 (1999), 127–158.

[13] BAEZA-YATES, R., AND GONNET, G. H. A new approach to text searching. *Communications of the ACM 35*, 10 (1992), 74–82.

[14] BAEZA-YATES, R. A., AND GONNET, G. H. A new approach to text searching. In *ACM SIGIR Forum* (1989), vol. 23, ACM, pp. 168–175.

[15] BAYNE, E. Accelerating digital forensic searching through GPGPU parallel processing techniques, 2017. PhD Dissertation.

[16] BAYNE, E. Openforensics: A digital forensics gpu pattern matching approach for the 21st century dfrws eu 2018 presentation. https://www.youtube.com/watch?v=YICTfSXMlb0, Jan 2019.

[17] BAYNE, E., FERGUSON, R. I., AND SAMPSON, A. Openforensics: A digital forensics gpu pattern matching approach for the 21st century. *Digital Investigation 24* (2018), S29–S37. The Proceedings of the Fifth Annual DFRWS Europe.

[18] BECKETT, J., AND SLAY, J. Digital forensics: Validation and verification in a dynamic work environment. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)* (2007), IEEE, pp. 266a–266a.

[19] BEEBE, N. Digital forensic research: The good, the bad and the unaddressed. In *IFIP International Conference on Digital Forensics* (2009), Springer, pp. 17–36.

[20] BEEBE, N., AND CLARK, J. Dealing with terabyte data sets in digital investigations. In *IFIP International Conference on Digital Forensics* (2005), Springer, pp. 3–16.

[21] BEEBE, N., AND DIETRICH, G. A new process model for text string searching. In *IFIP International Conference on Digital Forensics* (2007), Springer, pp. 179–191.

[22] BEEBE, N. L., AND CLARK, J. G. A hierarchical, objectives-based framework for the digital investigations process. *Digital Investigation 2*, 2 (2005), 147–167.

[23] BEEBE, N. L., AND CLARK, J. G. Digital forensic text string searching: Improving information retrieval effectiveness by thematically clustering search results. *Digital investigation 4* (2007), 49–54.

[24] BEEBE, N. L., CLARK, J. G., DIETRICH, G. B., KO, M. S., AND KO, D. Post-retrieval search hit clustering to improve information retrieval effectiveness: Two digital forensics case studies. *Decision Support Systems 51*, 4 (2011), 732–744.

[25] BEEBE, N. L., AND LIU, L. Clustering digital forensic string search output. *Digital Investigation 11*, 4 (2014), 314–322.

[26] BHAT, W. A., AND WANI, M. A. Forensic analysis of B-tree file system (Btrfs). *Digital Investigation 27* (2018), 57 – 70.

[27] BILENKO, M., MOONEY, R., COHEN, W., RAVIKUMAR, P., AND FIENBERG, S. Adaptive name matching in information integration. *IEEE Intelligent Systems 18*, 5 (2003), 16–23.

[28] BLEI, D. M., NG, A. Y., AND JORDAN, M. I. Latent dirichlet allocation. *Journal of machine Learning research 3*, Jan (2003), 993–1022.

[29] BOYER, R. S., AND MOORE, J. S. A fast string searching algorithm. *Communications of the ACM 20*, 10 (1977), 762–772.

[30] BRADLEY, J. R., AND GARFINKEL, S. L. Bulk extractor 1.4 user's manual. Tech. rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA DEPT OF COMPUTER SCIENCE, 2013.

[31] BREITINGER, F., GUTTMAN, B., MCCARRIN, M., ROUSSEV, V., AND WHITE, D. Approximate matching: definition and terminology. *NIST Special Publication 800* (2014), 168.

[32] BURROWS, M., AND WHEELER, D. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report* (1994), Citeseer.

[33] CANTONE, D., FARO, S., AND GIAQUINTA, E. A compact representation of nondeterministic (suffix) automata for the bit-parallel approach. In *Annual Symposium on Combinatorial Pattern Matching* (2010), Springer, pp. 288–298.

[34] CARRIER, B. *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[35] CARRIER, B., AND SPAFFORD, E. An event-based digital forensic investigation framework. *Digital Investigation* (2004).

[36] CASEY, E. *Digital evidence and computer crime: Forensic science, computers, and the internet*. Academic press, 2011.

[37] CASEY, E., FERRARO, M., AND NGUYEN, L. Investigation delayed is justice denied: proposals for expediting forensic examinations of digital evidence. *Journal of forensic sciences 54*, 6 (2009), 1353–1364.

[38] CASEY, E., NELSON, A., AND HYDE, J. Standardization of file recovery classification and authentication. *Digital Investigation 31* (2019), 100873.

[39] CHAU, M., XU, J. J., AND CHEN, H. Extracting meaningful entities from police narrative reports, 2002.

[40] CHITRAKAR, A. S., AND PETROVIC, S. Approximate search with constraints on indels with application in spam filtering. *Norsk informasjonssikkerhetskonferanse (NISK)* (2015), 22–33.

[41] CHITRAKAR, A. S., AND PETROVIC, S. Constrained row-based bit-parallel search in intrusion detection. *Norsk informasjonssikkerhetskonferanse (NISK)* (2016), 68–79.

[42] CHITRAKAR, A. S., AND PETROVIC, S. Collecting network evidence using constrained approximate search algorithms. In *IFIP International Conference on Digital Forensics* (2018), Springer, pp. 141–160.

[43] CHITRAKAR, A. S., AND PETROVIĆ, S. Crbp-optype: A constrained approximate search algorithm for detecting similar attack patterns. In *Cyber-ICPS 2017/SECPRE 2017*. LNCS 10683, 2018, pp. 163–176.

[44] CHO, G.-S. A computer forensic method for detecting timestamp forgery in ntfs. *Computers & Security 34* (2013), 36 – 46.

[45] CHO, G.-S., AND ROGERS, M. K. Finding forensic information on creating a folder in $logfile of ntfs. In *International Conference on Digital Forensics and Cyber Crime* (2011), Springer, pp. 211–225.

[46] COHEN, M. I. Advanced carving techniques. *Digital Investigation 4*, 3-4 (2007), 119–128.

[47] COHEN, W. Enron email corpus. https://www.cs.cmu.edu/$\sim$./enron/, May 2015.

[48] COWEN, D., AND SEYER, M. File system journal analysis. https://digital-forensics.sans.org/community/summits, 2013.

[49] DA SILVA, R., STASIU, R., ORENGO, V. M., AND HEUSER, C. A. Measuring quality of similarity functions in approximate data matching. *Journal of Informetrics 1*, 1 (2007), 35–46.

[50] DAMERAU, F. J. A technique for computer detection and correction of spelling errors. *Communications of the ACM 7*, 3 (1964), 171–176.

[51] DAVIS, J., AND GOADRICH, M. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning* (2006), pp. 233–240.

[52] DE WAAL, A., VENTER, J., AND BARNARD, E. Applying topic modeling to forensic data. In *IFIP International Conference on Digital Forensics* (2008), Springer, pp. 115–126.

[53] DECHERCHI, S., TACCONI, S., REDI, J., LEONCINI, A., SANGIACOMO, F., AND ZUNINO, R. Text clustering for digital forensics analysis. In *Computational Intelligence in Security for Information Systems*. Springer, 2009, pp. 29–36.

[54] DEPARTMENT OF DEFENCE CYBER CRIME CENTER. Dc3dd. https://sourceforge.net/projects/dc3dd/files/dc3dd/7.2.646/, 2012. Last visited: 2019-09-19.

[55] DEWALD, A., AND SEUFERT, S. Afeic: Advanced forensic Ext4 inode carving. *Digital Investigation 20* (2017), S83 – S91. The Proceedings of the Fourth Annual DFRWS Europe.

[56] DLUGOSCH, P., BROWN, D., GLENDENNING, P., LEVENTHAL, M., AND NOYES, H. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE Transactions on Parallel and Distributed Systems 25*, 12 (2014), 3088–3098.

[57] DÖMÖLKI, B. A universal compiler system based on production rules. *BIT Numerical Mathematics 8*, 4 (1968), 262–275.

[58] DTSEARCH. Search features - search types. https://www.dtsearch.com/PLF_Features_2.html.

[59] DYKSTRA, J. *Essential cybersecurity science: build, test, and evaluate secure systems*. " O'Reilly Media, Inc.", 2015.

[60] ELASTIC. Elasticsearch reference: Fuzzy query. https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-fuzzy-query.html, 2017.

[61] ELASTIC. Custom score for fuzzy matching based on levenshtein distance score. https://discuss.elastic.co/t/custom-score-for-fuzzy-matching-based-on-levenshtein-distance-score/125544, Mar 2018.

[62] EXT4 (AND EXT2/EXT3) WIKI. Ext4 disk layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout, Aug 2019.

[63] EXT4 DEVELOPMENT TEAM. Ext4 header file. https://github.com/torvalds/linux/blob/master/fs/ext4/ext4.h, 2019. Last visited: 2019-09-11, code from the master development on github.

[64] FAIRBANKS, K. D. An analysis of ext4 for digital forensics. *Digital investigation 9* (2012), S118–S130.

[65] FARO, S., AND LECROQ, T. Twenty years of bit-parallelism in string matching. *Festschrift for Borivoj Melichar* (2012), 72–101.

[66] FLATCAP, R. Linux ntfs project. https://flatcap.org/linux-ntfs/, 2004.

[67] FORENSIC FOCUS. Interpretation of ntfs timestamps. https://www.forensicfocus.com/articles/interpretation-of-ntfs-timestamps/, Apr 2013.

[68] FORENSICSWIKI.XYZ. File carving. https://forensicswiki.xyz/wiki/index.php?title=File_Carving, Sep 2012.

[69] FORENSICSWIKI.XYZ. Mac times. https://forensicswiki.xyz/wiki/index.php?title=MAC_times, Jul 2021.

[70] FOUNDATION, F. S. Id database utilities. www.gnu.org/software/idutils/manual/idutils.html, 2012.

[71] FRANKE, K., AND SRIHARI, S. N. Computational forensics: An overview. In *International Workshop on Computational Forensics* (2008), Springer, pp. 1–10.

[72] GARFINKEL, S., FARRELL, P., ROUSSEV, V., AND DINOLT, G. Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation 6* (2009), S2–S11.

[73] GARFINKEL, S. L. Forensic feature extraction and cross-drive analysis. *Digital Investigation 3* (2006), 71–81.

[74] GARFINKEL, S. L. Carving contiguous and fragmented files with fast object validation. *Digital Investigation 4* (2007), 2 – 12.

[75] GARFINKEL, S. L. Digital forensics research: The next 10 years. *Digital Investigation 7* (2010), S64–S73.

[76] GARFINKEL, S. L. Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security 32* (2013), 56 – 72.

[77] GARFINKEL, S. L., AND MCCARRIN, M. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. *Digital Investigation 14* (2015), S95–S105.

[78] GAUCH JR, H. G., GAUCH, H. G., AND GAUCH JR, H. G. *Scientific method in practice*. Cambridge University Press, 2003.

[79] GIANNELLI, P. C. Forensic science: Daubert's failure. *Case W. Res. L. Rev. 68* (2017), 869.

[80] GIRISH, K., AND SUNIL, J. J. General relations between partially ordered multisets and their chains and antichains. *Mathematical Communications 14*, 2 (2009), 193–205.

[81] GLADYSHEV, P., AND JAMES, J. I. Decision-theoretic file carving. *Digital Investigation 22* (2017), 46 – 61.

[82] GLADYSHEV, P., AND PATEL, A. Finite state machine approach to digital event reconstruction. *Digital Investigation 1*, 2 (2004), 130–149.

[83] GÖBEL, T., AND BAIER, H. Anti-forensic capacity and detection rating of hidden data in the ext4 filesystem. In *IFIP International Conference on Digital Forensics* (2018), Springer, pp. 87–110.

[84] GOLIĆ, J. D. Edit distance correlation attacks on clock-controlled combiners with memory. In *Australasian Conference on Information Security and Privacy* (1996), Springer, pp. 169–181.

[85] GOLIĆ, J. D., AND MENICOCCI, R. Edit distance correlation attack on the alternating step generator. In *Annual International Cryptology Conference* (1997), Springer, pp. 499–512.

[86] GOLIĆ, J. D., AND MIHALJEVIĆ, M. J. A generalized correlation attack on a class of stream ciphers based on the Levenshtein distance. *Journal of Cryptology 3*, 3 (1991), 201–212.

[87] GUARINO, A. *Digital Forensics as a Big Data Challenge*. Springer Fachmedien Wiesbaden, Wiesbaden, 2013, pp. 197–203.

[88] GUO, Y., SLAY, J., AND BECKETT, J. Validation and verification of computer forensic software tools—searching function. *Digital Investigation 6* (2009), S12–S22.

[89] GÖBEL, T., AND BAIER, H. Anti-forensics in ext4: On secrecy and usability of timestamp-based data hiding. *Digital Investigation 24* (2018), S111 – S120.

[90] HALL, P. A., AND DOWLING, G. R. Approximate string matching. *ACM computing surveys (CSUR) 12*, 4 (1980), 381–402.

[91] HAMM, J. Extended FAT file system. https://paradigmsolutions.files.wordpress.com/2009/12/exfat-excerpt-1-4.pdf, 2009. Last visited 2018-09-16.

[92] HAMM, J. Carve for records, not files. https://digital-forensics.sans.org/summit-archives/2012/carve-for-record-not-files.pdf, 2012. Last visited 2021-18-04.

[93] HAMMING, R. W. Error detecting and error correcting codes. *The Bell system technical journal 29*, 2 (1950), 147–160.

[94] HANSEN, J., PORTER, K., SHALAGINOV, A., AND FRANKE, K. Comparing open source search engine functionality, efficiency and effectiveness with respect to digital forensic search. *NISK Journal* (2018).

[95] HANSEN, K. H., AND TOOLAN, F. Decoding the APFS file system. *Digital Investigation 22* (2017), 107 – 132.

[96] HARICHANDRAN, V. S., BREITINGER, F., AND BAGGILI, I. Bytewise approximate matching: The good, the bad, and the unknown. *The Journal of Digital Forensics, Security and Law: JDFSL 11*, 2 (2016), 59.

[97] HOLUB, J. Bit parallelism-nfa simulation. In *International Conference on Implementation and Application of Automata* (2001), Springer, pp. 149–160.

[98] HOLUB, J. The finite automata approaches in stringology. *Kybernetika 48*, 3 (2012), 386–401.

[99] HUBER, P. Junk silence in the courtroom. *Val. UL Rev. 26* (1992), 723–755.

[100] HYYRÖ, H. Improving the bit-parallel NFA of Baeza-Yates and Navarro for approximate string matching. *Information Processing Letters 108*, 5 (2008), 313–319.

[101] INNOCENCE PROJECT. Why bite mark evidence should never be used in criminal trials. https://innocenceproject.org/what-is-bite-mark-evidence-forensic-science/, Apr 2020.

[102] INTELLA. Individual solutions, 2017. https://www.vound-software.com/individual-solutions.

[103] IQBAL, F., FUNG, B. C., AND DEBBABI, M. Mining criminal networks from chat log. In *2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology* (2012), vol. 1, IEEE, pp. 332–337.

[104] KARP, R. M., AND RABIN, M. O. Efficient randomized pattern-matching algorithms. *IBM journal of research and development 31*, 2 (1987), 249–260.

[105] KARRESAND, M., AXELSSON, S., AND DYRKOLBOTN, G. O. Using ntfs cluster allocation behavior to find the location of user data. *Digital Investigation 29* (2019), S51–S60.

[106] KARRESAND, M., DYRKOLBOTN, G. O., AND AXELSSON, S. An empirical study of the ntfs cluster allocation behavior over time. *Forensic Science International: Digital Investigation 33* (2020), 301008.

[107] KARRESAND, M., WARNQVIST, A., LINDAHL, D., AXELSSON, S., AND DYRKOLBOTN, G. O. Creating a map of user data in ntfs to improve file carving. In *Advances in Digital Forensics XV* (Cham, 2019), G. Peterson and S. Shenoi, Eds., Springer International Publishing, pp. 133–158.

[108] KNUTSON, T., AND CARBONE, R. Filesystem timestamps: What makes them tick? *GIAC GCFA Gold Certification 11* (2016).

[109] KOHONEN, T. *Self-organizing maps*, vol. 30. Springer Science & Business Media, 2012.

[110] KORNBLUM, J. Identifying almost identical files using context triggered piecewise hashing. *Digital investigation 3* (2006), 91–97.

[111] KORNBLUM, J., AND KENDALL, K. Foremost. foremost.sourceforge.net.

[112] LABORATORY, F. R. C. F. Annual report for fiscal year 2016, 2017.

[113] LAURENSON, T. Performance analysis of file carving tools. In *IFIP International Information Security Conference* (2013), Springer, pp. 419–433.

[114] LEPINSKY, R. Analyzing keywords in enron's email. https://rodgersnotes.wordpress.com/2013/11/24/analyzing-keywords-in-enrons-email/, 2013.

[115] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady* (1966), vol. 10, pp. 707–710.

[116] LIAO, Y.-C. A survey of software-based string matching algorithms for forensic analysis. *Annual ADFSL Conference on Digital Forensics, Security and Law* (2015), 77–86.

[117] LIEBLER, L., SCHMITT, P., BAIER, H., AND BREITINGER, F. On efficiency of artifact lookup strategies in digital forensics. *Digital Investigation 28* (2019), S116–S125.

[118] LILLIS, D., BECKER, B., O'SULLIVAN, T., AND SCANLON, M. Current challenges and future research areas for digital forensic investigation. *arXiv preprint arXiv:1604.03850* (2016).

[119] LILLIS, D., AND SCANLON, M. On the benefits of information retrieval and information extraction techniques applied to digital forensics. In *Advanced Multimedia and Ubiquitous Engineering*. Springer, 2016, pp. 641–647.

[120] LILLIS, D., AND SCANLON, M. *On the Benefits of Information Retrieval and Information Extraction Techniques Applied to Digital Forensics*. Springer Singapore, Singapore, 2016, pp. 641–647.

[121] LIN, C.-H., LIU, C.-H., CHIEN, L.-S., AND CHANG, S.-C. Accelerating pattern matching using a novel parallel algorithm on gpus. *IEEE Transactions on Computers 62*, 10 (2012), 1906–1916.

[122] LOPER, E., AND BIRD, S. Nltk: the natural language toolkit. *arXiv preprint cs/0205028* (2002).

[123] LOUIS, A., DE WAAL, A., AND VENTER, C. Named entity recognition in a south african context. In *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries* (2006), pp. 170–179.

[124] LOUIS, A., AND ENGELBRECHT, A. P. Unsupervised discovery of relations for analysis of textual data. *Digital Investigation 7*, 3-4 (2011), 154–171.

[125] LYLE, J. R. If error rate is such a simple concept, why don't i have one for my forensic tool yet? *Digital Investigation 7* (2010), S135–S139.

[126] MAAR, R. Ext4magic. https://github.com/gktrk/ext4magic, 2014.

[127] MARZIALE, L., RICHARD III, G. G., AND ROUSSEV, V. Massive threading: Using gpus to increase the performance of digital forensics tools. *Digital Investigation 4* (2007), 73–81.

[128] MCCASH, J. Timestamped registry & NTFS artifacts from unallocated space. https://digital-forensics.sans.org/blog/2010/05/04/timestamped-registry-ntfs-artifacts-unallocated-space, May 2010.

[129] MENEZES, A. J., VAN OORSCHOT, P. C., AND VANSTONE, S. A. *Handbook of applied cryptography*. CRC press, 2018.

[130] MICROSOFT. How ntfs works. https://docs.microsoft.com/pt-pt/previous-versions/windows/server/cc781134(v=ws.10)?redirectedfrom=MSDN, 2008.

[131] MICROSOFT. How basic disks and volumes work. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc739412(v=ws.10)?redirectedfrom=MSDN, Oct 2009.

[132] MIHOV, S., AND SCHULZ, K. U. Fast approximate search in large dictionaries. *Computational Linguistics 30*, 4 (2004), 451–477.

[133] MISHRA, S. Keyword indexing and searching for large forensics targets using distributed computing. Master's thesis, University of New Orleans Theses and Dissertations. 510, 2007. http://scholarworks.uno.edu/td/510.

[134] MOORE, GARFINKEL, FARRELL, ROUSSEV, AND DINOLT. 2018 lone wolf scenario. https://digitalcorpora.org/corpora/scenarios/2018-lone-wolf-scenario. Last visited: 2020-04-16.

[135] MUELLER, L. Search for windows 64 bit timestamps. http://www.forensickb.com/2008/01/search-for-windows-64-bit-timestamps.html, Jan. 2008.

[136] MYERS, G. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM (JACM) 46*, 3 (1999), 395–415.

[137] NADEAU, D., AND SEKINE, S. A survey of named entity recognition and classification. *Lingvisticae Investigationes 30*, 1 (2007), 3–26.

[138] NAVARRO, G. A guided tour to approximate string matching. *ACM computing surveys (CSUR) 33*, 1 (2001), 31–88.

[139] NAVARRO, G. Nr-grep: a fast and flexible pattern-matching tool. *Software: Practice and Experience 31*, 13 (2001), 1265–1312.

[140] NAVARRO, G., AND PREZZA, N. Universal compressed text indexing. *Theoretical Computer Science 762* (2019), 41–50.

[141] NAVARRO, G., AND RAFFINOT, M. A bit-parallel approach to suffix automata: Fast extended string matching. In *Annual Symposium on Combinatorial Pattern Matching* (1998), Springer, pp. 14–33.

[142] NAVARRO, G., AND RAFFINOT, M. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, 2002.

[143] NAVARRO, G., AND RAFFINOT, M. New techniques for regular expression searching. *Algorithmica 41*, 2 (2005), 89–116.

[144] NIST. Forensic string searching tool test requirements specification, public draft 1 of version 1.0. https://www.nist.gov/document/cftt-fss-forensic-string-search-specification-v1-draft-1, Jan 2008.

[145] NIST. Computer forensic reference data sets: Deleted file recovery. https://www.cfreds.nist.gov/dfr-test-images.html, Mar 2017.

[146] NIST. Computer forensics tool testing program (cftt). https://www.nist.gov/itl/ssd/software-quality-group/computer-forensics-tool-testing-program-cftt, 2019.

[147] NIST. Forensic string searching tool test assertions and test plan, public draft 1 of version 1.0. https://www.nist.gov/document/cftt-doc-forensic-string-searching-tool-test-assertions-and-test
Mar 2019.

[148] NOEL, G. E., AND PETERSON, G. L. Applicability of latent dirichlet allocation to multi-disk search. *Digital Investigation 11*, 1 (2014), 43–56.

[149] NORDVIK, R., GEORGES, H., TOOLAN, F., AND AXELSSON, S. Reverse engineering of ReFS. *Digital Investigation 30* (2019), 127 – 147.

[150] NORDVIK, R., PORTER, K., TOOLAN, F., AXELSSON, S., AND FRANKE, K. cPTS carve for potential timestamps. https://github.com/RuneN007/cPTS, 2020. Last visited 2020-03-20.

[151] NORDVIK, R., PORTER, K., TOOLAN, F., AXELSSON, S., AND FRANKE, K. Generic metadata time carving. *Digital Investigation 33* (2020), 301005. The Proceedings of the Twentieth Annual DFRWS USA.

[152] OOMMEN, B. J. Constrained string editing. *Information Sciences 40*, 3 (1986), 267–284.

[153] OOMMEN, B. J. Recognition of noisy subsequences using constrained edit distances. *IEEE transactions on pattern analysis and machine intelligence*, 5 (1987), 676–685.

[154] OOMMEN, B. J., AND LEE, W. Constrained tree editing. *Information sciences 77*, 3-4 (1994), 253–273.

[155] PAL, A., AND MEMON, N. The evolution of file carving. *IEEE signal processing magazine 26*, 2 (2009), 59–71.

[156] PALMER, G., AND CORPORATION, M. A Road Map for Digital Forensic Research. Tech. rep., Nov. 2001. accessed 29.04.17.

[157] PALMER, G., ET AL. A road map for digital forensic research. In *First Digital Forensic Research Workshop, Utica, New York* (2001), pp. 27–30.

[158] PERRY, J. W., KENT, A., AND BERRY, M. M. Machine literature searching x. machine language; factors underlying its design and development. *American Documentation 6*, 4 (1955), 242–254.

[159] PETROVIC, S., AND FRANKE, K. Improving the efficiency of digital forensic search by means of the constrained edit distance. In *Information Assurance and Security, 2007. IAS 2007. Third International Symposium on* (2007), IEEE, pp. 405–410.

[160] PETROVIC, S., AND FRANKE, K. A new two-stage search procedure for misuse detection. In *Future Generation Communication and Networking (FGCN 2007)* (2007), vol. 2, IEEE, pp. 418–422.

[161] PETROVIĆ, S. V., AND GOLIĆ, J. D. String editing under a combination of constraints. *Information sciences 74*, 1-2 (1993), 151–163.

[162] PLATZER, C., STUETZ, M., AND LINDORFER, M. Skin sheriff: a machine learning solution for detecting explicit images. In *Proceedings of the 2nd international workshop on Security and forensics in communication systems* (2014), ACM, pp. 45–56.

[163] PLUM, J., AND DEWALD, A. Forensic APFS File Recovery. In *Proceedings of the 13th International Conference on Availability, Reliability and Security* (New York, NY, USA, 2018), ARES 2018, ACM, pp. 47:1–47:10.

[164] POLLITT, M. The hermeneutics of the hard drive: Using narratology, natural language processing, and knowledge management to improve the effectiveness of the digital forensic process, 2013.

[165] POLLITT, M., AND WHITLEDGE, A. Exploring big haystacks. In *Advances in digital forensics II*. Springer, 2006, pp. 67–76.

[166] POLLITT, M. M. Triage: A practical solution or admission of failure. *Digital Investigation 10*, 2 (2013), 87–88.

[167] POMERANZ, H. Understanding ext4 (part 1): Extents. https://www.sans.org/blog/understanding-ext4-part-1-extents/, Dec 2010.

[168] POMERANZ, H. Ext3 file recovery via indirect blocks. https://www.fireeye.com/blog/threat-research/2011/01/ext3-file-recovery-indirect-blocks.html, Jan 2011.

[169] POMERANZ, H. Understanding ext4 (part 2): Timestamps. https://www.sans.org/blog/understanding-ext4-part-2-timestamps/, March 2011.

[170] POMERANZ, H. Understanding ext4 (part 6): Directories. https://www.sans.org/blog/understanding-ext4-part-6-directories/, Jun 2017.

[171] PORTER, K. Analyzing the darknetmarkets subreddit for evolutions of tools and trends using lda topic modeling. *Digital Investigation 26* (2018), S87–S97.

[172] PORTER, K., NORDVIK, R., TOOLAN, F., AND AXELSSON, S. Timestamp prefix carving for filesystem metadata extraction. *Forensic Science International: Digital Investigation 38* (2021), 301266.

[173] PORTER, K., AND PETROVIC, S. On application of constrained edit distance algorithms to cryptanalysis and digital forensics. *NISK Journal* (2017), 112–123.

[174] PORTER, K., AND PETROVIC, S. Obtaining precision-recall trade-offs in fuzzy searches of large email corpora. In *IFIP International Conference on Digital Forensics* (2018), Springer, pp. 67–85.

[175] QUICK, D., AND CHOO, K.-K. R. Data reduction and data mining framework for digital forensic evidence: storage, intelligence, review and archive. *Trends and Issues in Crime and Criminal Justice*, 480 (2014), 1–11.

[176] QUICK, D., AND CHOO, K.-K. R. Impacts of increasing volume of digital forensic data: A survey and future research challenges. *Digital Investigation 11*, 4 (2014), 273–294.

[177] QUICK, D., AND CHOO, K.-K. R. Big forensic data reduction: digital forensic images and electronic evidence. *Cluster Computing 19*, 2 (2016), 723–740.

[178] REES, T. Taxamatch, an algorithm for near ('fuzzy') matching of scientific names in taxonomic databases. *PloS one 9*, 9 (2014), e107510.

[179] RELATIVITY. Introduction to dtsearch. https://www.relativity.com/ediscovery-training/self-paced/introduction-to-dtsearch/. accessed 11.06.21.

[180] RICHARD, G., ROUSSEV, V., AND MARZIALE, L. In-place file carving. In *IFIP International Conference on Digital Forensics* (2007), Springer, pp. 217–230.

[181] Richard III, G. G., and Roussev, V. Scalpel: A frugal, high performance file carver. In *Proceedings of the Digital Forensic Research Conference* (2005).

[182] Roussev, V. Hashing and data fingerprinting in digital forensics. *IEEE Security & Privacy 7*, 2 (2009), 49–55.

[183] Roussev, V. Data fingerprinting with similarity digests. In *IFIP International Conference on Digital Forensics* (2010), Springer, pp. 207–226.

[184] Roussev, V., and Richard III, G. G. Breaking the performance wall: The case for distributed digital forensics. In *Proceedings of the 2004 digital forensics research workshop* (2004), vol. 94.

[185] Roy, I., Srivastava, A., Nourian, M., Becchi, M., and Aluru, S. High performance pattern matching using the automata processor. In *Parallel and Distributed Processing Symposium, 2016 IEEE International* (2016), IEEE, pp. 1123–1132.

[186] Russo, L., Navarro, G., Oliveira, A. L., and Morales, P. Approximate string matching with compressed indexes. *Algorithms 2*, 3 (2009), 1105–1136.

[187] Sankoff, D. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences 69*, 1 (1972), 4–6.

[188] Schicht, J. Logfileparser readme. https://github.com/jschicht/LogFileParser, 2018.

[189] School, C. L. Daubert standard. https://www.law.cornell.edu/wex/daubert_standard.

[190] Schulz, K. U., and Mihov, S. Fast string correction with levenshtein automata. *International Journal on Document Analysis and Recognition 5*, 1 (2002), 67–85.

[191] Sellers, P. H. The theory and computation of evolutionary distances: pattern recognition. *Journal of algorithms 1*, 4 (1980), 359–373.

[192] Siegenthaler, T. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on computers 1*, C-34 (1985), 81–85.

[193] Sipser, M. *Introduction to the Theory of Computation*, vol. 2. Thomson Course Technology Boston, 2006.

[194] SOLTOW, K. Vmware esxi disk provision. how does it work, what is the difference, and which one is better for me? https://www.vmwareblog.org/vmware-esxi-disk-provision-work-difference-one-better/, 21 2019.

[195] SON, N., LEE, Y., KIM, D., JAMES, J. I., LEE, S., AND LEE, K. A study of user data integrity during acquisition of android devices. *Digital Investigation 10* (2013), S3 – S11. The Proceedings of the Thirteenth Annual DFRWS Conference.

[196] STEWART, J., AND UCKELMAN, J. Searching massive data streams using multipattern regular expressions. In *IFIP International Conference on Digital Forensics* (2011), Springer, pp. 49–63.

[197] STEWART, J., AND UCKELMAN, J. Unicode search of dirty data, or: How i learned to stop worrying and love unicode technical standard# 18. *Digital Investigation 10* (2013), S116–S125.

[198] TOMMY TRACY, I., STAN, M., BRUNELLE, N., WADDEN, J., WANG, K., SKADRON, K., AND ROBINS, G. Nondeterministic finite automata in hardware-the case of the levenshtein automaton. *Architectures and Systems for Big Data (ASBD), in conjunction with ISCA* (2015).

[199] TWRP. Download twrp-3.3.1-2-dreamlte.img.tar, May 2019. https://eu.dl.twrp.me/dreamlte/twrp-3.3.1-2-dreamlte.img.tar.html.

[200] UKKONEN, E. Algorithms for approximate string matching. *Information and control 64*, 1-3 (1985), 100–118.

[201] UKKONEN, E. Approximate string-matching with q-grams and maximal matches. *Theoretical computer science 92*, 1 (1992), 191–211.

[202] VAN BANERVELD, M., LE-KHAC, N.-A., AND KECHADI, M.-T. Performance evaluation of a natural language processing approach applied in white collar crime investigation. In *International Conference on Future Data and Security Engineering* (2014), Springer, pp. 29–43.

[203] VAN DER MEER, V., JONKER, H., AND VAN DEN BOS, J. A contemporary investigation of ntfs file fragmentation. *Digital Investigation* (2021).

[204] VIDAS, T., ZHANG, C., AND CHRISTIN, N. Toward a general collection methodology for android devices. *Digital Investigation 8* (2011), S14 – S24. The Proceedings of the Eleventh Annual DFRWS Conference.

[205] WAGNER, R. A., AND FISCHER, M. J. The string-to-string correction problem. *Journal of the ACM (JACM) 21*, 1 (1974), 168–173.

[206] WANG, K., SADREDINI, E., AND SKADRON, K. Hierarchical pattern mining with the automata processor. *International Journal of Parallel Programming* (2017), 1–36.

[207] WEISSTEIN, E. W. Cover. `https://mathworld.wolfram.com/Cover.html`.

[208] WU, S., AND MANBER, U. Agrep–a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference* (1992), San Francisco, pp. 153–162.

[209] WU, S., AND MANBER, U. Fast text searching allowing errors. *Communications of the ACM 35*, 10 (1992), 83–92.

[210] WU, W., CHOW, K.-P., MAI, Y., AND ZHANG, J. Public opinion monitoring for proactive crime detection using named entity recognition. In *IFIP International Conference on Digital Forensics* (2020), Springer, pp. 203–214.

[211] YANG, M., AND CHOW, K.-P. An information extraction framework for digital forensic investigations. In *IFIP International Conference on Digital Forensics* (2015), Springer, pp. 61–76.

[212] ZHA, X., AND SAHNI, S. Fast in-place file carving for digital forensics. In *International Conference on Forensics in Telecommunications, Information, and Multimedia* (2010), Springer, pp. 141–158.

[213] ZHA, X., AND SAHNI, S. Gpu-to-gpu and host-to-host multipattern string matching on a gpu. *IEEE Transactions on Computers 62*, 6 (2012), 1156–1169.

[214] ZIV, J., AND LEMPEL, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory 24*, 5 (1978), 530–536.

ISBN 978-82-326-5619-6 (printed ver.)
ISBN 978-82-326-6396-5 (electronic ver.)
ISSN 1503-8181 (printed ver.)
ISSN 2703-8084 (online ver.)