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 depending on the computer hardware environment. 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 and during the program itself prioritizes other file types over 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 certain file types, such as MPG, AVI, and BMP. This is probably a design flaw since that it probably is due to the fact adding the computed/provided file length/size at the starting offset would give it a ending offset existing outside the realm of the disk.

**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 one of them or if it is a part of a longer signature. Due to the nature of some of the header signatures, such as AVI, MPG, and BMP, allowing for deviation in the bytes we had to provide different checks to find the signatures. The deviation in the signature is due to the fact that AVI and BMP both provide the file size in their header, and MPG allows the last byte to differ.

The next step may differ based on the file type found. If an AVI or BMP is found, we simply will create a dictionary and return the values of the current offset, type of file, and file size found in the header. Otherwise, we will return a dictionary containing the type of file and current offset. Additionally, we set the file size to -1 to indicate to the program that the size has not yet computed.

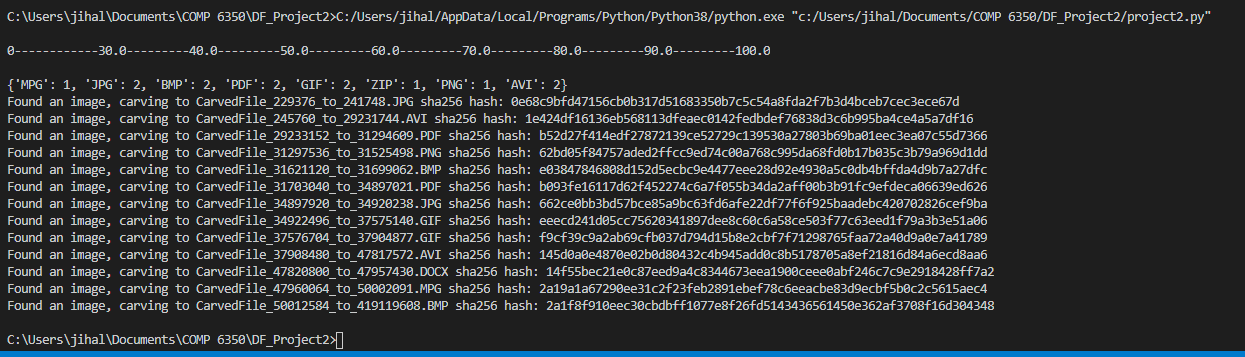
Once the dictionary has been returned, if we found an AVI or BMP we will append the file data, represented as a dictionary, to a list. After, we will skip over the bytes that make up the file data. If we find any other type, we change the mode in scanner/seeker to look for footers.

**File Recovery:**

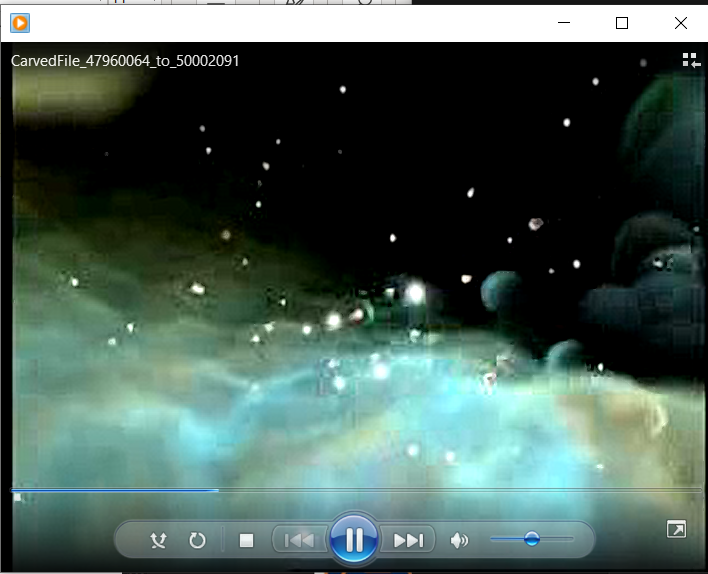
Once all the files have been identified, the program will enter the file recovery phase. The program begins by reading the entire disk and storing it a variable. Once that is complete, the program will begin iterating over the file data dictionaries. The first thing done inside the loop is to retrieve 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 offset 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 byte 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 the file name and both offsets of where the file begins and ends. Furthermore, the program will SHA hash the file, and generate and display the hash value. After, we close the file and we will repeat this until we are done iterating over the file data dictionaries in the list.

1. **Screenshots/Results**

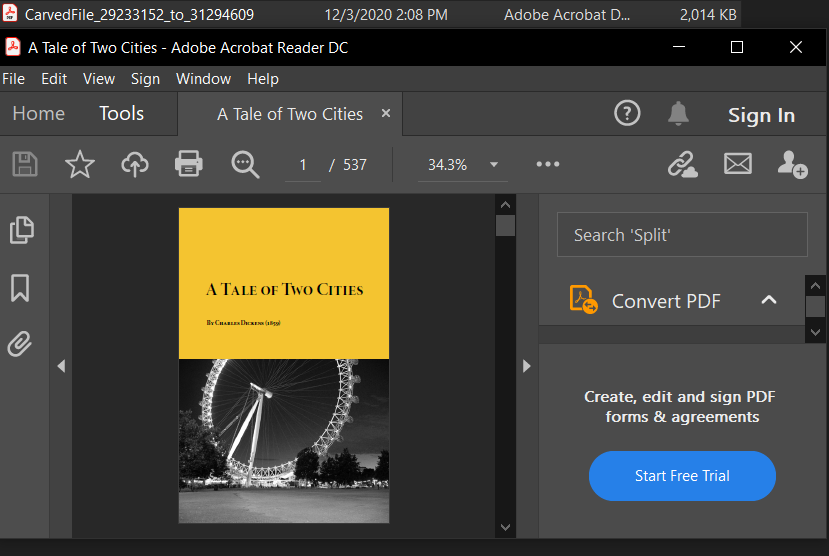
**Terminal Output:**

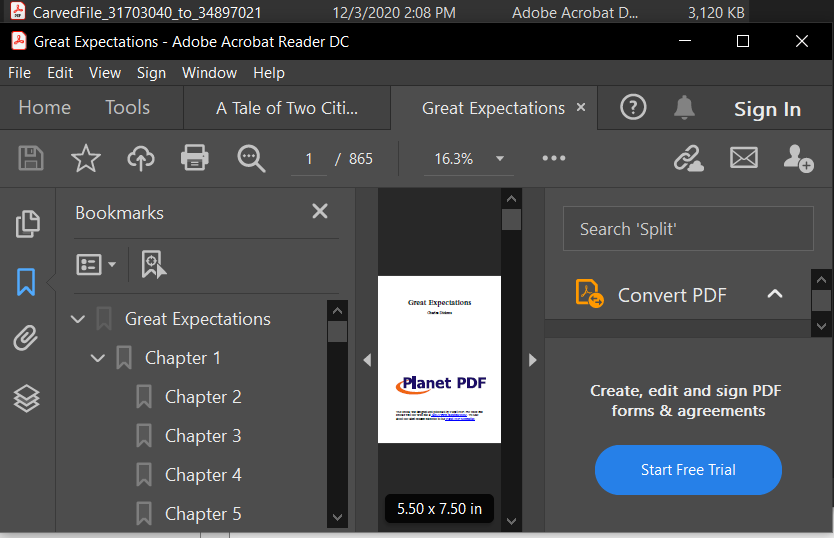
****

**MPG:**

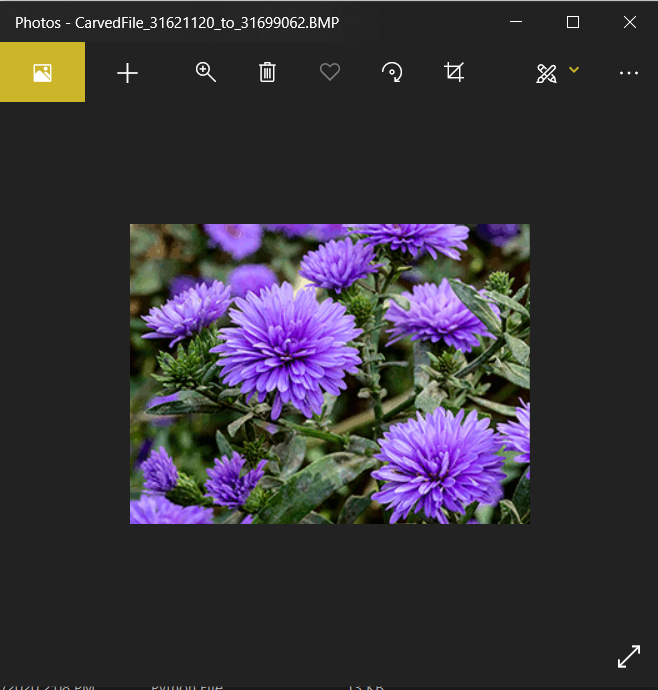


**PDF:**

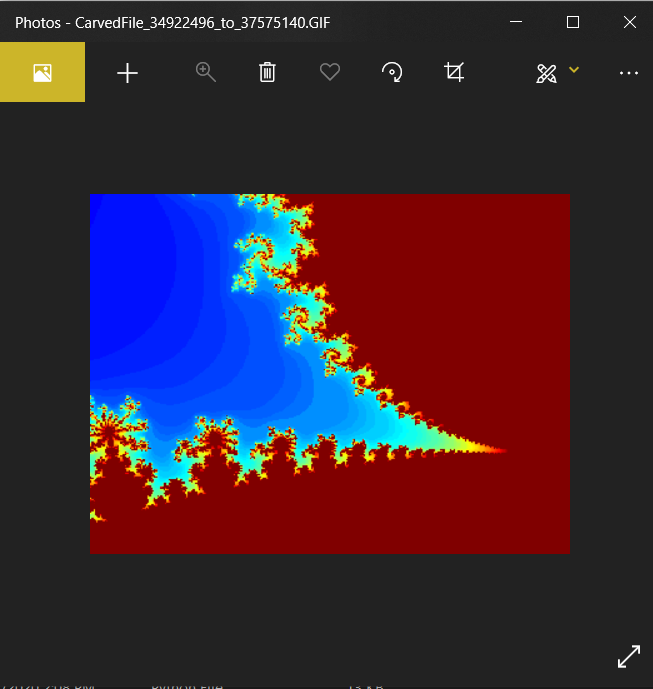


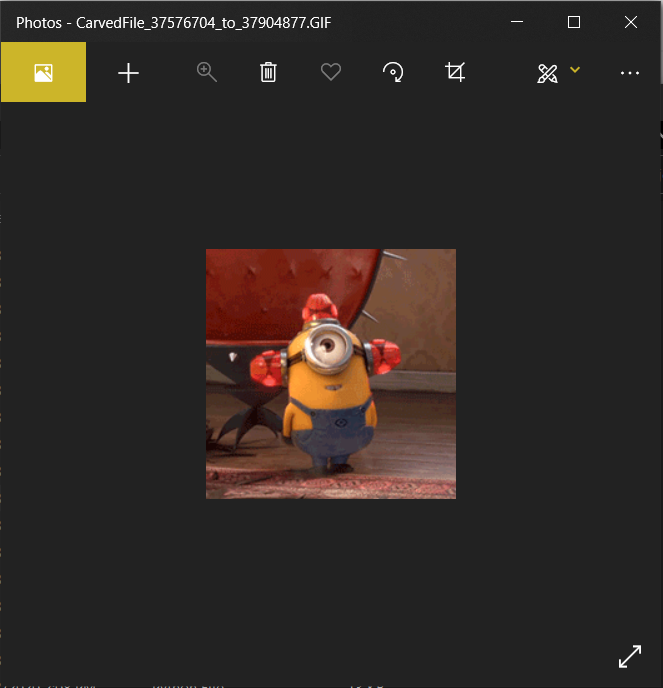


**BMP:**



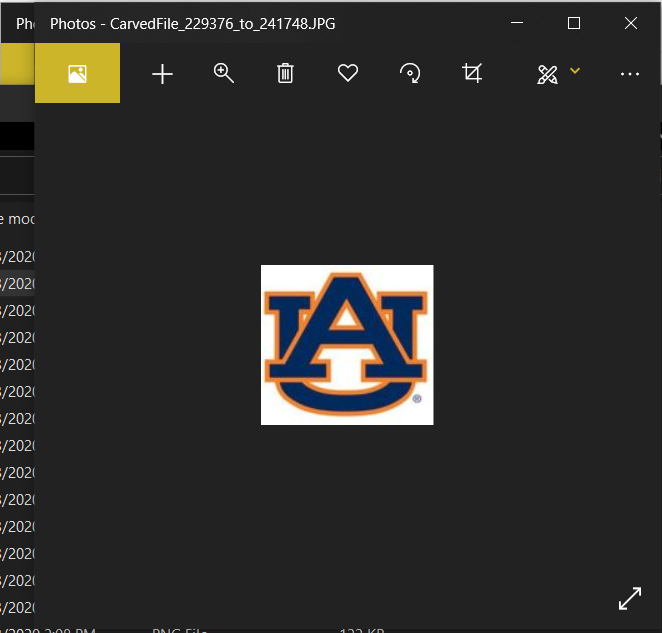
**GIF:**

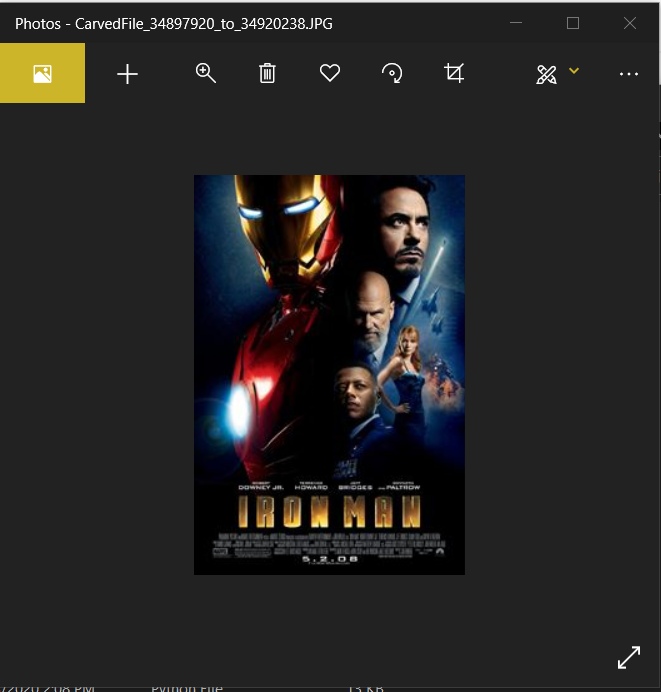




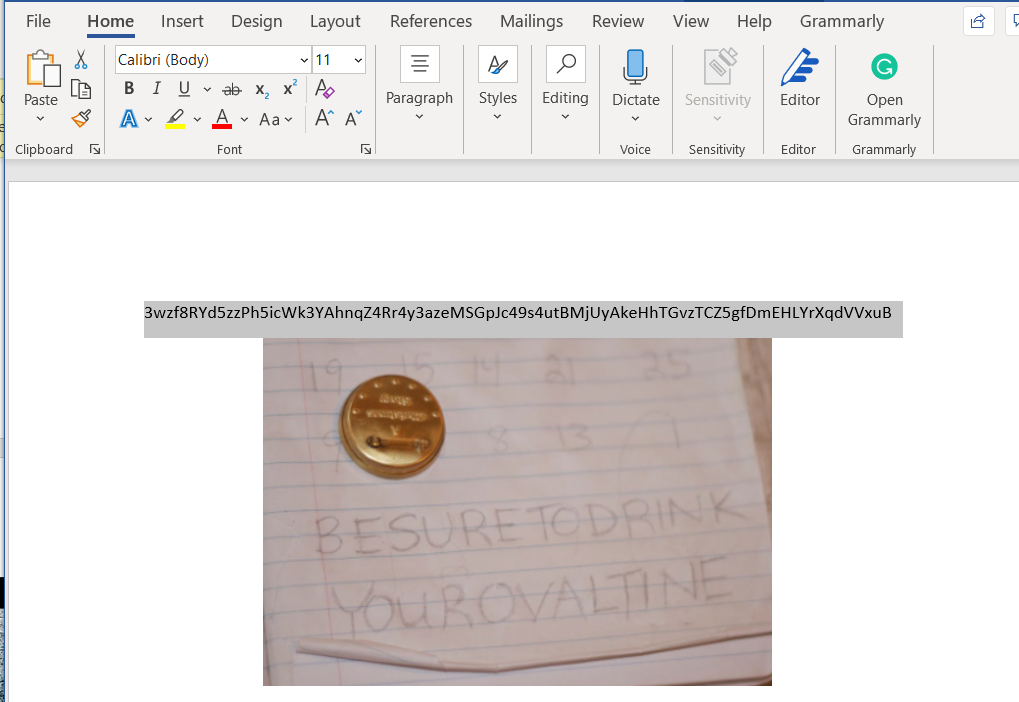
**ZIP:**

**JGP:**

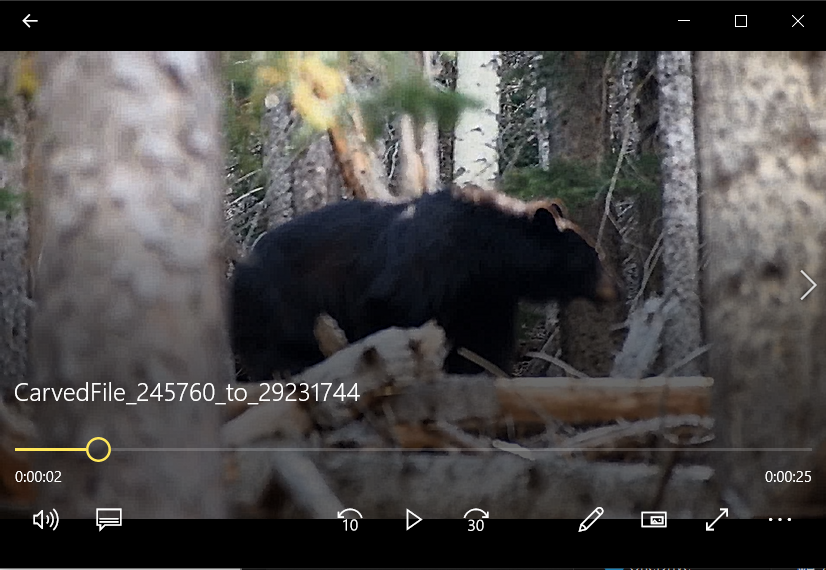


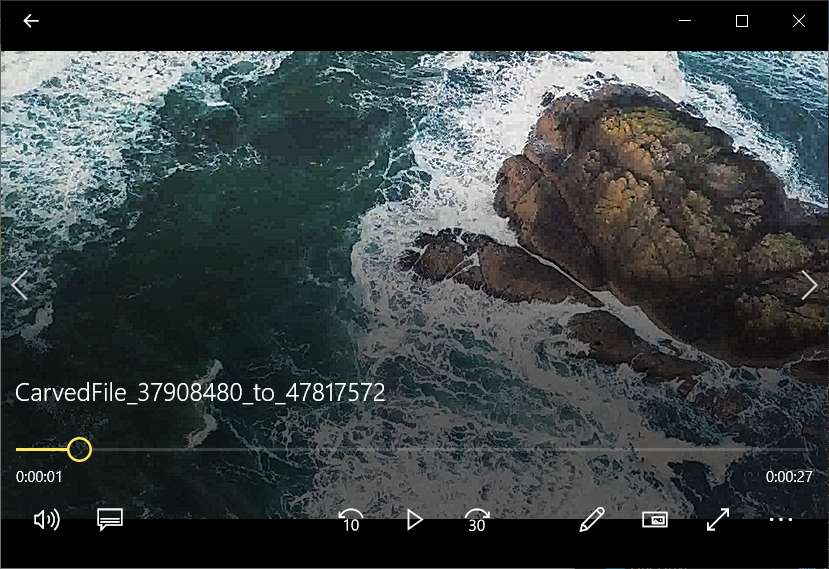


**DOCX:**

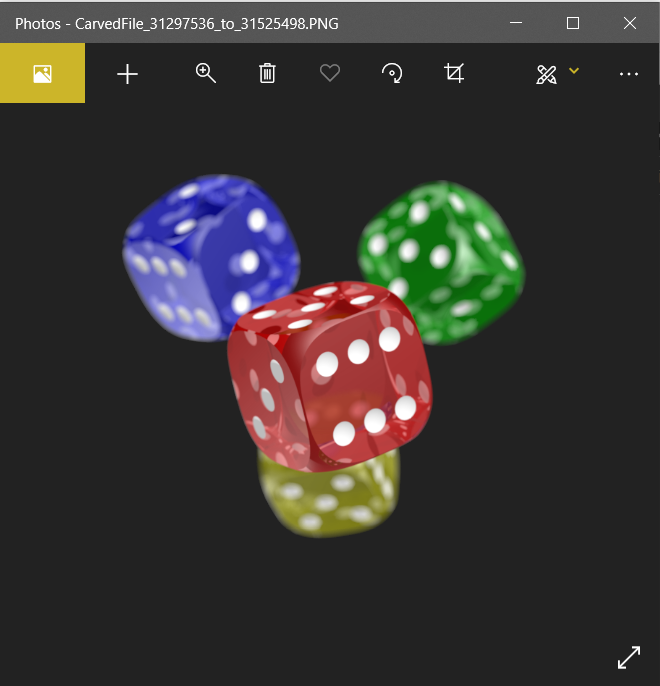


**AVI:**





**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.