CSEC 793 Capstone in Computing Security
Project Report

CARVING OF FRAGMENTED DOCX FILES

April 9, 2021

Connor Leavesley
Department of Computing Security
College of Computing and Information Sciences
Rochester Institute of Technology

1 Abstract

Fragmented file carving is the process of extracting discontinuous files at the data-layer from a disk image without file system assistance. Fragmented multimedia carving has been the primary area of study in current research, with little attention being paid to text-based files. This paper proposed a method of carving fragmented DOCX files through by individual carving of internal files in the DOCX specification, verifying their compressed XML data, and reassembling the pieces. It covers the algorithms for carving and reassembling these files in-depth and a custom implementation uses those algorithms for test. This paper uses the implementation to test the effectiveness of the algorithms and the carver in terms of run-time, CPU usage, and RAM usage on different number of fragments and spaces between fragments. These tests found that the carver was highly effective on carving forensically important fragmented files rapidly.

2 Introduction

File carving is the process of extracting files from a disk or drive without relying on file system assistance and metadata, an ability that has proven useful in forensics data collection. This technique has been useful in cases where the file system metadata became corrupted, or files had been deleted and the index node overwritten, but the file system had yet to overwrite the data blocks. This ability to gather deleted and corrupted digital evidence makes file carving an important ability in the tool belt of forensics analysts. There are two overarching carving types: contiguous and non-contiguous. Contiguous file carving, the process of extracting a file stored in one contiguous chuck on disk, is a solved problem. Tools like Foremost [1] and Scalpel [2] have implemented the needed file system intricacies to perform this task. Fragmented file carving, the process of carving fragmented files from disk and reassembling them, continues to be an open area of research.

The ability to carve multimedia images has increased the amount of digital evidence available to forensics investigators. Multimedia fragmented file carving has been the primary area of fragmented carving due to these ease of determining how the fragments connect, but researchers have largely ignored text based documents due to the difficulty in creating such a carver. Firstly, these file carvers must be capable of finding fragmented data and classifying those fragments into specific file types. Some of these file types are similar, making them hard to differentiate. Other files can blend in with the random data of the file system, making them hard to pick out from the random noise. Even with the ability to find file fragments and classifying them correctly, the fragmented file carver still needs the ability to reassemble these fragments, a potentially long and computation heavy process. All these challenges make a document fragmented file carver difficult to develop.

There is a clear lack of document-based fragmented file carver research in academia, with no source material being available on a practical implementation. This paper improves on the current research into fragmented files carving in two ways. First, this paper developed

a new open source file carver that supports contiguous file carving to replace the existing open source file carvers that have not been updated in at least eight years, with Scalpel getting its last update in 2014 [2]. This allows for the use of advances in coding techniques and the ability to use modern computing technology to its full potential. Second, this paper proposes and implements a method for carving fragmented DOCX files. Microsoft Word is the most widely used word processing program on the market, making Word files of extreme forensic importance. The ability to carve these fragmented files from the disk will be of great help to forensics investigators.

The outline of the rest of the paper is as follows: section 3 provides the required background for fragmented file carving, section 4 presents related work, section 5 introduces the methodology, including design and implementation, and section 6 presents the results from testing the implementation.

3 Background

Early file carving relied on structure-based carving, where the carver used the unique internal structure of files to carve documents [3]. These file carvers carved contiguous sectors, from the header, or magic bytes, at the beginning of the file all the way to the footer at the end. These headers and footers uniquely identify file types. Some files, such as ZIP files, also contain internal headers to signify the start of other internal data structures [4]. While many file types have footers, this is not guaranteed. Sometimes a carver may need to use metadata from the header of a file that defines the length of the file. This parameter would allow a file carver to carve the length of the file without needing a footer. Another form of file carving is "content-based carving" [5]. This carving method uses data content in a fragment to carve files. A fragment is a number of sectors, with each sector traditionally being about 512 bytes long. An example of a fragmented file can be seen in Figure 1. This method requires a classification function to determine what data corresponds to what type of file [5].

Figure 1: Fragmented File

While carvers can carve most files as one, contiguous fragment, there is a small amount of files that are split into several fragments on disk. Carving fragmented files requires a proximity function and a validation function. The proximity function determines a

fragment's closeness to another fragment [5]. At its simplest, a proximity function uses a distance measurement to predict whether a fragment is the next fragment in a file data stream. This could take the form of simple algorithms such as Euclidean or Manhattan distances, or it could also be expressed in more advanced ways, such as measuring the difference between the entropy of two fragments. The proximity function does not need to use distance either. Using fragment contents from text-based documents, formed words or phrases can determine fragment proximity. The validation function ensures that fragments are in the correct order. One easy way to automatically perform validation would be by using checksums from the file header itself to verify the file. In lieu of checksums, examining the file content, either automatically or manually, could be used to validate the file. Implementation of automatic content validation depends on the file type. If dealing with an image, the implementation would have to decompress the image. If there were no errors in decompression, then the file would be valid. If dealing with text-based files, validation could be done through decompression, depending on the exact file type, or through text analysis. Fragmented file carving are further split into two subcategories: bi-fragments and n-fragments. Bi-fragmented files are files split into two fragments. These were the focus on Garfinkel's paper on object validation [5]. Files with more than two fragments, n-fragment, are far more difficult to carve due to the computational complexity.

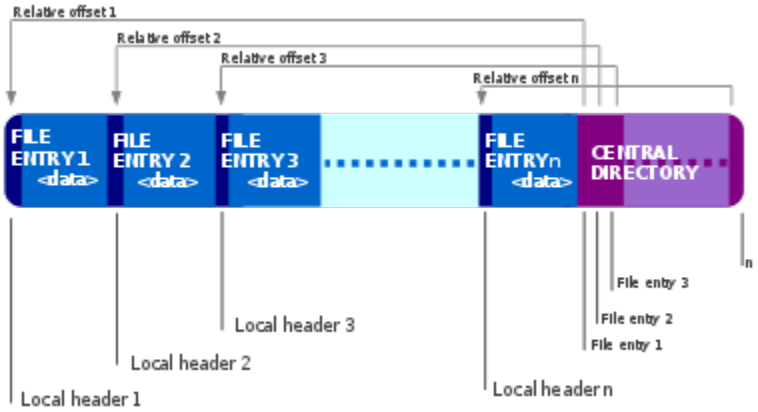

Figure 2: ZIP File Structure [6]

DOCX files are ZIP files containing XML and RELS files compressed using the Deflate compression algorithm which uses Huffman coding to compress data. The internal files follow a specific, defined naming scheme, meaning a file can be immediately attributed to a DOCX file based on name. As can be seen in Figure 2, ZIP files are split into local file headers and central directory records, with one present for each compressed file. The local file header marks the start of the compressed file. This structure stores the checksum (CRC), compressed and uncompressed size, filename, and extra field length, among other

pieces of data. Immediately after the local file header is the compressed data, which is the compressed size long. The local file header may also be followed by a data descriptor structure, which, when using ZIP64, stores the CRC and compressed and uncompressed size of the file. Since DOCX uses the Deflate compression algorithm, the start of the compressed data is the Huffman coding tree, followed by the compressed data.

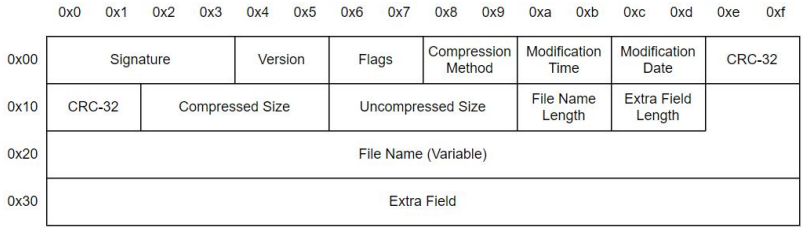

Figure 3: Local File Header Structure

After the local file headers are the central directory headers. These headers store back ups of the data in the local file headers as well has additional file attributes and a relative offset from the local file header for structure purposes.

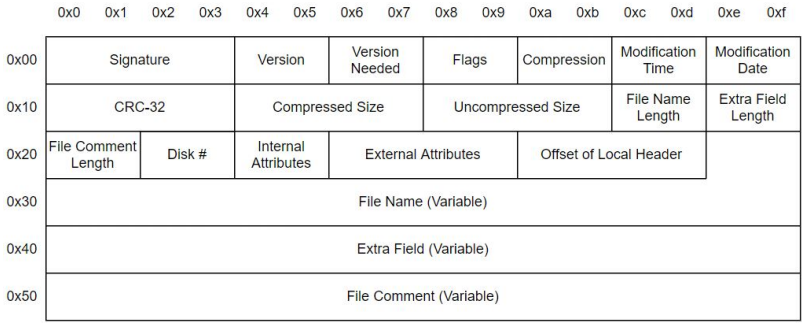

Figure 4: Local Central Directory Header Structure

4 Related Work

File fragmentation, while not common, is more likely to happen to files of forensic interest due to their typically larger size. Simon Garfinkel examined 324 drives collected from eBay that used NTFS, FAT, or UFS. Of those drives, roughly one-third were one to ten percent fragmented and only about a half had no fragmentation whatsoever. The entire set had an average of six percent fragmentation. Large files pulled from the examined disks were far more likely to be fragmented. For example, DOC files had an average file size in bytes of 85,358 and were fragmented 17% of the time, but a PNG with an average file size of 13,813

bytes was only fragmented 5% of the time. Microsoft Outlook PST files, a multi-megabyte file, fragmented 58% of the time. These fragmented files also had relatively small gaps between them, one to eight 512 byte sectors, which Garfinkel believed corresponded to the size of a cluster in the original file system on the drive [5]. The fragmentation statistics laid out by Garfinkel became a core reason for continued research into fragmented file carving, leading to major inroads for theoretical fragmented file carver design and practical JPEG carving. While fragmentation overall is not common, the need for ways to carve files these files is clearly needed.

Garfinkel [5] also laid out a basic system for carving fragmented files via a process of fragment selection and validation called "object validation". Object validation is the process of validating specific objects or files within another file rather than the whole file itself. Traditional contiguous file carving relied on validating header and footer locations, as well as structures within the file. This is a very fast process as it is just a static byte comparison [5]. This process works very well for contiguous files, however when dealing with fragmented files it can be difficult to find what sector a fragment ends and where the next begins. When working on carving fragmented JPEG files, Garfinkel [5] initially tried decompressing each sector with the JPEG decompressor. If the decompressor had errors, the sector was not part of the JPEG. The JPEG decompressor would go through several sectors of data before it would throw an error. Decompressing the JPEG to validate the end of the fragment proved to be a valid, albeit computation heavy tactic. This method was used to great effect for JPEGs, but would not work for uncompressed files. To counter this, Garfinkel [5] proposed a method called "semantic validation". Semantic validation finds fragment boundaries by comparing text between fragments. It worked very well at identifying fragments, but requires manual tuning, a long and tenuous process. With these validation methods in place, Garfinkel created a simple bifragment carving algorithm. To carve a bifragment, a carver needs to find the header and footer of a file, and then shrink the gap between the two fragment until one of the validation methods return that it is a valid file [5]. Pal and Memon [3] discussed several additional approaches for fragmented file carving algorithms centered around graph theoretic methods. Graph theoretic carvers use graph theory to reassemble fragmented files. One approach using this method was viewing fragment reconstruction as a Hamiltonian path problem. A proximity function was used to assign weights to edges between fragments, reducing the needed permutations to find the correct fragment order. To calculate proximity, a technique called Prediction by Partial Match was used. This technique uses previous data to try to predict the next characters in a sequence. This was an excellent method for structured data, but struggled on seemingly randomized data such as compressed images. But, by viewing fragment reassembly as a Hamiltonian path, other files' fragments are also considered, slowing down file recovery. To remedy this, recovery was refocused as a K-Vertex Disjoint Path problem. Instead of focusing on one file at a time, multiple files were reassembled as disjoint paths. Again, weights are assigned to edges between vertices via a proximity function. A third method examined was called Parallel Unique Path (PUP), a modified Dijkstra's algorithm. This

algorithm allowed a carver to carve multiple files at once. By removing sectors as they are added to a specific file, the algorithm can guarantee each file is a unique path. While not as accurate as more traditional methods such as Shortest Path First, it was far faster and had better scalability. Using the research into graph theoretic carvers, Pal and Memon [3] proposed SmartCarver, a file carver capable of carving files that were fragmented into more than two pieces. SmartCarver carves fragmented files in three stages: preprocessing, collating, and reassembly.

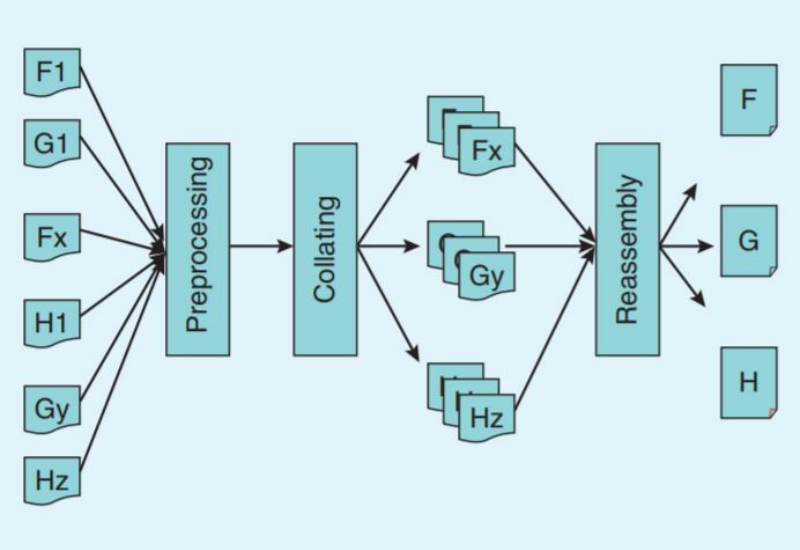

Figure 5: SmartCarver Process [7]

In the preprocessing phase, drives were decrypted and decompressed and known irrelevant clusters were removed. Next, in collation, each of the remaining clusters was classified according to their file type. This classification could be done through pattern matching, character frequency, entropy, or file fingerprints. File fingerprints is of particular interest, as it should not need to be tuned to the data set [3]. Byte frequency methods create a standard deviation of byte frequency between fragments, allowing for weights to be associated to edges. Karresand and Shahmehri [8] demonstrated Oscar, a byte frequency algorithm, to find JPEG fragments in RAM, proving the algorithms practicality. In the final phase of the SmartCarver, reassembly, files are reassembled by finding the fragmentation points of each file. Finding the fragmentation point can be done by using file structure or through a modified PUP algorithm [3]. Several approaches have been theorized and tested for carving fragmented files, but these method were only really tested and used on multimedia files. These approaches were created as file agnostic, generalized algorithms, yet they have not yet met their full potential.

Much of the current research into carving fragmented JPEG and other multimedia images

started with the Forensics Wiki and its "Carver 2.0 Planning Page" [9]. This planning page laid out the future of fragmented carving and led to the creation of an early iteration of a multimedia fragmented file carving framework [10] using the methods that Garfinkel laid out in his work on fast object validation [5]. Poisel researched ways to make a multimedia file carver in "Advanced File Carving Approaches for Multimedia Files" [11]. He splits determining file types into two different models: type-all recognition models and type-x recognition models. Type-all models are capable of classifying blocks into predefined file types, while type-x models can classify fragments of specific file types and allow for more adjustments than type-all models. As such, a type-x model was preferred for carving fragmented files. When reassembling an image, fragments must first be sorted into groups. Since there is no way to be 100% certain that a fragment is of a specific file type, a "block-gap" was introduced to combine fragments that are close to one another into larger fragments. Fragments are split where the header begins. Next the reassembly order must be found. The easiest implementation is brute force of the fragments, which is computationally intensive. The use of the previously discussed PUP algorithm worked far better. There have been several methods proposed to calculate the weights between image fragments. Image resolution would use the height and width of frames in a fragment to compare fragments. Another would be using intersections of pixels at the boundaries of fragments to compare their color value [11]. Pixel similarity was used to great effect in "Image Fragment Carving Algorithms Based on Pixel Similarity" [12]. To calculate the pixel similarity, the adjacent correlation factor is calculated between RGB vectors on the foot of the first fragment and the head of the second fragment. The sum of the adjacent correlation factors is then calculated to determine the distance between two fragments. This method was 8.5% more accurate than the method describing image reconstruction as a k-vertex path problem proposed by Pal and Memon [3][13]. Fragmented file carving has become very specialized for multimedia images because their fragments are easy to find and compare. The processes needed for carving these files have become very efficient and continue to be studied. But there are other files of forensic interest that can still not be carved.

To attempt to get past the issues with more conventional fragment classification functions, Memon et al. [14] attempted to used convolutional neural networks to identify fragment file types. Their solution, FiFTy, resulted in a fragment identification accuracy of 77.5% and was capable of processing one gigabyte of data every 38 seconds. This machine learning solution was a response to Sceadan [15], which had an fragment identification accuracy of 69% while taking 9 minutes per gigabyte. In a third machine learning method, Wang et al. used n-grams to determine file fragment type. This method resulted in, at best, a classification accuracy of 61.31% and an F_1 score of 60.71%, far worse than the neural networks [16]. Due to the wide array of file types and data in file fragment, machine learning solutions would still need to be specialized to specific files. In addition, finding enough of the correct features in the wild could prove to be challenging for continued research in this area.

Many algorithms and approaches have been theorized for carving fragmented files, but

they have only been applied to a small subset of forensically interesting files, namely, multimedia files. Multimedia files are a easy fragmented file type to carve, making them a prime target to test these methods and more specialized algorithms. But since these methods have only been used on JPEGs and other widely used image formats, there is little on how to apply them to other practical settings.

5 Methodology

This section consists of two major sections: the algorithms and implementation of the contiguous and fragmented file carver and the testing methodology. The algorithms laid out were specifically designed to use the internal DOCX structure to carve and build context between DOCX fragments. This context was imperative to reassemble the files. The file carver implementation was designed with modern programming standards in mind, compared to the currently used open source carvers that were released over eight years ago [1][2]. The testing methodology was designed to show the effectiveness of the algorithms in a theoretical testing environment.

5.1 General Design

Proper design and planning were paramount to ensure that a fragmented file carver utilizes system resources effectively. The amount of data in digital forensics investigations continues to increase, increasing computational time needed to carve contiguous files. While contiguous file carving is fast process, larger disks take a longer time to scan and carve. In addition, fragmented file carving requires additional calculations to discover and build context between fragments, increasing the overall time complexity. Unlike contiguous file carving, fragmented file carving can require multiple passes over the disk to develop the context needed to connect fragments, greatly increasing run time. To best utilize system resources, the implementation laid out in this paper was written in Golang. Golang's multi-threading capabilities and efficient memory and IO management made it ideal for fast disk operations and bulk fragment calculations.

To promote modularity, code cleanliness, and reduce code reuse, the carver implementation treated each file type as a separate module, and each module file holds the required logic for both contiguous and fragmented carving of that specific file type. A main module file housed common logic between all the modules, as well as two function jump tables. The first jump table was for contiguous file extraction, while the second jump table was for fragmented file collection. Because of these design choices, all a developer needs to do to add new functionality is add a new file with the required functionality and add the extract functions to the jump table. Developers will not need to be familiar with the other modules' functionality or the file carver internals, only the data that needs to be passed between the module and the internal carver.

5.2 Contiguous File Carver Design

Contiguous file carver development, while a very different entity, defined many of the early features of the fragmented carver. With the file data loaded into the file carver, the carver then proceeds to open the input image. For each 512 byte fragment in the input image, the carver performs a search of each file types' header to determine if a file begins at that location. As it searches, the carver actively switches between little and big endian on each fragment to determine the endianness of the image. Finding a header determines the overall endianness of the system. If nothing is found in the fragment, the reader moves onto the next fragment in the image, but if a header is found, the carver enters carving mode. In carving mode, the carver calls the module associated with the specific file via a function jump table in the main module Go file. The start fragment position and the name of the image is then passed onto the carving function. Due to the size of disk images, sections of the disk are loaded into memory one at a time in a method called a "sliding window". Foremost, for instance, has a default sliding window of 100 megabytes [1]. This paper's implementation has elected for a far smaller sliding windows of 512 bytes, the smallest block size possible. The purpose for this serves several purposes. One, the memory usage is significantly reduced in favor of more disk IO. In terms of a contiguous file carver, this is design choice may not make much sense. A larger window means less IO operations, meaning faster scanning of disk space when carving files. However, when dealing with a fragmented file carver, the computation required when comparing fragments is a compute intensive, blocking task, not an IO block. Secondly, there is little indication of gap locations between fragments. Going through images one sector at a time ensures that no extra data is gathered that isn't related to the file type being carved. To keep the data being given to the contiguous and fragmented file carvers the same, both carvers used a window of 512-bytes. For contiguous file carving, each fragment in the sliding window is append onto the header fragment until the a max possible size or the carver finds the footer. If the carver reaches a max size, the file is either corrupted or fragmented, but if the footer is found the collected fragments would be outputted to a file, creating a carved file.

5.3 Fragmented File Carver Design

Document based fragmented file carver development suffers from a number of issues, namely time complexity and context building. Building context and associating fragments with each other is an inherently compute intensive process. DOCX files are ZIP files containing XML and RELS files compressed using the Deflate compression algorithm which uses Huffman coding to compress data. It is not possible to look at the data and build context between fragments without attempting to decompress the data first. A DOCX fragmented file carver using context based carving would have to append fragments to one another and decompress the data before it could be confirmed that the fragment was part of the file. And even with that decompressed data, the carver would required knowledge about the internal XML

structure and would have to have a way to handle almost every possible case. These cases must be capable of telling valid XML subsections from randomized data, as well as ensuring those XML subsections hold data that makes sense. The constant validation makes for an extremely time and computational intensive task, as well as an extremely long development period.

While the time complexity and development issues have not been completely solved, the implementation proposed by this paper has made some inroads. This paper uses one major assumption to cut down on carving and development time; it reduces the search space by the distance fragments are from one another. The odds of fragments being more than a few file system defined blocks apart is highly unlikely, as Garfinkel conjectured [5]. To reduce the search space while still leaving a large possible gap space, a valid fragment is assumed to be no more than 16,384 bytes away from the last fragment in the sequence. This means that the carver assumes fragments on the disk are sequential. To further cut down on the number of fragments, only headers associated with XML, RELS, JPG, and PNG files were kept. XML and RELS files are the only types of files used for storing DOCX data and metadata, while JPG and PNG files were associated with photos inside the file. While these photos cannot currently be carved in the implementation, it was prudent to include the check for the future. Finally, by using Golang, our implementation had access to more advanced bulk search algorithms, multi threading in the form of goroutines, and memory efficient operations that normally would have to be developed by file carver developers. To split up carving to distinct tasks, the implementation is split into three major function categories: the header and footer files collected according to ZIP's internal file structure, reconstruction of damaged header metadata, and reassembly.

5.3.1 Fragment Identification and Gathering

Marked fragment collection was the process of collecting fragments that contained headers for both the file header and central directory sections of a ZIP file and the footer. For each fragment in the image, see if it contained the magic numbers for the header, central directory, or footer. When one of these magic byte strings were found, the carver would attempt to carve out the entire header, which is thirty bytes plus the length of the file name and the length of the extra field section, if applicable. From the header and central directory, the carver extracts three values of interest: the filename, uncompressed length of the data, and the length of the extra field. DOCX files follow the same standardized structure, allowing the file name to group files together into groups representing a single file. The carver used the uncompressed length as part of the validation function. The length of the extra field was a sanity check to ensure no data was missed. The extra field is typically used to store data not defined in the ZIP specification. Interestingly enough, DOCX pads this section for certain files, namely RELS files, with zeros, ranging from two hundred to up to five hundred bytes long. At the time of writing, I am unsure of the purpose. Once a header has been carved, the carver began header validation. As this carver is solely

looking for DOCX files, it discarded other ZIP file fragments based on filename. The footer contained no information of interest. When these fragments were finally validated, the carver stored their position on disk in a list of validated fragments.

5.3.2 Damaged Header Recovery

The magic bytes of first fragment in a file "[Content_Types].xml" will always be the first four bytes in the fragment. This is not guaranteed for the other byte sequences, however. There were two special cases that needed to be accounted for: headers that were split between two fragments and magic byte sequences that were split between two fragments.

Dealing with headers with still complete magic byte sequences were by far the easier case to deal with as the carver was still capable of detecting these marked fragments and catalog them. Using the process outlined in fragment gathering, our fragmented carver first gathered file headers, central directories, and footers. For each of these found fragments the carver performed several length checks. The initial header before the file name and extra field is thirty bytes, meaning that if the space from the start of the header and the end of the fragment is thirty bytes or lower, than the header is incomplete. If the header is more than thirty bytes, the carver then verifies that the full filename is in the header, if not, the header is also incomplete. Finally the carver checks the length of the extra field. Once again, if the length of the extra field is not zero and there is not enough room left in the fragment to contain that data, then the header is incomplete. These incomplete headers are not very useful, necessitating their reconstruction before the carver can continue. For each incomplete header, the carver appended the surrounding near sectors to the end, one at a time, and then checked to see if the missing values were restored to values to made sense. In the case of the headers with no filename, this meant that there was no way to be absolutely sure that the appended fragment was the correct one, meaning a list of potentials needed to be kept for that header. In this case, any data, such as the filename length, was used to verify that there was a match, but this method regularly resulted in at least a couple of potential matching fragments. Headers with a partial or full filename were far easier to match. In the case of partial filenames, it was simply a matter of finding a fragment that had the other half of the file name. This was a "best fit", and the carver assumed that the first matching fragment found was the only fit possible. Theoretically there is a chance that, in a heavily fragmented file system, there could be multiple breaks at the same point across multiple files resulting in more than one match, which is statistically unlikely. For headers will the full filename, no searching was required due to how DOCX was filling the extra field. If the length of the extra field was more that 512 bytes, the carver would pad the header fragment with 512 bytes, else it would just move onto the next incomplete header. While not a perfect method, it resulted in all incomplete headers having at least a potential match for reconstruction.

The other special case that needed to be consider was when the incomplete header was split through the magic bytes, which the implementation called "broken headers". These

broken headers were computationally expensive to reconstruct, leading us to choose to ignore the case in our implementation. For example, the magic bytes for a ZIP file local file header are "\x50\x4b\x03\x04". A fragment ending in "\x50" could potentially be the start of another header, meaning the surrounding sectors would be searched to see if they began with "\x4b\x03\x04". The odds of a fragment ending with "\x50" is not small, resulting in false positive overabundance. A better method for the future would likely be search for incomplete magic bytes at the beginning of fragments. Since the magic bytes were broken, the second half would have to begin at the start at the next fragment, meaning the filename would be fully intact. Since the filename would be fully intact, a carver could instead use the filename and other data to verify that it is a DOCX fragment instead of relying on the magic bytes.

5.3.3 Unmarked Fragment Collection

With the headers in hand, the carver must next locate fragments containing compressed data and associate it with the correct header. Each header has two arrays: a best fit array that stores the positions of a stream of fragments that perfectly fit the header, and a potential array, which stores a dictionary based tree of potential paths to the correct fit. This dictionary uses the starting position of the fragment as they key, and an array of fragments that follow it as the value. To locate the potential fragments, the carver once again goes through the image one fragment at a time. Each fragment is passed to a trio of goroutines that deal with local file headers, central directory headers, and incomplete headers. Each of these goroutines loop through their list of headers for each fragment. In this loop, the carver first ensures that the fragment is within 512,000 bytes of the header and the header is not yet "complete". If these two conditions are met, the fragment is passed to an internal routine. This internal routine employs a modified depth first search algorithm that iterates recursively through the dictionary tree of the header to create an array containing arrays of potential paths.

If the current fragment is less than 16,384 bytes away from the end of the data stream and is not a byte array of all zero, the carver appends it to the end of each of these potential paths, creating a new data stream. The carver then attempts to decompress the data stream using the Enflate algorithm, resulting in either malformed data, XML data, or malformed XML data. Next, the enflated data is split on the "<" character and was sent through a partial XML validator. This partial XML validator goes through the resultant array except for the last item in the array. In the event of incomplete XML data after decompression, the last item in the array is almost always guaranteed to be incorrect, and is therefore not beneficial to check. The items that are checked in the array are ensured to have the correct formatting of XML data. While incomplete, here is the current list of checks:

- An item beginning with "<?" ends with "?>".
- An item beginning with "</" ends with ">".

- An item beginning with a letter ends with ">" unless that item contains "w:t", "Status", "Display", "DOCVARIABLE", or "instrText".

For the last check, if the item beings with a letter, ends with ">", and does not contain those key works, that item is passed through a regular expression to extract phrases that are setting a key equal to a quoted values, such as 'w:x="0"'. Each of these sub items is then put through the following checks:

- An item contains "xmlns" also contains "com" or "org" or four "-"'s.
- An item contains "w:t".

In the event that any of these checks fail, the header is immediately skipped and the algorithm restarts the process on the next header. Else, if the XML validates correctly, the algorithm adds the fragment to the array at the end of the potential path it was associated with in the dictionary tree. This algorithm repeats until the depth of tree reaches 70 or a perfect fit is found where the length of the uncompressed data is the same as the uncompressed data value from the header. If that perfect data stream is found, it is written to the array in the "best fit" value and the header is marked as "complete" so operations are not continued on it. Using the process, the image needs to be gone over at most twice, as the first run through may start a stream halfway through the image, meaning the fragments that came before would not be considered if the stream was fragmented in a non-sequential way.

5.3.4 Fragment Reassembly

With all the headers and compressed data fragments in hand, reassembly is mostly trivial. For one, DOCX files follows the same order of files internally. Files that can have multiple different versions, such as themes or images, are labeled in sequential order as "theme1", "theme2", "theme3", and so on. This order removes most guess-work in the process. Secondly, the relative offset from the central directory header offers an excellent way to group headers together into potential files based on their compressed size. The odds of two files having all the same relative offsets is statistically unlikely, making it extremely easy to group these headers together.

To reassemble the DOCX files, the carver first takes all of the local file headers and the central directory headers. Each of these headers have a CRC of the compressed data. The carver attempts to match each of the local file headers to central directory headers by matching the CRC values. While CRC is not designed to be unique, it is unique enough to allow for fair certainty that these pairs are unique. Then, starting with the "[Content_Types].xml" header, the carver attempts to find a header pair with a relative offset that lines up with the end of the first header's offset. This algorithms continues until it exhuausts the pool or all needed files are accounted for in the file. Finally, the footer contains a value that corresponds to the number of central directory records expected. If

the footer with the corresponding value exists, it is appended to the end, else the potential file is invalid and it's headers are put back into the pool.

5.4 Testing Materials and Procedures

Properly testing the fragmented file carver proved to be a challenge in itself. While some fragmented images are hosts by NIST through the CFReDS project, these images were extremely outdated and did not contain DOCX files, only DOC files [17]. This necessitated new fragmented image creation. The first created images was an eight gigabyte SDHC card that contained two DOCX files from the available Word templates and was formated as a FAT32 filesystem. This drive was imaged using dd. This image was not fragmented and was solely used for testing of the carver during development. Actual fragmented files were created programmatically. First, five DOCX files were output to a text document as a raw byte stream. The program converted each of these byte streams to a Python bytearray and split them into the desired amount of fragments, each roughly equal in size. Next, the program opened a new file and it wrote 2,560,000 random bytes to the file via a cryptographic random number generator. Then, for each fragment in the bytearray, it wrote the fragment to the image, followed by a stream of random bytes 512 to 4096 bytes long. Between each file stream an additional 2,560,000 random bytes were written. This process was done for as many files were needed on disk. In total, eight disk images were created:

- 1-file-2-frag-512-break.img
- 1-file-3-frag-1024-break.img
- 1-file-4-frag-2048-break.img
- 2-file-3-frag-1536-break.img
- 2-file-4-frag-2048-break.img
- 3-file-3-frag-1536-break.img
- 3-file-4-frag-2048-break.img
- 4-file-1-frag-0-break.img

While still not as accurate as real file system data, the data looks more correct that it did in the NIST data set, where data that was not related to the file was a stream of all the same byte. In addition, the cut off points for the fragments are file system accurate instead of having magic bytes starting at the beginning of the fragment every time.

These images were split up into eight tests to evaluate the effectiveness and accuracy of the fragmented file carver. These tests were devised to test variable spaces between fragments, resource utilization when dealing with multiple fragmented files, and to test time requirements. The tests were ran on an Intel i7-1165G7 4-core CPU with 32 gigabytes of RAM.

- Test 1 - Four contiguous DOCX files

- Test 2 - One fragmented file split into two fragments, 512 bytes apart
- Test 3 - One fragmented file split into three fragments, 1024 bytes apart
- Test 4 - One fragmented file split into four fragments, 2048 bytes apart
- Test 5 - Two fragmented file split into three fragments, 1536 bytes apart
- Test 6 - Two fragmented file split into four fragments, 2048 bytes apart
- Test 7 - Three fragmented file split into three fragments, 1536 bytes apart
- Test 8 - Three fragmented file split into four fragments, 2048 bytes apart

As for the carver, it is currently in an incomplete state. The carver is currently capable of carving most of the internal files in a DOCX file, but struggles with XML files related to document properties due to incomplete XML validation. Until a method is devised to validate incomplete XML data, this will likely continue to be an issue. Since these files are missing, reassembly would not work. That said, the carver is still capable of carving all of the forensically important files from the system. The files that carver can carve are most of the property files and "document.xml", the file containing all of the actual user written text in a Word document.

6 Results

I conducted eight tests to evaluate the effectiveness of the partial implementation of the fragmented file carver. From each test I collected the real-time to complete the task, max CPU usage, and max memory usage. As can be seen in the table below, each time a new fragmented file was added, the time to execute goes up by a second or lower other than Test 1.

Table 1: Runtime and Resource Utilization

Test	Real Time	Max CPU Utilization	Max Memory
Test 1	6.83s	39%	31MB
Test 2	1.06s	36%	31MB
Test 3	1.07s	12%	30MB
Test 4	1.09s	34%	30MB
Test 5	1.82s	40%	30MB
Test 6	1.76s	41%	31MB
Test 7	2.23s	39%	31MB
Test 8	2.22s	37%	31MB

This small growth per added file is due to the computation reduction methods taken in our implementation, namely discarding new fragments that are more than 16,384 bytes away from the last fragment in the sequence. This means there were at most 32 fragments that needed to be checked against the end of the data stream, resulting a very small relative

search space. This is also reflected in the RAM usage. While storing all the data associated with the data stream is memory expensive, the small search space significantly reduces the amount the trees can grow. As for Test 1, the reason for the massive different in real-time is likely due to a multitude of factors. One, this test set had one more file than the other seven. Secondly, since the files were all contiguous, there was additional XML data that initially appeared correct and had to be verified by the incomplete XML validator. This XML data would be thrown out as invalid, but the extra computation would need to be done to reach this conclusion at least a couple hundred times.

The carver is capable of carving forensically important files within the internal DOCX structure, successfully carving the "document.xml" file 100% of the time. At first glance, the numbers in the below table look concerning as it would seem around 50% of the time the carver fails. This, however, can be explained by images in the DOCX files. The carver deals exclusively with XML data, which is not found in images. Therefore this carver is currently incapable of carving images within DOCX files. Between 16 and 21 of the files counted in the "failed carves" section were images. This significantly reduces the number of actual failed carves. With the images file removed from the set, the carver's true effectiveness is shown. The rest of the failed carves were either document properties, custom XML definitions, or relationship files. As these types of files are simply metadata definitions for DOCX files, they of are no forensic significance. Every file of forensic significance was carved correctly and successfully.

Table 2: Carving Performance

Test	Successful Carves	Partial Carves	Failed Carves	Actual Failed Carves
Test 1	50	4	33	12
Test 2	10	1	19	3
Test 3	10	1	19	3
Test 4	10	1	19	3
Test 5	22	3	26	6
Test 6	22	3	26	6
Test 7	52	3	30	9
Test 8	52	3	30	9

7 Future Work

The current implementation of the file carver still needs a lot of work and improvement. While very fast on small disk images, it struggles in terms of RAM usages and time needed on large disks. Reconstructing multiple potential paths and storing that data in RAM is expensive and requires further research to reduce memory usage while still limiting the amount of times the carver needs to read the disk. Secondly, the carver fails to carve certain internal DOCX files. This is due to the incomplete checks against XML data. This validator should fully cover the XML specification so nothing is lost. Future carvers will need a way

to quickly validate incomplete XML data. Finally, further research needs to be done in creating modern, file system accurate testing sets. NIST hosts a number of fragmented and non-fragmented images through the CFReDS project [17]. While somewhat useful, this data is extremely outdated and not representative of current file systems. In some cases the data is also straight wrong, with the positioning of headers not lining up with how a file system would place the data.

8 Conclusion

This paper presented a theoretical process to carve fragmented DOCX files through a process of carving smaller internal files and validation of their uncompressed data. These algorithms use the internal files of DOCX files and the XML structure to reconstruct fragmented files. It also presents an implementation capable of carving forensically important fragmented documents within the DOCX file structure, proving the theorized processes work. The outlined process is fast, efficient, and highly scalable, making it an excellent basis for future research into fragmented document file carvers.

References

[1] "foremost." [Online]. Available: http://foremost.sourceforge.net/

[2] "scalpel," October 2014. [Online]. Available: https://github.com/sleuthkit/scalpel

[3] A. Pal and N. Memon, "The evolution of file carving," *IEEE Signal Processing Magazine*, vol. 26, no. 2, pp. 59–71, 2009.

[4] ".zip file format specification." [Online]. Available: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT

[5] S. L. Garfinkel, "Carving contiguous and fragmented files with fast object validation," *Digital Investigation*, vol. 4, pp. 2–12, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287607000369

[6] Niklaus Aeschbache, "Zip-64 internal layout.svg," https://commons.wikimedia.org/wiki/File:ZIP-64_Internal_Layout.svg.

[7] A. Pal and N. Memon, 2009. [Online]. Available: https://ieeexplore.ieee.org/document/4806206/

[8] M. Karresand and N. Shahmehri, "Oscar - file type identification of binary data in disk clusters and ram pages," in *Security and Privacy in Dynamic Environments*, vol. 201, 07 2006, pp. 413–424.

[9] "Carver 2.0 planning page," September 2012. [Online]. Available: https://forensicswiki.xyz/wiki/index.php?title=Carver_2.0_Planning_Page

[10] R. Poisel, "mmc," September 2012. [Online]. Available: https://github.com/rpoisel/mmc

[11] R. Poisel, S. Tjoa, and P. Tavolato, "Advanced file carving approaches for multimedia files," *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)*, vol. 2, pp. 42–58, 12 2011.

[12] B. Li, L. Wang, Y. Sun, and Q. Wang, "Image fragment carving algorithms based on pixel similarity," in *2012 Fourth International Conference on Multimedia Information Networking and Security*, 2012, pp. 979–982.

[13] N. Memon and A. Pal, "Automated reassembly of file fragmented images using greedy algorithms," *IEEE Transactions on Image Processing*, vol. 15, no. 2, pp. 385–393, 2006.

[14] G. Mittal, P. Korus, and N. Memon, "Fifty: Large-scale file fragment type identification using convolutional neural networks," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 28–41, 2021.

[15] N. L. Beebe, L. A. Maddox, L. Liu, and M. Sun, "Sceadan: Using concatenated n-gram vectors for improved file and data type classification," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 9, pp. 1519–1530, 2013.

[16] F. Wang, T. Quach, J. Wheeler, J. B. Aimone, and C. D. James, "Sparse coding for n-gram feature extraction and training for file fragment classification," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 10, pp. 2553–2562, 2018.

[17] NIST, "Forensic images for file carving," https://www.cfreds.nist.gov/FileCarving/.