|  |
| --- |
| Developing File Extraction Methods for  Open-Source Forensics Software in C#  Mikolaj M. Mroz  BSc (Hons) Ethical Hacking, 2023  Supervised by Ian Ferguson and Dr. Ethan Bayne |

|  |
| --- |
| School of Design and Informatics  Abertay University |

Table of Contents

[Acknowledgements v](#_Toc135172391)

[Abstract v](#_Toc135172392)

[Context v](#_Toc135172393)

[Aim v](#_Toc135172394)

[Results vi](#_Toc135172395)

[Abbreviations, Symbols and Notation vi](#_Toc135172396)

[1. Introduction 1](#_Toc135172397)

[1.1. Background 1](#_Toc135172398)

[1.2. Aims 2](#_Toc135172399)

[1.3. Research Question 3](#_Toc135172400)

[Overview 3](#_Toc135172401)

[2. Literature Review 4](#_Toc135172402)

[2.1. Introduction to Literature 4](#_Toc135172403)

[2.2. String Searching 4](#_Toc135172404)

[2.3. File Detection 6](#_Toc135172405)

[2.3.1. Existing File Detection Algorithms 6](#_Toc135172406)

[2.3.2. File Header-Trailer (FHT) Algorithm 6](#_Toc135172407)

[2.3.3. Byte Frequency Analysis Algorithm 8](#_Toc135172408)

[2.3.4. File-Structure based Algorithm 9](#_Toc135172409)

[2.3.5. Object Validation 9](#_Toc135172410)

[2.3.6. Extension-based detection 10](#_Toc135172411)

[2.4. The Future of File Carving 11](#_Toc135172412)

[2.5 Summary 12](#_Toc135172413)

[3. Methodology 13](#_Toc135172414)

[3.1. Overview 13](#_Toc135172415)

[3.2. Planning 14](#_Toc135172416)

[3.3 Design 14](#_Toc135172417)

[3.3.1. Existing Structure 15](#_Toc135172418)

[3.3.2. Refactored Structure 17](#_Toc135172419)

[3.3.3. Filetype-Specific Targets 18](#_Toc135172420)

[3.4. Development 18](#_Toc135172421)

[3.4.1. General refactoring 18](#_Toc135172422)

[3.4.2. ZIP detection 19](#_Toc135172423)

[3.2.3. Integration 20](#_Toc135172424)

[3.2.4. Locating the Central File Header 21](#_Toc135172425)

[3.2.5. Locating the End of Central Directory 22](#_Toc135172426)

[3.2.6. Locating the End of File 22](#_Toc135172427)

[3.5. Testing 23](#_Toc135172428)

[3.5.1. End-of-Sprint Tests 23](#_Toc135172429)

[3.5.2. End-of-Development Tests 24](#_Toc135172430)

[3.5.3. Testing Environment 24](#_Toc135172431)

[3.5.4. Testing Methodology 25](#_Toc135172432)

[3.6.1. Review 27](#_Toc135172433)

[4. Results 29](#_Toc135172434)

[4.1. Completed Program 29](#_Toc135172435)

[4.2. Processing Speed 29](#_Toc135172436)

[4.3. Accuracy 30](#_Toc135172437)

[5. Discussion 31](#_Toc135172438)

[5.1. Original Algorithm Implementation 31](#_Toc135172439)

[5.2. Identifying Solutions 31](#_Toc135172440)

[5.3. Implementing a Solution 32](#_Toc135172441)

[5.4. Testing 32](#_Toc135172442)

[5.5. Evaluation of Results 32](#_Toc135172443)

[5.6. Limitations 34](#_Toc135172444)

[5.6.1. Out of Bounds Index Crash 34](#_Toc135172445)

[5.6.2. ZIP Varieties 34](#_Toc135172446)

[5.6.3. Lack of Available Testing Images 35](#_Toc135172447)

[5.6.4. Lack of C# refactoring Information 35](#_Toc135172448)

[5.6.5. Original OF Testing Setup 35](#_Toc135172449)

[5.6.6. RecoverMyFiles and Foremost Output 36](#_Toc135172450)

[5.6.7. OF Filetype Inconsistencies 36](#_Toc135172451)

[6. Conclusion 37](#_Toc135172452)

[6.1. Future Work 38](#_Toc135172453)

[7. References 39](#_Toc135172454)

[8. Appendices 42](#_Toc135172455)

[8.1 Diagrams 42](#_Toc135172456)

[8.1.1. ZIP End of File Calculation Diagram 42](#_Toc135172457)

[8.1.2. OpenForensics – Original Release Benchmark 43](#_Toc135172458)

[8.2. Output 43](#_Toc135172459)

[8.2.1. OpenForensics (Original) 43](#_Toc135172460)

[8.2.2 OpenForensics (Refactored) 44](#_Toc135172461)

[8.2.3. Foremost 46](#_Toc135172462)

[8.3 Code 47](#_Toc135172463)

[8.3.1. Analysis.cs 47](#_Toc135172464)

[8.3.2. Changes to FileTypes.xml 52](#_Toc135172465)

Table of Figures

[Figure 1 - OpenForensics Main Menu 5](#_Toc135171920)

[Figure 2 - Global OS Market Share (StatCounter, 2023) 10](#_Toc135171921)

[Figure 3 - An overview of OpenForensics' logic (Bayne, et al., 2018) 15](#_Toc135171922)

[Figure 4 - An example of a corrupt JPG (Barker Software, n.d.) 16](#_Toc135171923)

[Figure 5 - An overview of the refactored OpenForensics logic 17](#_Toc135171924)

[Figure 6 - Refactored ProcessLocations function 19](#_Toc135171925)

[Figure 7 - The reference to CalculateFileEnd in ProcessFoundLocations 20](#_Toc135171926)

[Figure 8 - A switch case passing in the filetype before initialising the various ZIP structs 21](#_Toc135171927)

[Figure 9 – A collation of all processing speed results for Test 1 and Test 2 after calculating the mean. 29](#_Toc135171928)

Table of Tables

[Table 1 - The internal structure of a ZIP file 19](#_Toc135171929)

[Table 2 - Local File Header Signature contents 22](#_Toc135171930)

[Table 3 - End of Central Directory contents 23](#_Toc135171931)

[Table 4 - Testing Environment Details 25](#_Toc135171932)

[Table 5 - Basic File Carving Test contents (Test 1) 26](#_Toc135171933)

[Table 6 - Numbered ZIP Test contents (Test 2) 26](#_Toc135171934)

[Table 7 - A summary of results of each software from Test 1 and Test 2 30](#_Toc135171935)

[Table 8 - A breakdown of each software's accuracy results for Test 1 30](#_Toc135171936)

[Table 9 - A breakdown of each software's accuracy results for Test 2 30](file:///C:\Users\mikol\Desktop\Thesis\Mikolaj_Mroz_Thesis2.docx#_Toc135171937)

# Acknowledgements

Firstly, a huge thank you to the Abertay course organisers, module leaders, and my supervisors Dr. Ethan Bayne and Ian Ferguson for equipping me with the knowledge I needed to tackle my learning and giving me opportunities and experience I could only dream of in preparation for making my way up in the world.

I would like to sincerely thank my amazing friends, my ever-supporting family, and my beautiful girlfriend Himali for helping me through all the rough times over the course of my studies, keeping my spirits high.

# Abstract

## Context

OpenForensics is an open-source file carving tool originally made to demonstrate the benefits of using advanced GPU pattern-matching techniques to perform file carving in a digital forensics investigation. Its primary aim was to showcase a quantifiable improvement in processing times for reading in and analysing digital forensic data when compared to other CPU-restricted tools commonly used today.

It successfully achieved this aim, albeit with a simplistic approach to file carving; the files it ‘carves out’ from disk images are done so using a basic ‘file header-footer’ carving algorithm, only taking into account one unique identifier at the beginning (the header) and one at the end (the footer). This implementation causes a number of problems with certain file structures such as .ZIPs, which use multiple nested headers and footers to contain their data, and as a result, are misidentified by the program and output as several broken files instead.

## Aim

This paper aimed to demonstrate the methodology behind refactoring OpenForensics’ file detection and validation algorithms to improve the program’s reliability in processing realistic forensic datasets and accurately reproducing a vast variety of different filetypes from a given disk image. These improvements were implemented into locally stored source code and its accuracy was compared to the original pre-refactor release of OpenForensics, as well as a group of currently available file-carving tools to determine a measurable impact on speed and accuracy, just as was originally done in the initial of the program.

## Results

Results showed that the introduction of a semantic file-detection algorithm, a success rate of 100% was achieved in the extraction of ZIP files during testing. Processing times for file searching and carving were also improved, taking up to 20% less time for the program to finish its file analysis phase. The paper concludes by saying that semantic file detection algorithms can greatly improve the detection rate of forensic evidence from a given target and urges further development in the field of digital forensics, particularly with regards to standardisation.

# Abbreviations, Symbols and Notation

|  |  |
| --- | --- |
| Abbreviation | Definition |
| OF | OpenForensics |
| DF | Digital Forensics |
| CPU | Central Processing Unit |
| GPU | Graphics Processing Unit |
| PC | Personal Computer |
| FHT | File Header-Trailer (Algorithm) |
| BFA | Byte Frequency Analysis (Algorithm) |
| BFC | Byte Frequency Cross-correlation (Algorithm) |
| Image | A digital disk image (.dd) file format |
| File | A reconstructed file object, extracted from an image by a carving program |

# 1. Introduction

## 1.1. Background

The unprecedented rate at which digital technology has developed and expanded in recent years has piqued the interest of criminals around the world in search of a new, effective, and convenient medium to facilitate their cybercrimes – digital storage. An estimated 90% of all crimes committed now make use of at least one digital element, spurring digital forensics to develop as a progressively more sophisticated and standalone branch of forensic science. With this expansion comes a desperate need for reliable software tools to help digital forensics agents stay two steps ahead of the criminals they investigate, as their own malicious ways of covering their digital tracks grow increasingly complex in a never-ending arms race.

On top of this, investigating and producing evidence in a court of law can be a complicated and lengthy procedure, and any delays in producing the necessary refined results has the potential to greatly delay the judicial process. His Majesty’s Inspectorate of Constabulary and Fire & Rescue Services (HMICFRS, 2022) recently reported that there exist a record-breaking 25,000 seized devices that remain unexamined by forensic forces due to delays. This greatly insufficient, particularly for a process which aims to protect people from potentially life-altering dangers on the internet. The processing speeds and thoroughness of the searches these tools typically utilise during the forensic acquisition process is imperative to the success of an investigation, though the tools of the “golden age” of digital forensics (characterised by large-scale coordinated efforts to produce and improve software such as file carvers) are unfortunately becoming outdated. Garfinkel, a data analyst whose work has become greatly respected in the field of digital forensics, accurately predicted this crisis in his article “Digital Forensics: The Next 10 Years” (2010), citing a lack of standardisation and innovation.

OpenForensics (OF) was created as an answer to this lack of development. OF is an opensource digital file carving tool for Windows which aims to improve on the shortfalls of other currently available solutions, namely with its state-of-the-art advanced GPU processing algorithm which allows it to carve files from file systems faster than any other CPU-bound tool currently available. OF can extract data from both disk images and physical drives, utilising a two-pass approach to first discover files relevant to the user’s requirements before extracting them from an internally stored database of each file encountered during the search, along with various characteristics such as the header location, footer location, file length, and other additional flags.

Although OF seems like the perfect solution to the issues plaguing digital forensics in recent years, it has one major flaw keeping it back from achieving its full potential – its file detection and validation algorithm.

## 1.2. Aims

The aim of this project is to refactor OpenForensics with improved file validation algorithms in order to produce more reliable results from its file carving operations, in turn making the tool more fit for purpose in a professional forensic acquisition setting. The objectives of this report are listed below:

* Investigate the techniques employed by the current version of OpenForensics (ver. 1.85b) to identify and validate files during file carving and analysis.
* Identify relevant techniques which, if implemented, would quantifiably improve OpenForensics’ reliability in analysing and reproducing carved files.
* Implement a semantic, file structure-based validation algorithm to expand upon the existing header-trailer extraction methods.
* Test the newly refactored solution in a manner that replicates the testing performed by the author in its initial release, comparing the performance of tool against its own original results, as well as those of comparable tools.
* Evaluate and discuss the results gathered, highlighting aspects of the software that have been improved with the introduction of the new file detection algorithm.

## 1.3. Research Question

* How can the accuracy of open-source digital forensic software be affected through the use of semantic file carving algorithms?

## Overview

The following chapter explores the literature surrounding the topic of digital forensics, file carving, as well as file validation, highlighting current technologies and what can be expected in the future of digital forensics acquisition tools. The third chapter, the methodology, describes in detail the actions taken to during the planning and development stages of creating the final project artefact as a means of documenting and justifying the respective procedures. Finally, the report closes with a discussion on the results gathered, reviewing how successfully the project met its aims before making a conclusion and recommending any future work that remains to be done in the field.

# 2. Literature Review

## 2.1. Introduction to Literature

The following section aims to give a critical overview of various journals, papers, and publications related to the research of file detection all the while detailing relevant trends, concepts, and issues found in current implementations of file detection and string searching algorithms. Relevant aspects of this research will then be incorporated into the final practical solution for the project, implementing solutions based on research and findings.

## 2.2. String Searching

One of the most important aspects of a successful file detection is the use of an appropriate string searching algorithm. Before any file can be stored and processed, it must first be discovered within the target dataset, and it must be done quickly to minimise the amount of processing time during any given operation. These are the roles of a string searching algorithm, it being the foundation upon which file detection is built on.

The Boyer Moore (BM) algorithm (Boyer & Moore, 1977) employs the use of a skip table to jump past characters irrelevant to the searched string. A pre-processing phase takes place where the algorithm determines which symbols are present in the search string as well as their locations, allowing the algorithm to jump in dynamic increments through the target data resulting in improved performance with longer search patterns. However, it is only well suited for single-pattern searching operations, and has therefore needed to be customised to work effectively within multi-pattern searching operations such as those within Foremost (Kendall & Mikus, n.d.) and Scalpel (Richard III & Roussev, 2005). Its worst-case matching complexity is O(mn), and best case is O(n/m) where n is the length of the text and m is the length of the target pattern. However, the processing time of an unmodified BM algorithm increases linearly with the number of patterns to search and is therefore greatly inefficient for file carvers (Zha & Sahni, 2010).

OF’s use of the GPU for its file carving operations places lots of importance on its chosen multi-pattern string searching system which utilises a modified Aho-Corasick (AC) algorithm (Aho & Corasick, 1975). The algorithm constructs a dictionary of the words being searched and uses a trie structure to effectively analyse if a target string is found, quickly transitioning between failed matches and potential word endings if a string only partly matches. Modern academic sources agree that AC is currently the best available option to fit this purpose (Arudchutha, et al., 2013, Skrbina & Stojanovski, 2012) citing its impressive performance in multi-pattern searching scenarios. A modified Parallel Failureless AC (PFAC) algorithm (Lin, et al., 2013) is used within OF achieving a best-case complexity of O(1) and worst-case O(m). This makes it significantly quicker than BM, and its compatibility with multiple target strings made it an ideal fit for OF.

A screenshot of a computer

Description automatically generated

Figure - OpenForensics Main Menu

Simpler ‘brute-force’ (BF) string searching algorithms also have unique use cases in software development; the creation of such an algorithm can take seconds as all that needs to be done is to create a loop with a start point, an endpoint, and a check that runs to determine if the correct value has been found or the end has been reached. In C#, this can be done with a simple ‘for‘ loop and may be the best solution for less complex problems particularly when time constraints are an issue. As a trade-off, BF searches are typically much slower than both BM and AC. The searches produce a low performance overhead if used with appropriately small-scale problems but can greatly slow programs down if applied to bigger issues. One should never implement a BF to be the main process behind file detection or carving.

## 2.3. File Detection

### 2.3.1. Existing File Detection Algorithms

Many different methods can be used to perform file carving, though they can generally be summarized into a small number of groups to simplify identification as noted by [source].

* Header/Footer carving
* Header/Embedded length carving
* File structure-based carving
* Carving with validation
* Header/Maximum file size carving.

It is important to note that while these are considered ‘carving’ algorithms, their title is in fact a reference to how files are detected and verified before they are extracted (carved), rather than the algorithm responsible for performing the carve itself.

### 2.3.2. File Header-Trailer (FHT) Algorithm

The Header/Footer algorithm is a relatively simple method of carving data from storage. The algorithm searches through a defined dataset looking for file headers, an identifier present at the beginning of most file types, and footers which exist at the end of the file. Using the footer and trailer as markers to begin and end the carve respectively, it can then theoretically extract everything between these two markers to produce a fully functional output file.

In order for the algorithm to produce a valid result however, the following criteria must be met as noted by Christiaan Beek, lead scientist of McAfee CTO (2017):

* Files cannot be fragmented across several storage clusters.
* The beginning of the file (footer) cannot be cut off.
* The markers being searched for cannot appear outside of file header definitions.

While this algorithm can technically function, these three points make FHT a very flawed method of successfully extracting any amount of data from a realistic dataset where files are frequently fragmented across a range of distances and to varying extents as demonstrated by Simson Garfinkel’s study into the frequency of file fragmentation (2007). It is also impossible to guarantee a dataset free of any strings the algorithm may target that appear outside of the expected file header definitions, making it a poor choice for anything outside of the most basic file detection or validation processes. This is exemplified by the issues FHT has correctly detecting and extracting nested file structures in the original version of OpenForensics.

To account for the shortfalls of FHT, many improved file detection algorithms were developed to consider the contents of each different file structure with the aim of producing more accurate results from file carving tools. These range from algorithms that verify the existence of special flags unique to the filetype prior to extraction, to more complex methods that analyse the frequency of different byte values and compare them to sample file “fingerprints”, with each solution showing a qualitative improvement in results over those from FHT.

### 2.3.3. Byte Frequency Analysis Algorithm

The Byte Frequency Analysis (BFA) algorithm investigates the frequency at which certain byte values occur under the theory that each filetype has its own unique byte frequency “fingerprint”. The algorithm uses these fingerprints to compare to the data extracted from an image without needing to read any metadata as is often done with FHT and other popular algorithms.

BFA was first developed for the 36th annual Hawaii International Conference on System Sciences (HICSS) to propose a new way of determining the type of file scanned under the hypothesis that these fingerprints will be able to accurately determine filetypes faster than a FHT algorithm. McDaniel et al. first built ‘fingerprints’ of each filetype – their unique byte frequency graphs – by simply counting the occurrence of each byte value (ranging from 0 to 255) and plotting it on a graph. An average line was then calculated and stored onto a fingerprint file which the program would then use during runtime to compare to. Within this study, they also modified the BFA algorithm to consider the average difference in frequency between every byte pair as well as the “correlation strength” calculation found in the BFA algorithm, dealing with outlier results more effectively. This was named the Byte Frequency Cross-correlation algorithm (BFC)

Results show that the BFA algorithm was the fastest of the three tested (BFA, BFC, FHT) only taking 0.010 seconds to compare 25 fingerprints, improving marginally over FHT’s 0.015 seconds. BFC took the longest at 1.19 seconds. While the processing times may first appear impressive, both the BFA and BFC algorithms struggled greatly in accuracy, its second aim, managing to successfully identify only 29.17% and 45.83% of the files respectively. These results are leagues behind the common forensic standard algorithm, FHT, which boasts an impressive 95.83% accuracy rating, showing that Byte Frequency-based algorithms require further development before they can be considered practical in a professional forensic setting.

### 2.3.4. File-Structure based Algorithm

File Structure-based algorithms, also referred to as ‘Deep Carvers’ or ‘Semantic Carvers’, use the internal structure of a given file to influence its extraction process. This includes but is not limited to any embedded file size information that can help the carver function more accurately determine the true EOF. It is commonly used in conjunction with advanced object validation to select extracted bytes and to confirm that they function correctly as part of the carved whole file (Cohen, 2007), which can be used to combat the issue of fragmented files. This affects approximately 6% of all carved files according to Garfinkel’s report, and while a 94% non-fragmentation rate is objectively high, there remains a chance that digital information vital to a forensics investigation may not be extracted, impacting its outcome. Therefore, object validation is an important component of file carving to consider when producing or refactoring forensic tools.

### 2.3.5. Object Validation

Nick Mikus, co-author of the file carving tool Foremost, implemented object validation into the software due to its initial simplistic header-footer carving algorithm, similar to what can be found in the original version of OpenForensics. While writing his master’s thesis, Mikus discovered that by implementing object validation methods, he was able to greatly improve the reliability of Foremost in extracting files in a controlled environment when compared to other similar tools like FTK and ILOOK, as well as the original version of Foremost (v0.69) (Mikus, 2005).

Test results showed that his object validation implementation was able to successfully extract 7/11 (63%) of JPEG files in a given NTFS image, 10/14 (71%) varied filetypes in a FAT32 image, and 5/10 (50%) in an EXT2 sample while keeping the MD5 hash intact to that of the originals.

While comparing MD5 hashes is a near fool proof way of validating that a file has been transferred and extracted correctly, it is worth noting that there were several instances where a file was successfully reconstructed but the MD5 hashes did not match due to miniscule differences in bytes. Taking this into account, the results are as follows for the revised Foremost object validation: 11/11 (100%) of the JPEG files in an NTFS image, 14/15 (93%) of varied files in a FAT32, and 10/10 (100%) in an EXT2 context where Foremost v0.69 was successfully extracting around four in the worst-case scenario and ILOOK managing only one successfully extracted file.

### 2.3.6. Extension-based detection

Extension-based file detection is one of the simplest and most naïve ways of examining the type of a file and as a result, it has very little weight in the field of digital forensics. The algorithm offers extremely fast processing due to its simplicity and can therefore process huge volumes of data in a very small timeframe if paired with an appropriate string searching algorithm; this can be greatly beneficial to forensic investigations that rely on strict deadlines and schedules. Extensions are also commonplace in Windows operating systems which make up 65.89% of the global Desktop OS market share as of April 2023, thus being applicable to the vast majority of global computer systems.

Figure - Global OS Market Share (StatCounter, 2023)

The main issues lie in how easy it is to spoof a file extension – in the case of Windows 11, the latest and OS from Microsoft, all that needs to be done is to right click on a file and rename it. Renaming successfully tricks the OS into thinking it is a different file and causes a host of issues for extension-based algorithms, antivirus software, and end users which without proper checks in place, will misidentify any file that has undergone this change. Therefore, this method is not appropriate to use in a professional file carver.

## 2.4. The Future of File Carving

In “Digital forensics research: The next 10 years”, a review into the state of digital forensics towards the end of the early 2000s, Garfinkel (2010) argues that the golden age of forensic research and development had come to an unfortunate and bleak end. The author cites the Sisyphean struggle of an ever-increasing number of storage devices that need to be analysed during an investigation -a side-effect of the growth of Internet of Things (IoT) devices- and a progressive rise in research and development costs for improved analysis software. A lack of standardisation within the field of file analysis has also greatly slowed progress, as developers are found to be collaborating on projects much less often than in previous years due to differences in priorities and a general lack of ‘togetherness’. These factors contribute to a stagnation in progress towards faster and more accurate file detection and validation techniques, resulting in forensic investigators relying on the same tools and their same flaws that have been present for over a decade.

This backlog has been thoroughly detailed in other articles, the authors of which agree that the increase in cases requiring digital analysis and the volume of data being gathered from IoT and cloud devices are some of the most egregious factors contributing to delays within law enforcement agencies (Montasari & Hill, 2019). Delays in receiving analysis results have risen to 4 years in extreme cases and are expected to increase further with the growth of the digital storage and IoT industries (Scanlon, 2016).

Finally, more recent research unanimously agrees that the potential academic research and development in file carving holds can be game-changing for the field. In “Performance Analysis of File Carving Tools”, Laurenson (2013) first praises the research performed from 2007-2009 on state-of-the-art file carvers investigating new, faster ways of extracting data through the use of GPUs investigated by Marziale et al (2007), de-fragmentation (Sencar, 2009), and in-place file carving (Richard III, Roussev, & Marziale, 2007), bringing innovation to field which, according to Garfinkel, had reached the end of its golden age. However, the author concludes that despite this innovation, future research remains to be done to improve file carving efficiency and accuracy all the while minimising false results. Laurenson concludes by saying “Future advancement of techniques and tools based on academic research could greatly improve the performance of file carving tools”, suggesting new data abstraction techniques, further development in file validation, and the creation of an automated post-processor for extracted results, agreeing with Garfinkel’s call for more coordinated development in the digital forensics landscape.

## 2.5 Summary

In summary, the technologies behind file detection and file carving within digital forensic tools continue to remain underdeveloped. Many new algorithms have been researched and developed in recent years, but FHT and its file-structure based alternatives continue to hold the best trade-off between processing speed and accuracy. Despite this, it is vital for more academic research to be done into producing more state-of-the-art file analysis techniques in the hope of a new breakthrough in order to decrease wait times for law enforcement evidence results, and perhaps bring forth a new forensic golden age to catch up to the modern developments in digital storage, IoT, and cloud computing.

# 3. Methodology

## 3.1. Overview

A modified Agile development methodology was chosen for the project. This allowed for a repeated development cycle consisting of planning, designing, developing, and testing the code before reviewing progress and future plans with the supervisor on a bi-weekly basis to ensure that sufficient progress was being made in each cycle or “sprint”, with the ultimate goal of keeping the project on track until it was finished according to the deadlines defined in the Gannt chart. The “deployment” section of the agile cycle was deemed irrelevant in the development of this program where no code is definitively deployed anywhere until after development was finished, and thus was omitted from this framework.

The agile methodology was chosen as careful planning and regular supervisor reviews are at this model’s core, setting a strong foundation for consistent and reliable progress throughout the development cycle. Agile is also a particularly scalable framework, not only because of its customisable nature as demonstrated by altering the definition of the “deployment” section, but it is often noted as being a highly effective framework for a range of different projects and team sizes (Visual Paradigm, 2023).

In this case, development featured two other participants with whom communication was maintained over the development cycle:

* The supervisor, who oversaw the project as a whole and provided feedback regarding progress, functionality, and to whom any project issues were raised during a bi-weekly review.
* The author of OpenForensics, who published the original OpenForensics solution and paper and was able to offer extensive knowledge of the program’s workings.

## 3.2. Planning

At the beginning of each Agile sprint, a simple bullet-pointed plan of objectives was written up on how to implement or improve the features discussed with the supervisor in the review. Planning took place on Microsoft Teams during the discussion of next steps for the project as it allowed for detailed brainstorming and immediate feedback on ideas from the supervisor, which improved understanding about the project goals for all parties involved.

During the first planning meeting, it was decided that it would be most effective to begin development on the simpler filetypes present in the file detection section of the program, with the JPEG filetype being chosen as the best starting point. The plan was to gradually go through discussed filetypes in increasing complexity under the supervisor’s guidance before finally implementing an improved ZIP detection algorithm, which is one of the main objectives of this investigation. This allowed for ample time to gain familiarity with the code itself as well as the chosen development workflow before reaching the more complex detection algorithms towards the end of development.

Regardless of the chosen filetype, the solution had to follow best programming practices, and therefore be scalable such that new filetype detection algorithms could be implemented with the aid of the added framework. In this situation, a C# ‘switch’ case was determined to be best fit as a solution since it could pass the filetype variable through the encompassing function and determine which filetype-specific steps to take to discover the EOF.

## 3.3 Design

In order to meet the aims of the project, a suitable file detection algorithm first had to be chosen. The solution needed to be well documented, effective, memory-efficient, lightweight, and did not need for any particularly large chunks of the source code of OpenForensics to be re-written, keeping to the scope of the sections of code relevant to file detection and validation. It was also important to select a solution that was feasible to implement in time for the given deadline, the workload of which could easily be segmented into chunks could later be used to form the main subject of each subsequent review session.

### 3.3.1. Existing Structure

Prior to refactoring, the file detection algorithm was described by Bayne et al. (2018) as using a two-pass approach for detecting and extracting files from the selected drive or disk image. The first pass iterates through the contents and marks the beginning and the end of each predefined file until the endpoint of the drive or disk image is reached. These locations are stored into an array each time a valid file pattern is found, which the CPU then pairs with a matching file footer until the full scan is complete and file carving is started by the user.

A picture containing text, diagram, screenshot, plan

Description automatically generated

Figure - An overview of OpenForensics' logic (Bayne, et al., 2018)

From this information, it was clear that the carving algorithm itself did not need to be altered, as file detection algorithms are only concerned with determining the start and endpoint of the extraction which are inherently adaptable – extractable files will always be in different places when using different sources of data. Thus, what needed to be changed was the method which the program uses to determine the end of the file. Simply scanning for the next occurrence of a footer and extracting everything up to that point causes issues for filetypes that may have multiple footers, resulting in the file being extracted in an incomplete or corrupted form.

A sunflower in the sky

Description automatically generated with low confidence

Figure - An example of a corrupt JPG (Barker Software, n.d.)

Another important aspect of designing the solution was how it would interact with the filetypes that were part of the existing structure but outside the scope of the project. Firstly, developing bespoke EOF calculations for every originally supported filetype is an incredibly technical and time-consuming task, and therefore could not be possible within the given timeframe. It is also vital for the refactored code to not limit the existing structure to any newly implemented functionality. In other words, while the existing file detection process utilising footer lookup-tables is flawed, it still produces more reliable results than if it was entirely removed or replaced with an incomplete alternative. Because of this, any new functionality must be layered on top of the detection code that already exists.

### 3.3.2. Refactored Structure

A file structure-based algorithm was selected as it best suited the required criteria. This algorithm was based on file detection code used in Foremost, which during testing presented higher rate of successfully extracting complex filetypes. The design of this algorithm is largely filetype dependant but the refactoring process of each supported filetype follows a common pattern:

A picture containing diagram, text, plan, technical drawing

Description automatically generated

Figure - An overview of the refactored OpenForensics logic

From Figure 5, it can be seen that the refactored structure makes minimal changes to existing code, and instead builds on what already exists because of the programs initial well-structured build. The most significant changes occur around the file validation functionality which performs additional calculations on sections of the buffer to determine the true End of File (EOF) location as opposed to searching for the next footer occurrence. The way this is achieved varies between filetypes, but most commonly involves either finding the size of the file written within its internal structure or otherwise ensuring the bits being treated as the EOF are in fact appropriate and will not result in a broken file export.

### 3.3.3. Filetype-Specific Targets

## 3.4. Development

The development stage defined the act of taking relevant aspects of planning, research, and design and incorporating them throughout the core OF refactoring process. Development took place in a local Visual Studio 2022 environment due to its sizable extension library and scalability with projects of different sizes. The program’s use of IntelliSense, a powerful code completion and debugging aid, also greatly benefitted the workflow of the project by reducing overall debugging time along with its built-in performance profiler which helped in gathering accurate timing and memory efficiency data for testing.

### 3.4.1. General refactoring

Before implementing the file-specific detection algorithms, a general refactoring had to be performed on the processes and functions within OpenForensics as the pre-existing code was not built to accommodate semantic file carving. An example of this is the filetypes.xml file which lists the headers, EOF, and other characteristics of the files to be carved. The issue with this implementation is that some files do not utilise EOF at all, relying instead on a built-in file size value in its metadata or use multiple headers and footers, which the pre-existing algorithm could not extract correctly. A proposed solution to this problem was to move the EOF value from Filetypes.xml which is inherently static into the appropriate C# source code file (Analysis.cs) in order to perform the appropriate EOF calculations. These cannot be done in an XML file and will allow the program to function identically on a surface level if implemented correctly. Another change had to be made to the function responsible for looking through the header and footer lookup table. The lookup table is utilised by the program’s PFAC string searching algorithm to locate matching FHT byte sequences quickly and efficiently within the buffer. However, it was necessary to stop the program from using its original file detection process for ZIP files. This was done by adding a line of code to the end of the following if statement to ensure ZIP files are not processed using the original method.

A screen shot of a computer code

Description automatically generated with low confidence

Figure - Refactored ProcessLocations function

### 3.4.2. ZIP detection

ZIP files are commonly utilized as a way of grouping files together, acting similarly to a standard windows ‘folder’ on the surface level. There are however a number of differences between these two structures:

* ZIPs compress any data put into them, while typical windows folders do not. This is done to save on storage space and to make the files easier to transmit across a network without losing any information in the process.
* The internal file structure is fundamentally different as result, with ZIPs utilising the following format (Table 1).

|  |
| --- |
| Internal ZIP Structure |
| Local file header 1 |
| File data 1 |
| File descriptor 1 |
| … |
| Local file header (n) |
| File data (n) |
| File descriptor (n) |
| Central Directory |
| End of Central Directory |

Table - The internal structure of a ZIP file

Based on research performed by Mikus (2005), it was deemed most efficient to jump between each these file headers until the EOF was reached. Each local file structure must contain a “compressed size” field, which the algorithm can use as a clue to jump to the next local file header instance until finally a search can be conducted for the end of the central directory record. This process ultimately saves on processing time, as only the most vital components of the file structure are considered when determining the end of the central file directory.

### 3.2.3. Integration

Firstly, the file headers written in Filetypes.xml had to be located within the buffer, a segmented stream of data from the target file or directory. This process is a part of the original OF release and therefore did not need to be adjusted. Once all instances of the headers were located, each header must be passed through a new function called “CalculateFileEnd”. This function requires the found filetype, its location, and passes the entire buffer in by reference such that calculations could be performed based on the data after the header.

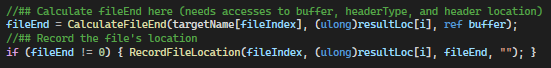


Figure - The reference to CalculateFileEnd in ProcessFoundLocations

The function makes use of a switch statement to determine what actions to take for each filetype. If a ZIP header (0x504B0304) is passed, it must make use of the appropriate ZIP case whereupon the local file header, central file header, and end of central directory structs are initialised, ready to be filled out by the results of the calculations that follow. A switch case was chosen as it allows for cleaner code when only one variable is expected to change – in this case, the filetype being passed in.

It is vital to set the struct variables to be unsigned where no negative results are expected, using uint instead of int and ushort instead of short. Failure to do this may cause values to overflow into the negative and potentially undermine the accuracy of the program. The header structs mimic the structure of each ZIP header for visualisation and scalability purposes, and therefore they must use accurate datatypes. (See Appendix 8.3.1.1).

A picture containing text, screenshot, font

Description automatically generated

Figure - A switch case passing in the filetype before initialising the various ZIP structs

A sanity check must first be performed to ensure the correct header has been passed into the switch case by checking that the third and fourth bytes after the result location are in fact 0x03 and 0x04. If the header is valid, the calculations may continue otherwise the function is broken out of and the remaining headers are processed.

### 3.2.4. Locating the Central File Header

The function stores the values located in byte offsets 14, 18, 26, and 28 from the local file header. These correspond to whether the contents are encrypted (CRC), the compressed file length, the filename length, and any extra information at the end of the local file header which must all be used to locate the next header. By moving along to offset 30 and applying these found lengths, another check can take place to see if the current position belongs to the central file header (CFH), which comes directly after all other contained file data. Unless the ZIP file only contains one file, the program must loop back to checking for the local file header again until the CFH is finally found. This was done using a While loop, as it is not possible to predict exactly how may times this loop will need to run.

|  |  |  |
| --- | --- | --- |
| Total Byte Offset | Field Name | Size |
| 0 | Local file header signature (0x504B0304) | 4 bytes |
| 4 | Version | 2 bytes |
| 6 | General purpose bit flag | 2 bytes |
| 8 | Compression method | 2 bytes |
| 10 | Last modification time | 2 bytes |
| 12 | Last modification date | 2 bytes |
| 14 | CRC-32 | 4 bytes |
| 18 | Compressed Size | 4 bytes |
| 22 | Uncompressed size (or 0xffffffff for ZIP64) | 4 bytes |
| 26 | File name length (*n*) | 2 bytes |
| 28 | Extra field length (*m*) | 2 bytes |
| 30 | File name | N bytes |
| 30+n | Extra field | M bytes |

Table - Local File Header Signature contents

### 3.2.5. Locating the End of Central Directory

By this point, the CFH had been located, and the discovery of the End of Central File Directory (EOCD) and End of File locations (EOF) could take place by string searching for 0x0506. This was done using a simple bruteforce string search as the EOCD is very often only a few bytes away from the end of the CFH, and thus does not have a quantifiable impact the processing speed. In the rare case that the CFH is found but the EOCD is not, the program must break. This would suggest that the file is fragmented or corrupted and is therefore not carvable. This will also occur if the EOCD is found but its comment length is not, the value of which is typically located at byte offset 18 within the EOCD.

### 3.2.6. Locating the End of File

Finally, after storing the EOCD commend length, the position in the buffer is moved along to the end of the EOCD structure and the comment length is applied, leading to the EOF. This location is then returned as “FileEnd” by the function (CalculateFileEnd) and is recorded using the RecordFileLocation function which appends the result to the list of all other calculated header and footer locations. This marks the end of the refactored code. See Appendix 8.1.1. for a flow diagram design of the process undertaken by the refactored version of OpenForensics.

|  |  |  |
| --- | --- | --- |
| Total Byte Offset | Field Name | Size |
| 0 | End of central directory signature  (0x504B0506) | 4 bytes |
| 4 | Number of this disk | 2 bytes |
| 6 | Number of the disk with start of central directory | 2 bytes |
| 8 | Total number of entries in this disk’s central directory | 2 bytes |
| 10 | Size of the central directory | 4 bytes |
| 14 | Offset of start of central directory with respect to starting disk number | 4 bytes |
| 18 | .ZIP comment length (N) | 2 bytes |
| 20 | .ZIP comment | N bytes |
| N | End of File |  |

Table - End of Central Directory contents

## 3.5. Testing

Testing was performed multiple times throughout the course of the project, with each test being grouped into one of two categories: end-of-sprint tests and end-of-development tests.

### 3.5.1. End-of-Sprint Tests

These tests were performed to ensure the code written during a particular sprint achieved its aims in preparation for the review stage, where various aspects of the code’s functionality were discussed with the project supervisor. The results of this testing acted as guidance for next stages of the project and as a general check-up on progress made.

Over the course of the sprint, the Digital Forensics Tool Testing Images by Carrier and Mikus (2010) were used to test the effectiveness of the newly implemented algorithm. These images were chosen as they contain a variety of different filetypes both fragmented and unfragmented to varying extents while also including details about the number of files that are stored and all the characteristics that may prevent them from being acknowledged by primitive file detection algorithms.

Using these details, it was possible to compare the original OF file detection algorithm as well as the refactored version against what should be detected and recovered. The co-founder of this image file project, Nick Mikus, is also closely associated with the sophisticated “Foremost” file carving tool, making the images an effective test case for the OF refactoring project.

### 3.5.2. End-of-Development Tests

End-of-Development tests were performed after the final development cycle was completed in order to demonstrate the final effectiveness of the implemented solutions. By comparing the processing speed, memory usage, and reliability in extracting files from both digital and physical forms of evidence, it was possible to determine the success of the project as a whole and identify areas for improvement.

### 3.5.3. Testing Environment

See Table 4 for a breakdown of the testing systems. The Windows machine was used to test RecoverMyFiles as well as OF and its refactored version – these programs were built for a Windows system. A Kali Linux machine had to be utilized to perform MD5 hash calculations using the ‘md5sum’ command and file carving using Foremost, which was built for use in the Linux command-line.

|  |  |
| --- | --- |
| BASE PC | |
| Platform Role | Desktop |
| Operating System | Windows 11 Home Version 10.0.22621 Build 22621 |
| Processor | AMD Ryzen 7 3700X 8-Core Processor, 3600 MHz, 8 Cores, 16 Logical Processors |
| Memory | Corsair Vengeance LPX Black 16GB (2x8GB) 3600 MHz DDR4 |
| GPU | PowerColor Red Devil Radeon™ RX 590 8GB GDDR5 |
| Storage | WD Blue SN550 NVMe™ SSD M.2 2280 (Read – 2400MB/s, Write – 1950MB/s) |
| KALI | |
| Platform Role | Virtual Machine |
| Operating System | Kali Linux 2023.1 |
| Processors | 2 |
| Memory | 2GB |
| Video Memory | 128MB |

Table - Testing Environment Details

### 3.5.4. Testing Methodology

Results were taken from OF (v.1.85-beta) and its refactored version, Foremost (v. 1.5.7), and RecoverMyFiles (v. 6.4.2.2590). This allowed for a direct comparison against the results gathered from OF’s original release which were also taken against the same software. The latest versions of the software had to be used as the versions utilized in the original testing were no longer available.

Following the process defined in the original released paper for OpenForensics, a digital disk image was used to demonstrate the capabilities of the refactored program. The tests were retaken on a readily available computer system matching the testing logic as closely as possible – comparing OF’s performance to that of equivalent tools.

Multiple DD files were used as targets to best demonstrate the capability of the refactored program. The first DD image used was Mikus’ “Basic Data Carving Test #1”, a small and relatively straightforward disk image with very little in the way of any complex structures used to test the basic functionality of the program. This file also allowed for a comparison of processing times and accuracy within smaller DD files which don’t occur very often in realistic settings but can give insight into the impact of target DD file sizes to performance. Its contents are listed below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Num** | **Name** | **MD5 Hash** | **Size (bytes)** |
| 1 | 2003\_document.doc | e72f388b36f9370f19696b164c308482 | 19968 |
| 2 | enterprise.wav | 7629b89adade055f6783dc1773274215 | 318895 |
| 3 | haxor2.jpg | 84e1dceac2eb127fef5bfdcb0eae324b | 24367 |
| 4 | holly.xls | 7917baf0219645afef8b381570c41211 | 23040 |
| 5 | lin\_1.2.pdf | e026ec863410725ba1f5765a1874800d | 1399508 |
| 6 | nlin\_14.pdf | 5b3e806e8c9c06a475cd45bf821af709 | 122434 |
| 7 | paul.jpg | 37a49f97ed279832cd4f7bd002c826a2 | 29885 |
| 8 | pumpkin.jpg | 6c9859e5121ff54d5d6298f65f0bf3b3 | 444314 |
| 9 | shark.jpg | d83428b8742a075b57b0dc424cd297c4 | 99298 |
| 10 | sm1.gif | d25fb845e6a41395adaed8bd14db7bf2 | 5498 |
| 11 | surf.mov | 5328d2b066f428ea95b2793849ab97fa | 550653 |
| 12 | surf.wmv | ff085d0c4d0e0fdc8f3427db68e26266 | 1036994 |
| 13 | test.ppt | 7b74c2c608d92f4bb76c1d3b6bd1decc | 11264 |
| 14 | wword60t.zip | c0be59d49b7ee0fdc492d2df32f2c6c6 | 78899 |
| 15 | domopers.wmv | 63c0c6986cf0a446cb54b0ac65a921a5 | 8037267 |
| Total Disk Image Size: 61.9MB | | | |

Table - Basic File Carving Test contents (Test 1)

For the second test, a USB drive was inserted into the PC and filled with a 10 ZIP files. Each of these contained a different sample file supported by OF’s file detection and carving processes. This drive was then converted into a DD file using FTK Imager’s “Create Disk Image” function and stored on the same SSD as all other tests to ensure a fair comparison between results. To keep the disk image contained into one single file, the fragmentation option in FTK imager was set to 0mb. See Table 6 for a list of the contents.

|  |  |  |
| --- | --- | --- |
| **Name** | **MD5 Hash** | **Size (bytes)** |
| 1\_jpg.zip | bbc3922fbc44688b4f4d868da1d76bfc | 6123 |
| 2\_png.zip | f02dd3e5f33147958bc7c23d966e4a02 | 2269 |
| 3\_gif.zip | 5bb84339605daef368b49963dde2663e | 137441 |
| 4\_iso.zip | 83f6bbd2ed3d4ce927098e45cf0d841f | 3391385 |
| 5\_mp4.zip | 13256abc66669b5c008186f9fa95ea80 | 1132208 |
| 6\_m4a.zip | e4d81352aace5844033401a015b6db77 | 11704808 |
| 7\_docx.zip | 3dbb8fffa0c6566af6633106b32bdf35 | 21099 |
| 8\_pdf.zip | 158fe82f501f5850a94ad7bc03449988 | 163970 |
| 9\_rar.zip | c4682c98e5ce55e08b97fc0eb4ab738c | 244 |
| 10\_xar.zip | a7b0cd4e265f3ba00d7f3561c30e9b2f | 189036 |
| Total Disk Image Size: 3.76GB | | |

Table - Numbered ZIP Test contents (Test 2)

This was split into two tests – the initial DD file examination as well as an examination of the physical USB drive itself, named “Numbered ZIP Test (Image)” and Numbered ZIP Test (Drive) accordingly. This tested the program’s capabilities in handling physical drives, allowing for a speed and accuracy comparison with the test’s DD image file counterpart while also narrowing the testing to its new file structure-based ZIP detection.

Each test was repeated three times such that the mean average could be calculated for the processing times, which OF includes as a part of its output. When testing OpenForensics, the program was run using the Release (x64) configuration on VS to mimic how the program would work once compiled and released by enabling various optimizations.

## 3.6.1. Review

The review stage of development required a thorough overlook of the progress up to that point with the help of the project supervisor, particularly demonstrating any newly added features as well as discussing next steps and gathering feedback for the coming sprint. Agile reviews typically take feedback from the users (consumers) of the developed software solution as a starting point for discussions about feature implementations and revisions, though in this case, there was no userbase to draw criticism and suggestions from. Therefore, the supervisor played the critical role of offering advice over the course of the project development cycle instead.

Reviews took place over Microsoft Teams and followed a largely informal discussive structure, though there were a number of things on the agenda that allowed for the supervisor to receive sufficient information about the accomplishments of the sprint. Some of the following points were also discussed to provide a better picture of the coming sprint as well:

Progress Review

* Demonstrating and discussing implemented features.
* Receiving feedback.
* Discussing issues and solutions.
* Reviewing progress according to Gannt chart.

Next Steps

* Determining the aims of the next sprint.
* Creating a list of possible risks for the next sprint goals.
* Ensuring the aims are feasible.
* Brainstorming ways to implement next steps.

# 4. Results

## 4.1. Completed Program

The refactored version of OpenForensics is very similar to its original release, but makes use of a new function, CalculateFileEnd, which has been placed during the header processing stage of file analysis. The function checks each processed header to see if it has a file-structure based EOF calculation available within CalculateFileEnd, and if not, the program continues to process headers as normal. If a ZIP is passed in, it uses the new semantic calculations to determine its EOF rather than the previously used FHT algorithm.

These calculations take into account the various unique characteristics of a ZIP file, moving a pointer variable dynamically through the buffer to locate them and extract their values before finding the EOF. First, the CFH is located by considering the contained file’s encryption state, compressed size, filename length, and any extra data length gathered from its LFH. Once the CFH has been found near the end of the ZIP, the program searches for 0x504B0506 to locate the EOCD, where it can extract the comment length present at the end of the EOCD header, apply it, and find the EOF. See Appendix 8.1.1. for a visual breakdown of the refactored ZIP detection algorithm.

## 4.2. Processing Speed

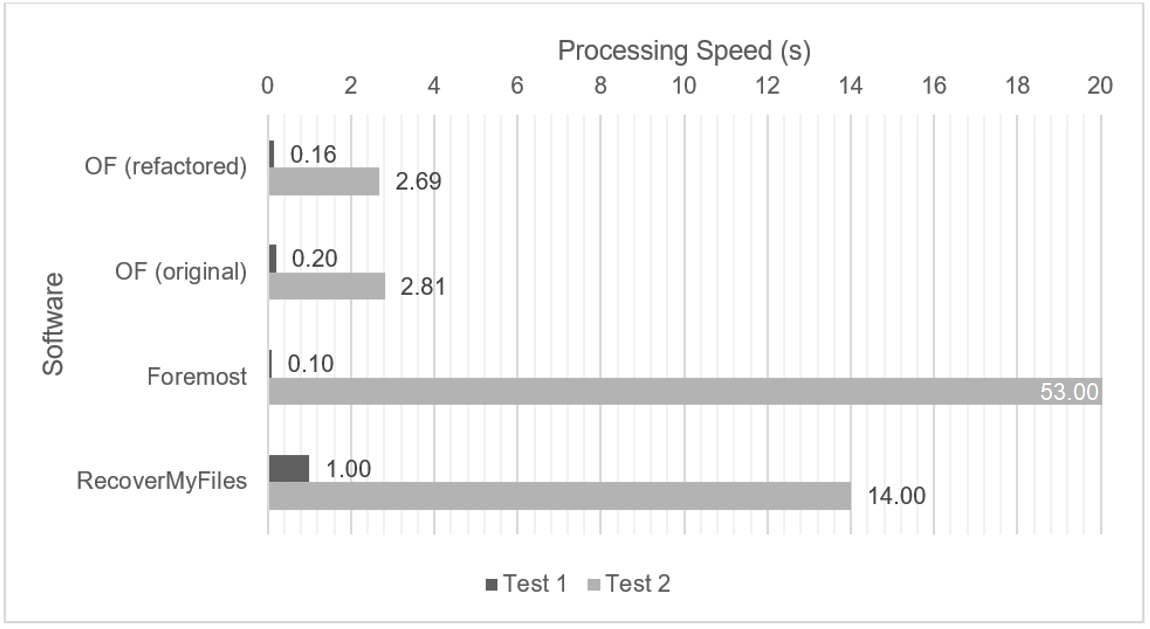


Figure – A collation of all processing speed results for Test 1 and Test 2 after calculating the mean.

## 4.3. Accuracy

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Accuracy** | **Test 1** | **Test 1 (MD5)** | **Test 2** | **Test 2 (MD5)** |
| **OF (original)** | 4/15 | 1/10 | 0/10 | 0/10 |
| **OF (refactored)** | 5/15 | 2/10 | 10/10 | 10/10 |
| **Foremost** | 13/15 | 9/10 | 9/10 | 9/10 |
| **RecoverMyFiles** | 8/15 |  | 10/10 |  |

Table - A summary of results of each software from Test 1 and Test 2

A picture containing text, screenshot, diagram, crossword puzzle

Description automatically generated

A picture containing text, screenshot, line, square

Description automatically generatedTable - A breakdown of each software's accuracy results for Test 1

Table - A breakdown of each software's accuracy results for Test 2

# 5. Discussion

The aim of this project was to refactor OpenForensics with improved file validation algorithms in order to produce more reliable results from its file carving operations, subsequently evaluating its accuracy and processing speed results by comparing them to those of its original release and various competitors. This chapter is a discussion of how successfully this aim is met, looking at the initial project requirements, methods undertaken, and analysing gathered results.

## 5.1. Original Algorithm Implementation

The first objective of the project was to investigate the techniques used in the current version of OF. To achieve this, various sources were analysed, including OF’s original documentation. Based on these findings, it was found that OF uses a two-pass File Header Trailer (FHT) algorithm to first locate all header instances before locating footers and carving the data in between, with no existing framework for semantic file analysis. From this information, it was possible to begin planning how a solution might look within the code, specifically looking at regions within the source code that relate to header analysis.

## 5.2. Identifying Solutions

The next step in refactoring OF was to choose a suitable file detection technique that could reliably analyse and aid in carving out ZIP files. A relevant and effective framework also had to be coded to allow other filetypes to be supported. Based on the results of a thorough literature review, a semantic file structure-based detection algorithm was chosen as the most effective solution as it kept to the defined project scope and did not require large amounts of code to be rewritten. Instead, the solution builds on existing code in an integrated, scalable fashion. A modified Agile sprint methodology was adopted for the project due to its scalability for teams large and small, and its strong focus on communication with clients – in this case, the project supervisor and author of OF who were communicated with throughout the development of the project.

## 5.3. Implementing a Solution

Once a solution was identified, it needed to be implemented. This was done over the course of the design and development stages of each Agile sprint with rigorous testing being done before each end-of-sprint review, which were repeated until the desired outcome was achieved. The end result is a working, reliable, refactored version of OF with an improved file detection framework and support for semantic ZIP file analysis.

## 5.4. Testing

With the development complete, it was necessary to test the program in various scenarios. Results were taken by comparing the processing speed and accuracy of OF before and after its refactoring, as well as a variety of comparable file carving tools, each of which use different methods to achieve their results. Three measurements were taken of each processing speed score such that a mean could be calculated from each program’s outcomes, ensuring the end results are fair for each included software. A basic file carving test image was obtained from the website of Foremost’s co-creator, and a larger, ZIP-focussed disk image was created to cover all possible use cases for the software.

## 5.5. Evaluation of Results

By comparing the processing time of each solution, it can be seen that the refactored version of OF is 20% faster in the Basic File Carving Test (Test 1), and 4% than the original in the Numbered ZIP Test (Test 2), showing an overall improvement in processing speeds. This can be attributed to the fact that OF no longer has to produce ZIP lookup results or search for any of the additional ZIP EOF values that were present in the original version of FileTypes.xml, saving a few milliseconds throughout the file carving process.

Foremost has the quickest processing time of all results for Test 1, but also has the longest for Test 2, going off the scale and reaching 53 seconds. Because Foremost is a single-core application, it does not have to initialise any threads like OF, which can decrease the processing time by a short amount. This results in it being able to search through smaller datasets quicker than other tested solutions. Its single-core nature is also the cause of its off-the-scale results for Test 2. Since Test 2 is a much bigger file at almost 4GB, it requires much more processing power to perform file carving on quickly. Analysing it with a single core is much slower than the time taken for OF to initialise its threads and finish its file carving operations using its much more effective GPU-based file carving algorithm.

This also impacts RecoverMyFiles, a multi-core CPU-bound application which spends the longest time of all tested software to scan through Test 1, which may be attributed to its own thread initialisation overhead; as it is a file recovery tool, RecoverMyFiles also needs to make a bigger lookup table and may find more files, taking more time to process its results. However, RecoverMyFiles’ source code is not readily available, and these are ultimately an educated guess at its inner workings based on the methods utilised in the other tested solutions.

When comparing accuracy results, defined by each solutions success in extracting files from both disk images, the refactored version of OpenForensics scored one point higher in Test 1 than the original. This can be attributed to it successfully carving a ZIP file with its MD5 hash intact, and since no other filetypes are supported by the refactored code besides ZIPs, there were no other semantic algorithms to impact the other results.

Where the original version of OpenForensics scored no successful extractions in Test 2, the refactored version scored all ten, reproducing them with their respective MD5 hashes intact, demonstrating a potential 100% success rate at extracting valid, non-fragmented ZIP files. Implementing an algorithm to account for fragmentation is a highly technical task and would require major changes to be made to the way OF detects valid data between a file’s beginning and its end. This is outside the agreed scope of semantic file detection and would have placed significant time constraints on the rest of the project.

## 5.6. Limitations

### 5.6.1. Out of Bounds Index Crash

Using files larger than 100MB would always result in a crash to desktop failure (CTD). The cause was investigated thoroughly and determined to be the result of the data types utilised in the various ZIP header structs. These were initially declared as signed integers and shorts, and therefore were prone to rolling over into negative values after reaching their maximum size. The issue was fixed by converting the relevant variables to their unsigned counterparts, uint and ushort within the struct. This allowed the use larger datasets for more accurate testing once the limitation was fixed.

### 5.6.2. ZIP Varieties

The program was refactored to detect and analyse standard ZIP files, following the structure defined in Appendix 8.1.1. ZIPs that use other header values and internal structures are therefore not recognised and not extracted successfully. The result of this is that only the ZIP files that follow the exact implemented patterns will be successfully extracted unlike how Foremost handles ZIP files.

Foremost has specific checks at the beginning of each header processing phase that allow it to determine the ZIP type and version more accurately before branching to that particular variety’s extraction algorithm. While this approach was considered, it would have required the creation of multiple other complete filetype cases, each with their own unique calculations. A method of moving between these cases would also have needed to be investigated and implemented By keeping the focus of the project entirely to standard ZIPs, the implementation was able to meet all the requirements within the given time frame where a broader one had the potential to cause time constraint issues.

### 5.6.3. Lack of Available Testing Images

At this time, there are very few documented sources of DD files specifically made to test forensic tools which impacted the testing phase of the project. Such tools require a large amount of time and effort to produce, needing to compile a realistic dataset of different files onto an image file, noting their locations, calculating each file’s MD5 hash, and testing that there are no unrealistic inconsistencies within the image file. Therefore, it was not possible to produce them as a part of this project outside of the Numbered ZIP test which remains relatively small compared to full true-to-life drive images.

### 5.6.4. Lack of C# refactoring Information

There currently also exist no sources that describe refactoring forensic tool source code, particularly in relation to C#. This left the outcomes of the planning, designing, and developing stages of the project dependant on other less relevant but still helpful sources, past experience, knowledge, and intuition. This is largely to do with a lack of standardisation within the world of digital forensics which also impacted the number of documented disk image files for testing.

### 5.6.5. Original OF Testing Setup

For testing, it was not possible to replicate the exact setup employed in the original documentation of OF because it would push the project far outside of the defined spending scope. The time taken to gather and build these testing systems would also require more time to be allocated to the project which was not possible. However, the resulting setup was built to be as close as possible to the original OF tests, using the same tools albeit using more modern versions. This ultimately had no quantifiable impact on the gathered results with each tested software working as it should, and potentially improved the quality of results gathered due to the various optimization updates that have occurred for RecoverMyFiles since the original release of OF.

### 5.6.6. RecoverMyFiles and Foremost Output

Both the RecoverMyFiles and Foremost output files lacked vital details in relation to the output files provided by OF. Information such as time taken, number of threads utilised, MD5 hash outputs, and calculated data throughput were inconsistent or not supported. Both Foremost and RecoverMyFiles fail to mention exact data throughput utilised during their calculations which made it difficult to gauge how well each of the programs make use of memory and storage.

RecoverMyFiles’ data recovery feature is locked in the free version, meaning that MD5 hashes were not able to be calculated for its output based on the scope of the project. However, the output it presents in-software accurately displays the files it found and their contents. Based on the original OF testing results, it is likely the accuracy would have been closer to that of OF than Foremost and its processing speed somewhere in the middle of the two (Appendix 8.1.2). This did not impact the testing phase too much, as the focus was how successful OF was refactored against its original version, with previous documentation indicating that it was faster than both Foremost and RecoverMyFiles at a minor cost to accuracy already.

### 5.6.7. OF Filetype Inconsistencies

While testing OF, the program would often fail to detect any headers in the target, particularly when testing the original release of OF against the Numbered ZIP test. This is likely an error with the way the program determines which files to target in the user interface, with certain filetypes being grouped into “All Files”, ”Images”, “Videos”, and “Miscellaneous” in the PopulateFileTypes function within OpenForensics.cs. A workaround was found by just selecting the target to be ZIP files, though this didn’t improve the original version of OF’s results in the Numbered ZIPs test, instead just extracting several broken MPG files. This suggested that the program just was not able to accurately detect the ZIP files present, which was expected based on given background information about OF’s handling of ZIP files.

# 6. Conclusion

In conclusion, the aim of the paper was to refactor OpenForensics with improved file validation algorithms in order to produce more reliable results from its file carving operations. Findings from throughout project showed that despite the growth of the internet, Internet of Things, and artificial intelligence, the digital forensics landscape remains stagnant, yet to re-enter its once golden age. A lack of standardisation and communication between developers only negatively impacts the field, causing a lack in innovation. Despite this, OpenForensics and its GPU-based file carving methods breathe new life into the field of forensics, providing file carving procedures capable of processing data faster than ever before. The tool shows great promise in overtaking its competitors with its GPU-based processing and new scalable semantic file-carving algorithm.

The final artefact of the project is an improved version of OpenForensics, utilising a tried-and-tested implementation of a file structure-based detection algorithm. The results gathered during testing demonstrate the effectiveness of an appropriately applied semantic file carving algorithm, with the most relevant results coming from comparisons between OpenForensics’ original release and its refactored version. Though the resulting implementation is only limited to one filetype and requires further work, particularly in regards to optimisation and filetype support, long-term support from collaborators and the rise of modern competition can potentially re-launch digital forensics into a golden age.

The research question upon which this paper is based was “How can the accuracy of open-source digital forensic software be affected through the use of semantic file carving algorithms?” and the answer lies entirely in the results of various tests and comparisons performed. A well implemented, thorough, and modular semantic file carving algorithm ultimately has the potential to greatly improve the results of a piece of open-source digital forensics software. Naturally, this is greatly dependant on the methodology undertaken, the available tools and skills, and will vary between each project. As more passion projects begin to appear aiming to overhaul existing methods to keep up with the ever-accelerating development of modern information technology, so too will standardised methodologies, test kits, and sample data.

## 6.1. Future Work

If more time was able to be allocated to the project, there are a number of ways the research could be improved. Firstly, more filetypes supported by the semantic file detection algorithm would greatly benefit the results of the project, offering insight into how the more complex or more simple file structures are reflected in the processing speed results. Secondly, a much larger test image would be created and documented, not only to aid the demonstration of processing speeds within this project, but also to create a much lacking test kit to aid in the development of future forensics tools. This image would note the various file sizes, filenames, types, and MD5 hashes of each carefully included file. Finally, more file carving tools would be collected and compared in the results to demonstrate the variance in accuracy and processing speeds between the tools available today, or how far forensics tools have gotten since the advent of file analysis.

As an early start on future work, various tests and alterations have already been performed on the code available open source on GitHub (Bayne, 2023). The main differences in the latest branches include optimizations to ensure the GPU returns and processes all semantic file detection instances improving its processing speed and memory allocation, with new functionality to validate found ZIPs, preventing them from incorrectly extending in size to multiple Gigabytes.

# 7. References

Aho, A. V. & Corasick, M. J., 1975. Efficient string matching: an aid to bibliographic search. *Communications of the ACM,* 18(6), pp. 333-340.

Arudchutha, S., Nishanthy, T. & Ragel, R. G., 2013. *String Matching with Multicore CPUs: Performing Better with the Aho Corasick Algorithm.* Peradeniya, Sri Lanka, IEEE, pp. 231-236.

Barker Software, n.d.. *Corrupt JPEG Checker GUI for Windows XP, 7 & 8.* [Online]   
Available at: https://www.barkersoftware.com/support/corrupt-jpeg-checker-gui-for-windows-xp-7-8/  
[Accessed 15 May 2023].

Bayne, E., 2023. *OpenForensics (filetype-enhancements).* [Online]   
Available at: https://github.com/ethanbayne/OpenForensics/tree/filetype-enhancements  
[Accessed 16 May 2023].

Bayne, E., Ferguson, I. & Sampson, A., 2018. OpenForensics: A digital forensics GPU pattern matching approach for the 21st century. *Digital Investigation,* 24(Supplement), pp. S29-S37.

Beek, C., 2017. *Introduction to File Carving,* Santa Clara, CA: McAfee.

Boyer, R. S. & Moore, J. S., 1977. A fast string searching algorithm. *Communications of the ACM,* 20(10), pp. 762-772.

Cohen, M., 2007. Advanced carving techniques. *Digital Investigation Issues 3–4,* Volume 4(3-4), pp. 119-128.

Garfinkel, S., 2010. Digital Forensics Research: The Next 10 Years. *Digital Forensics 7,* Volume 7, pp. 64-73.

Garfinkel, S. L., 2007. Carving contiguous and fragmented files with fast object validation. *Digital Investigation,* Volume 4, pp. 2-12.

HMICFRS, 2022. *Digital Forensics: An inspection into how well the police and other agencies use digital forensics in their investigations,* London: HMICFRS.

Kendall, K. & Mikus, N., n.d. *Foremost.* [Online]   
Available at: https://foremost.sourceforge.net/  
[Accessed 20 April 2023].

Laurenson, T., 2013. *Performance Analysis of File Carving Tools.* Berlin, Springer Berlin Heidelberg, pp. 419-433.

Lin, C.-H., Liu, C.-H., Chien, L.-S. & Chang, S.-C., 2013. Accelerating Pattern Matching Using a Novel Parallel Algorithm on GPUs. *IEEE Transactions on Computers,* 62(10), pp. 1906-1916.

Marziale, L., Richard III, G. G. & Roussev, V., 2007. Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools. *Digital Investigation 4,* pp. 73-81.

Mikus, N., 2005. *An Analysis of Disc Carving Techniques,* Monterey, California: Naval Postgraduate School Press.

Montasari, R. & Hill, R., 2019. *Next-Generation Digital Forensics: Challenges and Future Paradigms.* London, IEEE, pp. 205-212.

Richard III, G. G. & Roussev, V., 2005. Scalpel - A Frugal, High Performance File Carver. *Digital Forensics Research Conference (DFRWS)*, 1 January, pp. 71-77.

Richard III, G. G., Roussev, V. & Marziale, V., 2007. *In-Place File Carving.* Boston, Springer, pp. 217-230.

Scanlon, M., 2016. *Battling the digital forensic backlog through data deduplication.* Dublin, IEEE, pp. 10-14.

Sencar, M., 2009. : Identification and Recovery of JPEG Files with Missing. *Digital Investigation 6(suppl.),* pp. 88-98.

Skrbina, N. & Stojanovski, T., 2012. *Using parallel processing for file carving.* Bitola, Republic of Macedonia, CIIT, pp. 175-179.

StatCounter, 2023. *Operating System Market Share Worldwide.* [Online]   
Available at: https://gs.statcounter.com/os-market-share  
[Accessed 15 May 2023].

Visual Paradigm, 2023. *Agile Framework Tools: From Small Teams to Scaling Agile.* [Online]   
Available at: https://www.visual-paradigm.com/scrum/from-small-teams-to-scaling-agile/  
[Accessed 15 May 2023].

Zha, X. & Sahni, S., 2010. *Fast In-Place File Carving For Digital Forensics.* Berlin, Heidelberg, Springer, pp. 141-158.

# 8. Appendices

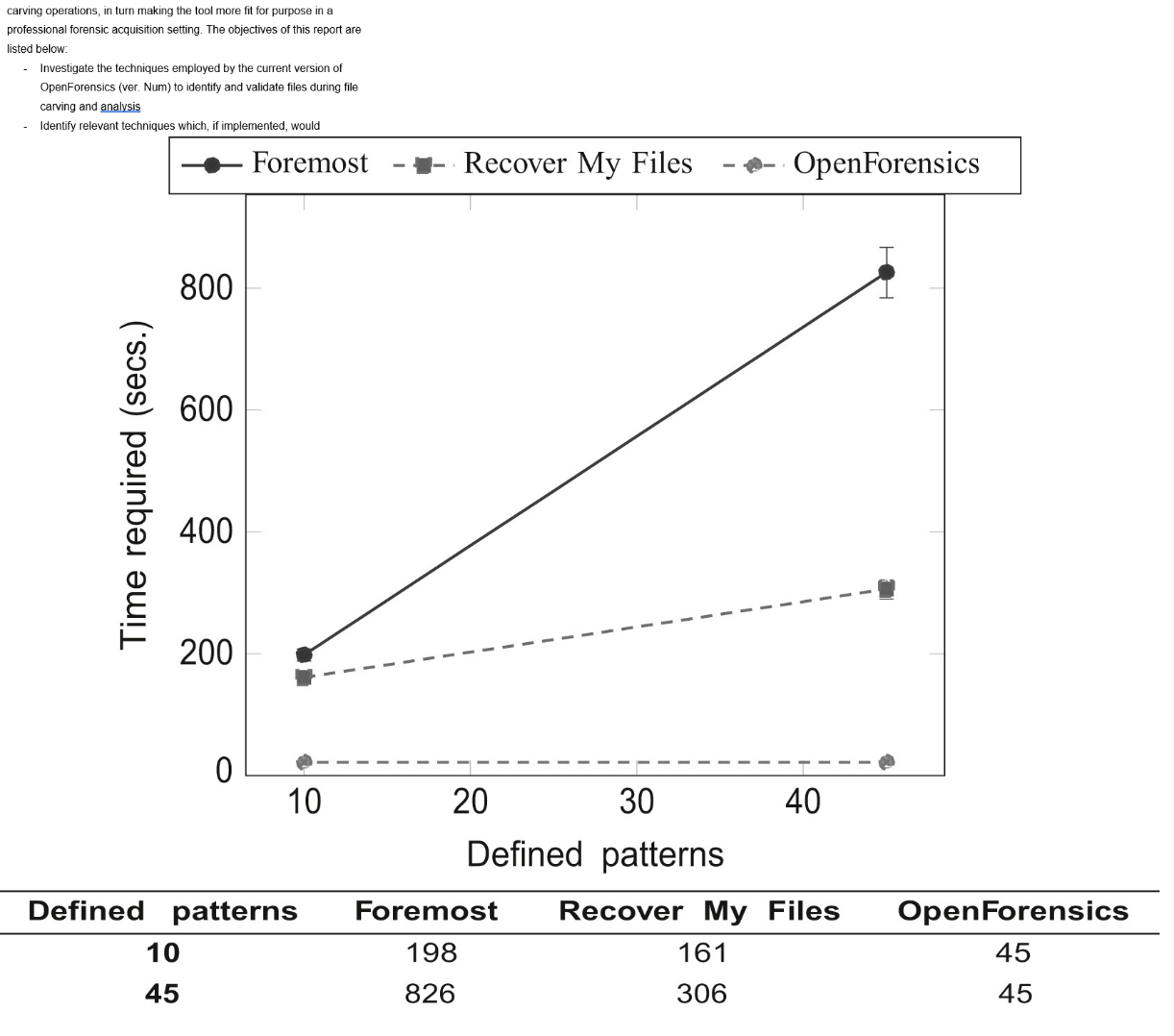
## 8.1 Diagrams

### 8.1.1. ZIP End of File Calculation Diagram

A diagram of a flowchart

Description automatically generated with low confidence

### 8.1.2. OpenForensics – Original Release Benchmark



(Bayne, et al., 2018)

## 8.2. Output

### 8.2.1. OpenForensics (Original)

#### 8.2.1.1. Test 1 (Basic File Carving)

A picture containing text, screenshot, graphic design, brand

Description automatically generated

A screenshot of a computer program

Description automatically generated with medium confidence

#### 8.2.1.2. Test 2 (Numbered ZIP)

A screenshot of a video

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

### 8.2.2 OpenForensics (Refactored)

#### 8.2.2.1. Test 1 (Basic File Carving)

A screenshot of a computer program

Description automatically generated with medium confidence

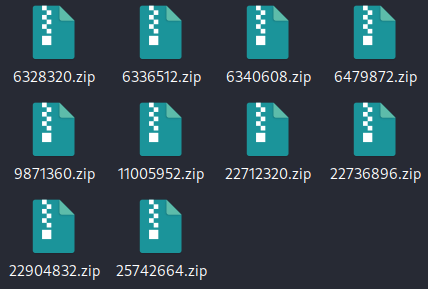
A screenshot of a computer

Description automatically generated with medium confidence

#### 8.2.2.2. Test 2 (Numbered ZIP)

A screenshot of a computer program

Description automatically generated with medium confidence



### 8.2.3. Foremost

#### 8.2.3.1. Test 1 (Basic File Carving)

A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated

#### 8.2.3.2. Test 2 (Numbered ZIPs)

A screen shot of a computer code

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

## 8.3 Code

### 8.3.1. Analysis.cs

#### 8.3.1.1. File-Specific Structs

//## Enables the storage of various section lengths

//## Used to calculate EOF in File Carving Operations > Semantic File Structure-Based Detection

//## ZIP

public struct ZipLocalFileHeader

{

// BITS IN TARGET

public uint signature; // 0

public ushort version; // 4

public ushort genFlag; // 6

public ushort compression; // 8

public ushort last\_mod\_time; // 10

public ushort last\_mod\_date; // 12

public uint crc; // 14

public uint compressed; // 18

public uint uncompressed; // 22

public ushort filename\_len; // 26

public ushort extra\_len; // 28

}

public struct ZipCentralFileHeader

//## Unused - Central FHs are skipped to improve speed

//## and thus are not processed. Mainly for reference.

{

public uint signature; // 0

public string version\_extract; // 4

public string version\_madeby; // 6

public ushort genFlag; // 8

public ushort compression; // 10

public ushort last\_mod\_time; // 12

public ushort last\_mod\_date; // 14

public uint crc; // 16

public uint compressed; // 20

public uint uncompressed; // 24

public ushort filename\_len; // 28

public ushort extra\_len; // 30

public ushort filecomment\_len; // 32

public ushort disk\_number\_start; // 34

}

public struct ZipEndCentralFileHeader

{

public uint signature; // 0

public ushort diskNum; // 4

public ushort compression; // 6

public ushort centralDirStart; // 8

public ushort centralDirEntriesNum; // 10

public uint centralDirSize; // 12

public uint offset; // 16

public ushort comment\_len; // 20

};

#### 8.3.1.2. Integration within ProcessFoundResults

private bool ProcessFoundResults(ref byte[] buffer, int threadNo, ref ulong count, ref byte[] resultID, ref int[] resultLoc)

{

int i = (threadNo \* (resultLoc.Length / procShare));

int end = ((threadNo + 1) \* (resultLoc.Length / procShare));

ulong fileEnd;

//BackgroundWorker imageGenerator = new BackgroundWorker();

while (i < end && !shouldStop)

{

// +1 to file type "traces" if header and collate resultID and resultLoc to foundRecords

if (resultID[i] % 2 != 0)

{

int fileIndex = ((resultID[i] + 1) / 2) - 1;

Interlocked.Increment(ref results[fileIndex]);

// If the file is a jpg, try and generate a thumbnail

if (imagePreview && (targetName[fileIndex] == "jpg"))// || targetName[fileIndex] == "bmp" || targetName[fileIndex] == "gif" || targetName[fileIndex] == "png" || targetName[fileIndex] == "tiff"))

{

int start = resultLoc[i];

int finish = 0;

int footerType = FindFooterID(targetName[fileIndex]);

for (int j = i; j < resultID.Length && !shouldStop; j++)

{

if (resultID[j] == footerType)

{

finish = resultLoc[j];

break;

}

}

// If file end found and file size > 300KB, then add a thumbnail from the data whilst in memory.

if (finish != 0)// && (finish - start) > 300000) // Commented out the >300KB filter for Visualisation Experiment

{

ulong fileID = (count + (ulong)start);

byte[] fileData = new byte[finish - start];

Array.Copy(buffer, start, fileData, 0, finish - start);

//addThumb(fileData, fileID.ToString());

thumbnailQueue.StartNew(async delegate

{

await addThumb(fileData, fileID.ToString());

}, TaskCreationOptions.PreferFairness).Unwrap();

}

}

//## Calculate fileEnd here (needs accesses to buffer, headerType, and header location)

fileEnd = CalculateFileEnd(targetName[fileIndex], (ulong)resultLoc[i], ref buffer);

//## Record the file's location

if (fileEnd != 0) { RecordFileLocation(fileIndex, (ulong)resultLoc[i], fileEnd, ""); }

}

foundRecords.Add(new foundRecord((ulong)resultLoc[i], resultID[i]));

i++;

}

updateFound();

return true; // Task.FromResult(true);

}

#### 8.3.1.3. CalculateFileEnd

//## New

//## "Developing File Extraction Methods for Open-Source Forensics Software in C#"

//## by Mikolaj Mroz (2003114@uad.ac.uk)

#region Semantic File Structure-based Detection

private ulong CalculateFileEnd(string fileType, ulong headerStart, ref byte[] buffer)

{

/\* ## Structure inspired by Foremost (Nicholas Mikus)

\* ## Mikus, N., 2005. An Analysis of Disc Carving Techniques, Monterey, California: Naval Postgraduate School Press

\*

\* ## ZIP structure information sources

\* ## Wikipedia, 2023. ZIP (file format). [Online] Available at: https://en.wikipedia.org/wiki/ZIP\_(file\_format) [Accessed 10 May 2023].

\* ## PKWare, 2022. APPNOTE.TXT - .ZIP File Format Specification. [Online] Available at: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT [Accessed 10 May 2023].

\*

\* ## Tested on

\* ## Mikus, N., 2005. Basic Data Carving Test #1. [Online] Available at: https://dftt.sourceforge.net/test11/index.html [Accessed 10 May 2023].

\*/

//## Reset fileEnd to 0 for reuse

ulong fileEnd = 0;

uint currentPos = (uint)headerStart; //## Set current position in buffer to the index of the header

bool foundCentralDir = false; //## Central Directory

bool foundEOCD = false; //## End of central directory record

switch (fileType)

{

//## If the header passed to the function belongs to a ZIP...

case "zip":

{

ZipLocalFileHeader localFH;

ZipCentralFileHeader centralFH;

ZipEndCentralFileHeader endFH;

while (foundEOCD == false) //## Perform loop until End of Central Directory Record is found (ZIP footer)

{

while (foundCentralDir == false) //## Perform loop until Central File Directory is found

{

//## Validate local file header is correct

if ((buffer[currentPos + 2]) != 0x03 && (buffer[currentPos + 3]) != 0x04) { break; }

// locations 3 and 4 should be 0x0304 (LocalFileHeader). If not, it is not valid.

else

{

//## ---------------> LOCAL FILE HEADER <---------------

//MessageBox.Show("ZIP HEADER FOUND! Header = " + (int)(buffer[currentPos]) + (int)(buffer[currentPos + 1]) + (int)(buffer[currentPos + 2]) + (int)(buffer[currentPos + 3]) + " (should be 807534) relative pos: " + (currentPos - (int)headerStart) + (" should be 0"));

//## move to location 14 to check if files are encrypted (affects descriptor length)

currentPos += 14;

localFH.crc = (uint)BitConverter.ToInt32(buffer, (int)currentPos);

//## move to location 18 to store file length

currentPos += 4;

localFH.compressed = (uint)BitConverter.ToInt32(buffer, (int)currentPos);

//MessageBox.Show("compressed file length = " + localFH.compressed);

//## move to location 26 to store filename length

currentPos += 8;

localFH.filename\_len = (ushort)BitConverter.ToInt16(buffer, (int)currentPos);

//MessageBox.Show("filename\_len = " + localFH.filename\_len);

//## move to location 28 to store extra field length

currentPos += 2;

localFH.extra\_len = (ushort)BitConverter.ToInt16(buffer, (int)currentPos);

//MessageBox.Show("extra\_len = " + localFH.extra\_len);

//## move to location 30 and apply filename length (n)

currentPos += 2;

currentPos += localFH.filename\_len;

//MessageBox.Show("relative pos after filename\_len: " + (currentPos - (int)headerStart));

//## move to location 30+n to apply extra length (m)

//## final location in LocalFileHeader is 30+n+m

currentPos += localFH.extra\_len;

//MessageBox.Show("relative pos after extra\_len: " + (currentPos - (int)headerStart));

//## move forward compressed filesize amount to enter data descriptor

currentPos += localFH.compressed;

//MessageBox.Show("relative pos after compressed: " + (currentPos - (int)headerStart));

//## ---------------> DATA DESCRIPTOR and CENTRAL FILE HEADER <---------------

//## move past the descriptor. 12 or 20 shifts if crc is present, 8 or 12 if not. depends on the ZIP type.

//## it's safer to just use a for loop to locate the CFH, the length to search can vary from 0 to 20 bytes.

uint shiftVal;

for (shiftVal = 0; shiftVal < 20; shiftVal++)

{

if ((buffer[currentPos + 2]) == 0x01 && (buffer[currentPos + 3]) == 0x02)

{

//MessageBox.Show("CENTRAL FH FOUND! \n shift value: " + (shiftVal) + ". header = " + (int)(buffer[currentPos]) + (int)(buffer[currentPos + 1]) + (int)(buffer[currentPos + 2]) + (int)(buffer[currentPos + 3]));

foundCentralDir = true;

break;

}

}

currentPos = currentPos += shiftVal;

}

continue;

}

//## ---------------> End of Central Directory Record <---------------

//MessageBox.Show("in EOCD record calculation now!");

for (; currentPos < currentPos + chunkSize || foundEOCD == true; currentPos++)

{

if ((buffer[currentPos + 2]) == 0x05 && (buffer[currentPos + 3]) == 0x06) // should be 0x50 0x4B 0x05 0x06 (EOCD)

{

endFH.centralDirStart = (ushort)currentPos;

foundEOCD = true;

//MessageBox.Show("found EOCD! " + (int)(buffer[currentPos]) + (int)(buffer[currentPos + 1]) + (int)(buffer[currentPos + 2]) + (int)(buffer[currentPos + 3]));

break;

}

}

//## move to location 20 in EOCD to store comment length

currentPos += 20;

endFH.comment\_len = (ushort)BitConverter.ToInt16(buffer, (int)currentPos);

//MessageBox.Show("comment length = " + endFH.comment\_len);

//## move to location 22 in EOCD to add comment length

currentPos += 2;

currentPos += endFH.comment\_len;

//## locate true End of File (end of comment)

fileEnd = (ulong)currentPos;

}

break;

}

}

return fileEnd; //## Return EOF to store as a result.

}

#endregion

#### 8.3.1.4. Implementation within ProcessLocations

if (foundRecords[j].patternID == footerType && targetName[fileIndex] != "zip") //## Showcasing semantic file detection on ZIPs in ProcessFoundResults

{ //## therefore we want to exclude it from the orignal EOF calculations

fileEnd = foundRecords[j].location + (ulong)targetEnd[fileIndex].Length;

fileEnd = footerAdjust(fileEnd, targetName[fileIndex]);

RecordFileLocation(fileIndex, foundRecords[i].location, fileEnd, "");

break;

}

### 8.3.2. Changes to FileTypes.xml

<Format>

<Type>Misc</Type>

<Name>zip</Name>

<MaxLengthMB>10</MaxLengthMB>

<!--##<EOF>504B</EOF>-->

<Value>504B0304</Value>

<!--##<Value>504B0506</Value>-->

<!--##<Value>504B0708</Value>-->

<!--##<Value>504B030414000100630000000000</Value>-->

</Format>