Project 2 Report

1. **Executive Summary**

For project 2, we are tasked with developing a program that will automatically recover certain file types found on a disk image. We should be able to perform this task without taking account of the file system, rather we are going to use the known file headers and footers. From our work, we have recovered \_\_\_files from the provided disk image. The program can be broken up into two parts: an identifier and carver. The program itself is not optimized for efficiency; therefore, the file recovery process can take up to 10 minutes depending on the computing resources available. Additionally, the program displays the files starting offset, ending offset, file extension, and the SHA-256 value. Furthermore, the program may provide false positives near the end of the file for bitmap (BMP) due to the signature being small.

1. **Problem Description**

For this assignment, our team was tasked with creating a program that can identify and recover select file types found on the provided disk, named Project2Updated.dd. The file types that were selected are MPG, PDF, BMP, GIF, ZIP, JGP, DOCX, AVI, and PNG. Additionally, the program needs to generate a SHA-256 hash value for each file that has been recovered during the process. Note, the assignment came with some restrictions. For the assignment, the team is not making use of the Python exec() function to simulate a command line. Therefore, the team must make use of built in Python file commands to recover the files.

1. **Process Description**

**Overview:**

The program can be broken down into 2 phases. The first phase of the program deals with identifying all the files found on the disk image and storing the relevant information for later. The second phase of the program deals with using the stored information and the disk image to recover the files from the disk. However, there is an exception to this rule since the ZIP header is contained inside the DOCX header. The carver will check for the difference between ZIP and DOCX files. False positives may occur at the end of file with BMPs because the signature is only two bytes and can occur in random or text data.

**File Identification:**

During the first phase, the program begins by opening the disk image in binary mode through the open function and providing “rb” as the second parameter. This allows the program to read the file in byte by byte when we use the read() function. It is important to note we need to provide a numerical value to the read function as a parameter since this will allow the read function to know how many bytes we are reading in. During this phase, the value will always be 1.

Once the byte is read in, it will be added to the byte array, that will be acting as a running signature. This signature will be compared to the known header signatures to see if it matches a file header. This process can take time as there are numerous headers to check against, some of which, like MPG, JPG, or GIF, have variations in their headers.

The next step may differ based on the file type found. If the file has the length indicated in the header, the identifier will extract the length from the header and store the file information including file type, offset and length, in a list for the carver to use, it will also skip forward in the disk by the length of file to reduce false positives (except for BMPs which are prone to false positives). If the file length is not in the header, the identifier will begin searching for the matching footer. Upon finding the matching footer, the identifier will calculate the file length and store the file data for the carver to use.

The process for finding footers can be somewhat more complicated in the case of PDFs and GIFs. This is because these types of files can have multiple footers and one must take the last footer. When searching for the end of a PDF or GIF the program will search for both footers and headers (except for BMP headers because cause a high degree of false positives). The program will store the location of the last footer found and continue searching until it finds a header. Once it has found a header it will return with location of the last found footer and allow the file data to be recorded.

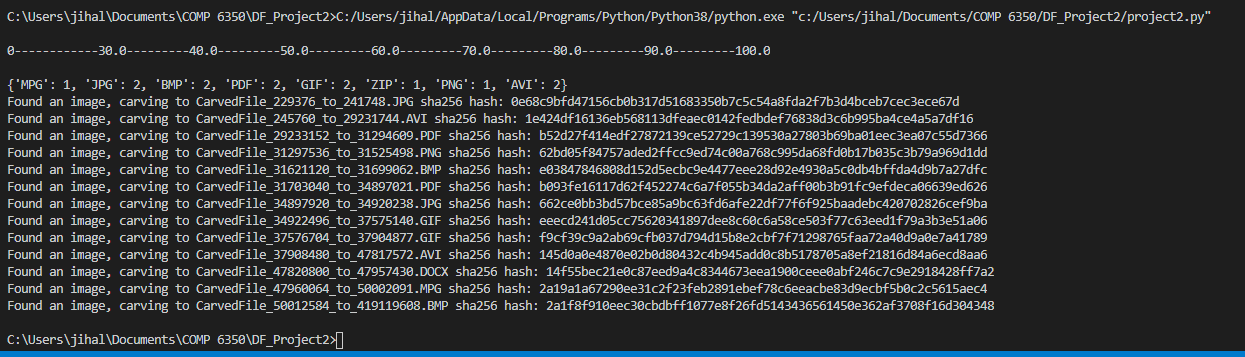
After the disk has been scanned the identifier will attempt to remove false positive BMPs. Without this process the program identifies nearly 1900 BMPs on the provided disk. This is because the signature for BMP files is so small. In order to clean them up, the identifier uses two double layer while loops. The first set of loops runs over every BMP and checks if it runs over the area of any non-BMP files, since the likelihood of a non-BMP file producing a false positive is significantly lower, the BMPs are assumed to be false and removed from the files list. The second set of loops checks for BMPs that conflict with each other, prioritizing the earlier BMP instance. After this cleaning the number of bitmaps found falls to two, one of which is still a false positive.

**File Recovery:**

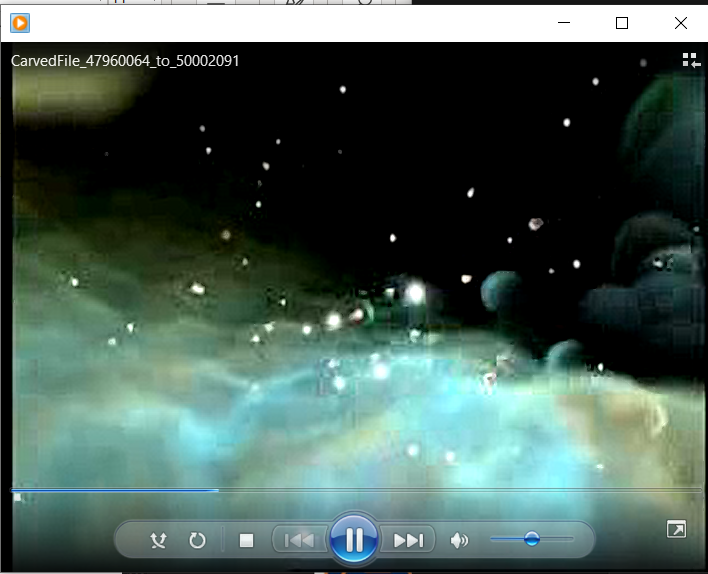
Once all the files have been identified, the program will enter the file recovery phase. The program begins by reading the entire disk and storing it a variable. Once that is complete, the program will begin iterating over the file data dictionaries. The first thing done inside the loop is to retrieve the data from the disk. This is done by accessing the variable and jumping to the byte offset and adding the computed file length. This is stored to a variable that will be used to write the information into the newly created file. Once the data is retrieved, we create a new file using the type field as the extension and using the beginning and ending offsets as the identifier. However, it is important to note that if we encounter the file type ZIP we are actually going to check the 5th-8th bytes and see if it is actually a DOCX file. Once there, we will create a new file with the corresponding extension, we will write back the data to the newly created file. Additionally, the program will display the recovered file name (which includes both offsets of where the file begins and ends) and the SHA-256 hash the of the file. This process is repeated for all identified files.

1. **Screenshots/Results**

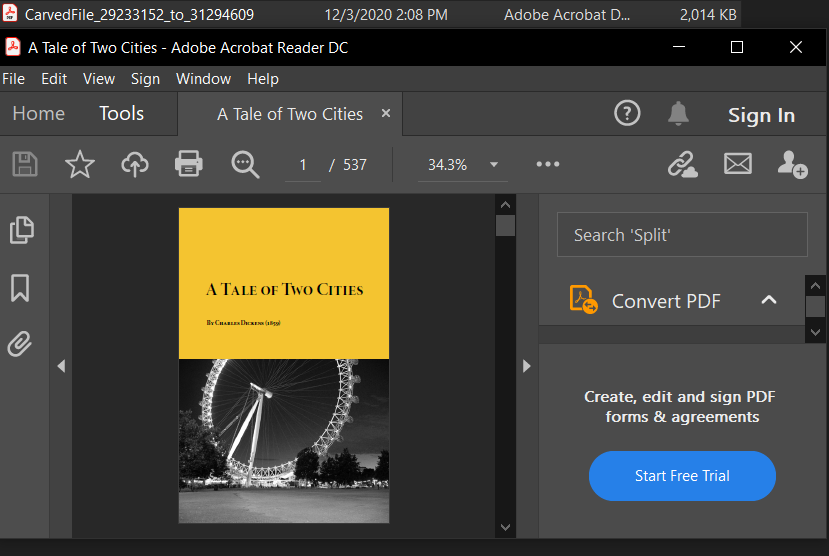
**Terminal Output:**

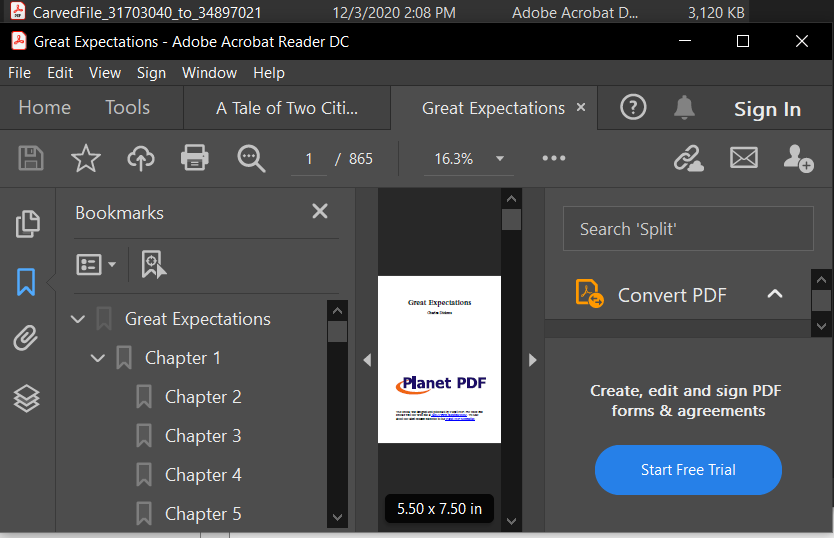
****

**MPG:**

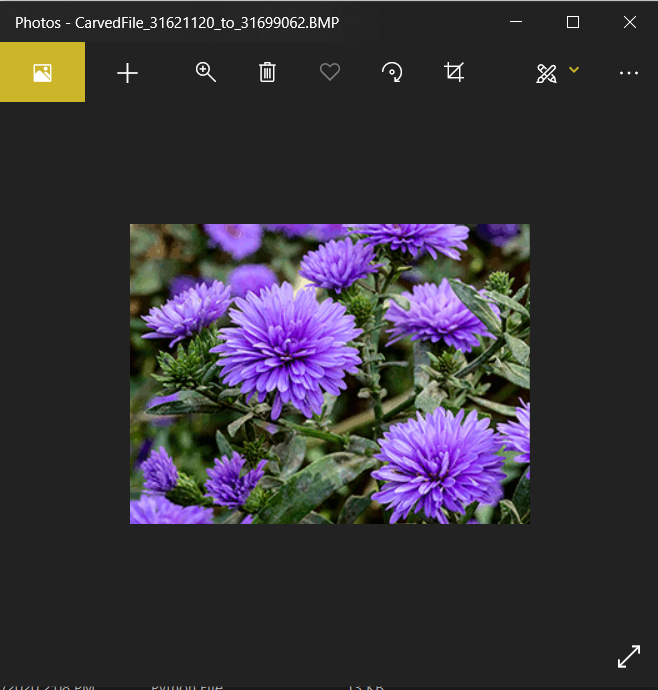


**PDF:**

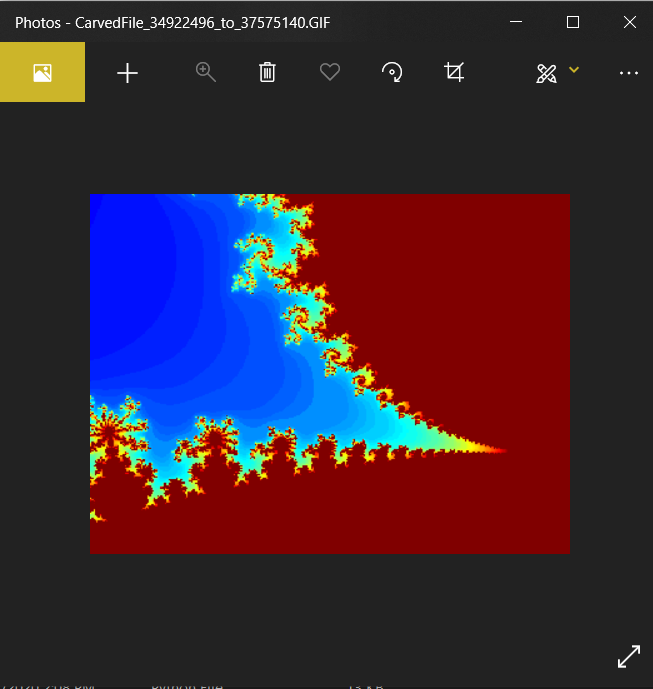


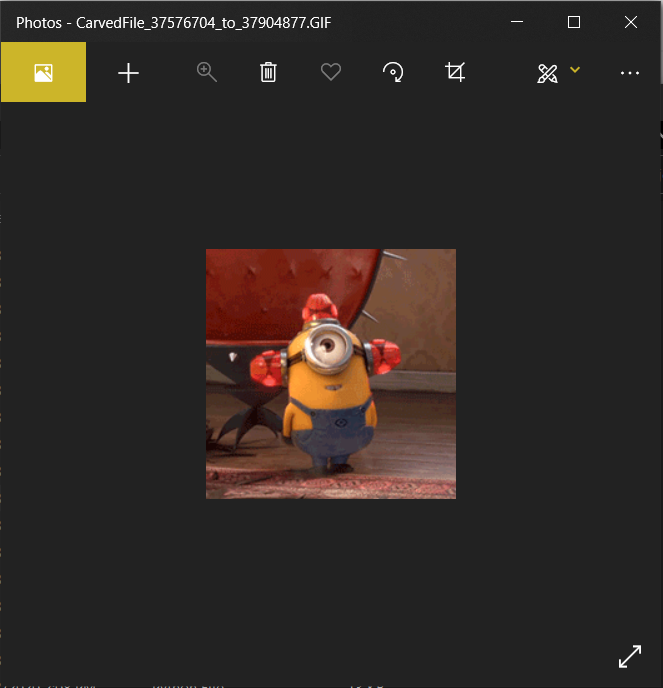


**BMP:**



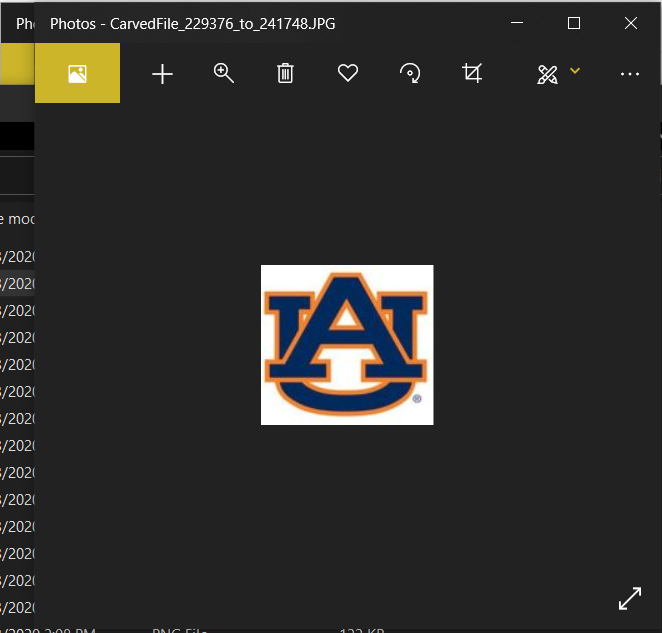
**GIF:**

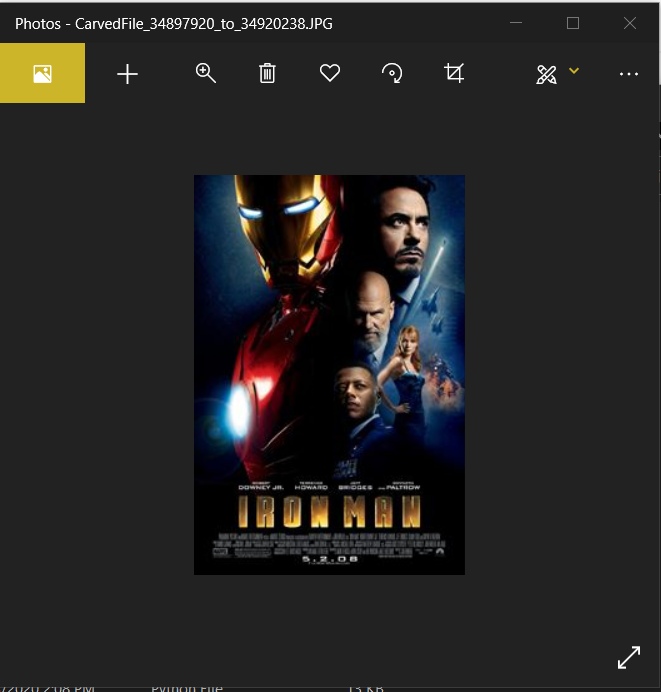




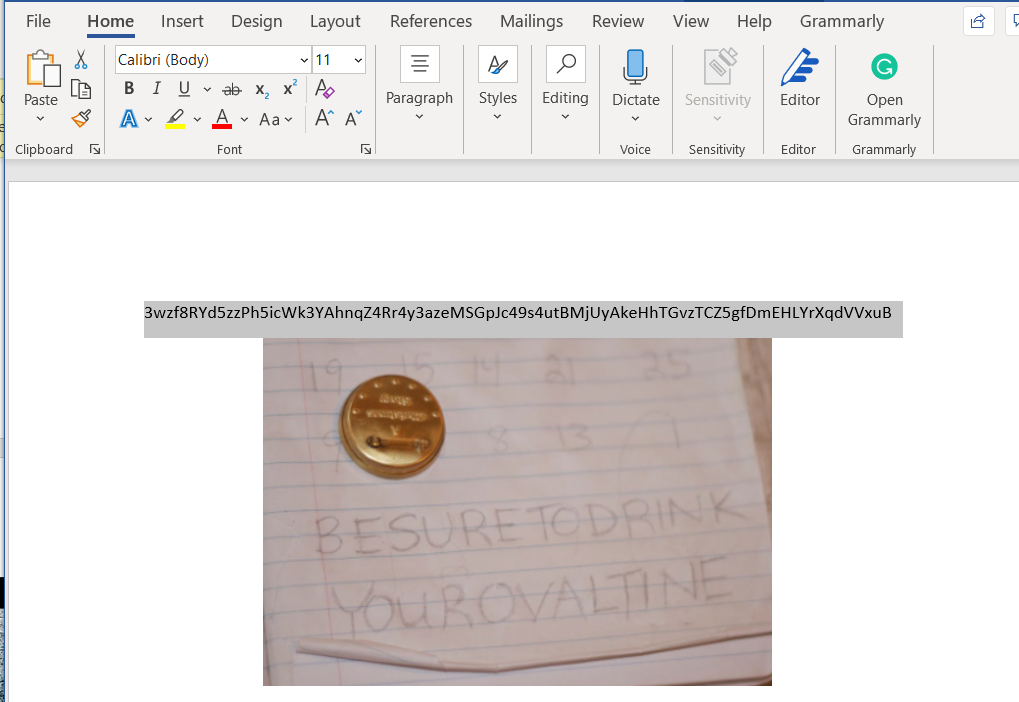
**ZIP:**

**JGP:**

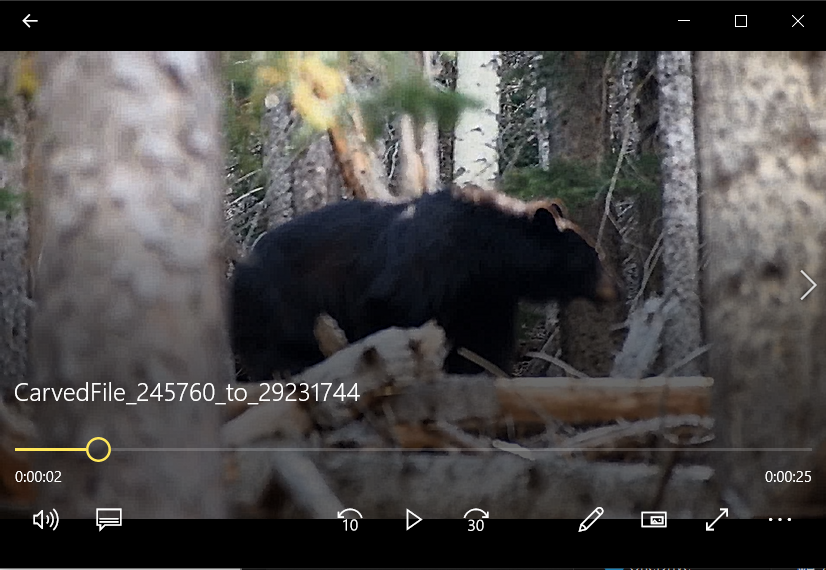


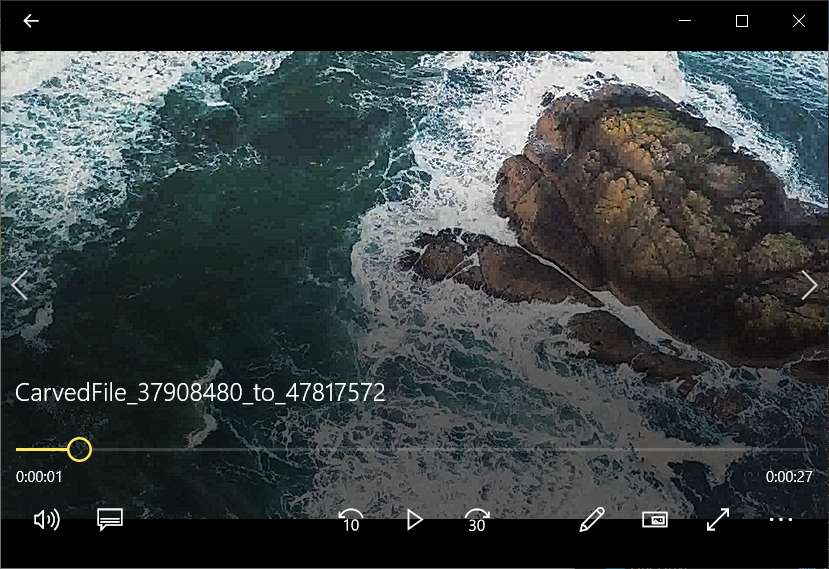


**DOCX:**

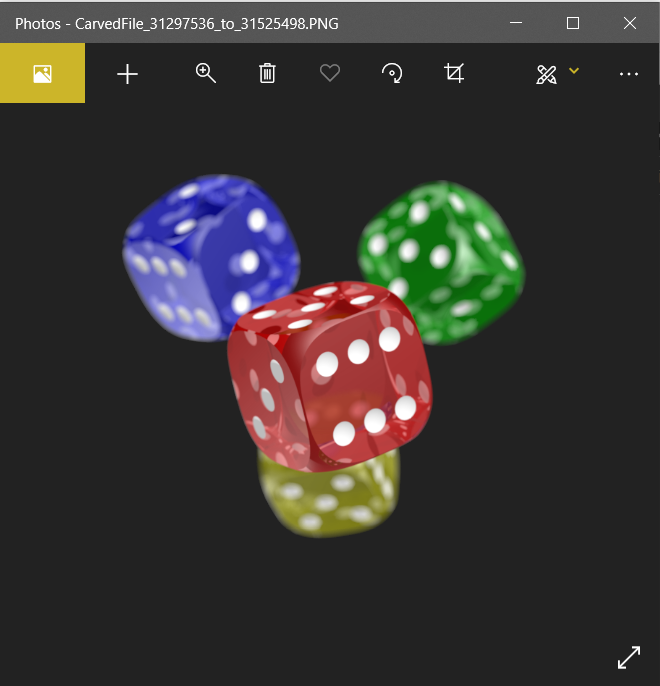


**AVI:**





**PNG:**



1. **Conclusions/Recommendations**

To conclude, the program can successfully identify and recover files; however, it is important to note that our code is not optimized for efficiency. In the perfect scenario, the program would probably identify and recover during the same time. For instance, as it would begin by looking for header signature, and once a header is found it would generate a new file and begin the writing to the file. The program would continue to write until it encountered the corresponding footer if there was one needed. Otherwise, it would just write the number of bytes specified in the header signature. PDF would be the only thing that differs from these two scenarios as there can be multiple footers and we need to take the last one. However, the main point is that the disk image would have been read through once compared to the several times we have to open the disk image.

The main reasoning behind our design decision was to allow each component to be worked on separately. This allowed the file carver and file identifier to be worked on without being reliant on the other. Due to this design choice, both parts would need access to the disk and the identifier would need to relay additional information required to successfully recover the files.