Question - You should submit one zip file containing your Python files and the required output. All submissions

will be tested with the Python configuration used on the lab machines – ensure your submission

works as expected before submission. No additional libraries will be installed to test submissions.

Assessment Details:

A disk was found next to an employee’s computer that appears to be corrupt. As part of the AcmeForensics team responding to the incident, you have been given the task of trying to recover data from the corrupted fat16 partially wiped disk.

Marking Guidelines:

• 10% Quality of code, including commenting and code structure

• 10% Identifying form of corruption, as evidenced in summary report

• 10% Identification of number of files on disk; should be stated in summary report

• 20% Producing directory of correctly recovered files

o The directory must be named RecoveredFiles and submitted in your zip file

o Recovered files must be named FILE0001.BIN, FILE0002.BIN etc.

o These files will almost certainly have incorrect lengths as each will be a whole

number of clusters, but should otherwise be correct

o Files must be recovered without making assumptions about the filesystem layout,

e.g., making assumption that adjacent blocks will be part of same file is not enough

• 5% Trimming files to correct lengths

o Trimmed files must be named FILE0001.TXT, FILE0002.TXT etc.

o These files must be stored along with the .BIN files in the RecoveredFiles directory

• 15% Identifying which, if any, of the recovered files are in fact directories

o A listing of these directories and the files they contain must be added to the zip file

o Correctly named copies of these files must be in a GoodFiles directory in your zip

▪ These files must have the correct length as given in the directory

• 10% Identifying unlinked files, i.e., those not part of a recovered directory

o Corresponding .BIN and .TXT files must be in a directory named Unlinked in the zip

• 5% Submission of paperwork (completed as necessary), submitted in your zip file

• 5% Identification of any files of forensic interest

• 10% Properly structured and well-formatted forensics report (a PDF in the zip):

o 1 page summary of findings

© 2024 Andrew Scott

Submission

Your submitted zip file should contain the following content… Six top-level directories, as follows:

• Python files

o Code to complete exercise, including sub-directories if required

o Brief text file with essential instructions on how to operate your program to produce

all expected output. This should be complete but not contain unnecessary detail – this

is not intended to be full user documentation.

• Report

o Associated paperwork, completed as necessary (PDF format)

o PDF copy of your summary report – maximum 1 page

▪ Describe form of corruption seen in filesystem

• This must identify number of files and directories found

• RecoveredFiles

o Recovered .BIN and trimmed .TXT files, named as described above

• GoodFiles

o List of directories and the files they contain – a single file called listing.txt

o Files identified from recovered directories with their correct names and lengths

• Unlinked

o Any files not appearing in identified directories - .BIN and .TXT files

• Evidence

o A copy of any files of particular forensic interest

Making a Start

Overleaf you will find sample code that implements a basic FAT16 class that can be called as follows:

fat = FAT( 'fat16.img' )

print( fat )

fat.close( )

Note that this only handles FAT16, i.e., volumes with 4085 to 65524 clusters, which should be fine

for what you are doing. That said, make sure you are testing your code with FAT16 images – the

script from the FAT lab will always create a FAT16 filesystem.

Hints

1. You must not make assumptions about the filesystem layout; however, there is no need to

fully decode everything... you are trying to find and reconstruct files, and then check whether

they are regular files or directories. If you are going much beyond this and just dumping

blocks/ clusters to your h-drive, you may be overthinking what is expected.

2. You are not required to handle long filenames; you can safely stick to the main 8.3 entries.

3. While the level of corruption is significant, the problem should be easily identifiable if you

followed and understood the FAT lab. The hex dump utility can be useful for spotting issues.

4. Big hint: remember that in a FAT, every file has a clearly identifiable end.

5. The first directory entries are special and easily identifiable – the first in more than one way.

6. You can easily test your program on known content by using the script from the FAT lab.

© 2024 Andrew Scott

Getting Going: basic FAT16 code

import sys

import struct

class FAT :

def getSector( self, sector: int ) -> bytes :

self.fatFile.seek( sector \* self.BytsPerSec )

return self.fatFile.read( self.BytsPerSec )

def \_\_init\_\_( self, imageFileName: str ) -> None :

self.fatFile = open( imageFileName, 'rb' )

self.BytsPerSec = 512 # We assume 512 bytes until we know better

block0 = self.getSector( 0 ) # Read metadata from first sector

# Unpack filesystem metadata

#

self.jmpBoot, self.OemName, self.BytsPerSec, self.SecPerClus, \

self.ResvdSecCnt, self.NumFATs, self.RootEntCnt, \

self.TotSec16, self.Media, self.FATSz16, self.SecPerTrk, \

self.NumHeads, self.HiddSec, self.TotSec32, self.FATSz32 = \

struct.unpack( '<3s8sHBHBHHBHHHLLL', block0[ : 40 ] )

# Calculate some useful values -- See Microsoft Whitepaper

#

self.RootDirSectors = int(

( self.RootEntCnt \* 32 + self.BytsPerSec - 1 ) /

self.BytsPerSec )

self.FirstDataSector = self.ResvdSecCnt + (

self.NumFATs \* self.FATSz16 ) + self.RootDirSectors

self.DataSec = self.TotSec16 - ( self.ResvdSecCnt +

( self.NumFATs \* self.FATSz16 ) + self.RootDirSectors )

self.CountOfClusters = int( self.DataSec / self.SecPerClus )

self.FATStart = self.ResvdSecCnt # Sectors; and 1st root sector is:

self.RootDirStart = self.ResvdSecCnt + self.NumFATs \* self.FATSz

def \_\_str\_\_( self ) -> str :

return f'\n{self.BytsPerSec=} bytes\n{self.SecPerClus=}

sectors\n{self.ResvdSecCnt=} sectors\n{self.NumFATs=}\n{self.RootEntCnt=}

entries\n{self.TotSec16=} sectors\n{self.FATSz16=} sectors\n{self.HiddSec=}

sectors\n\n'

def close( self ) -> None :

self.fatFile.close( )

Code walkthrough –

1) import struct

import os

class FAT:

    def \_\_init\_\_(self, image\_filename):

        try:

            # Read the first sector (boot sector)

            block0 = self.get\_sector(0)

            # Unpack metadata (focus on relevant information)

            self.jmp\_boot, self.oem\_name, self.bytes\_per\_sector, self.sectors\_per\_cluster, \

            self.reserved\_sectors, self.num\_fats, self.root\_entries\_count, \

            self.total\_sectors\_16, self.media\_type, self.fat\_size\_16 = \

                struct.unpack('<3s8sHBHBHHBHHHLLL', block0[:40])

            # Calculate derived values

            self.root\_dir\_sectors = int((self.root\_entries\_count \* 32 + self.bytes\_per\_sector - 1) / self.bytes\_per\_sector)

            self.first\_data\_sector = self.reserved\_sectors + (self.num\_fats \* self.fat\_size\_16) + self.root\_dir\_sectors

            self.data\_sectors = self.total\_sectors\_16 - (self.reserved\_sectors + (self.num\_fats \* self.fat\_size\_16) + self.root\_dir\_sectors)

            self.cluster\_count = int(self.data\_sectors / self.sectors\_per\_cluster)

            self.fat\_start = self.reserved\_sectors  # Sector number of the first FAT table

        except struct.error as e:

            print(f"Error unpacking boot sector: {e}")

            sys.exit(1)

        self.image\_file = open(image\_filename, 'rb')

        self.bytes\_per\_sector = 512

    def get\_sector(self, sector\_number):

        """

        Reads a sector from the disk image.

        """

        self.image\_file.seek(sector\_number \* self.bytes\_per\_sector)

        return self.image\_file.read(self.bytes\_per\_sector)

    def close(self):

        self.image\_file.close()

    def is\_valid\_directory\_entry(self, directory\_entry):

        """

        Checks if a directory entry is valid (excluding unused and end-of-directory markers).

        """

        return (

            0 != directory\_entry[0] and  # First byte not zero (unused)

            directory\_entry[0] != 0xE5  # First byte not 0xE5 (end of directory)

        )

    def parse\_filename(self, directory\_entry):

        """

        Extracts the filename (8.3 format) from a directory entry.

        """

        filename = ""

        # Extract characters for filename (excluding spaces)

        for char\_offset in range(0, 8):

            char = directory\_entry[char\_offset]

            if char != 0x20:  # Exclude spaces

                filename += chr(char)

        # Extract characters for extension (excluding spaces)

        extension = ""

        for char\_offset in range(8, 11):

            char = directory\_entry[char\_offset]

            if char != 0x20:  # Exclude spaces

                extension += chr(char)

        # Combine filename and extension (if present)

        if extension:

            filename += "." + extension

        return filename

    def follow\_cluster\_chain(self, start\_cluster, recovered\_data, recovered\_files):

        """

        Follows a cluster chain, potentially recovering data, identifying end markers, detecting potential files,

        and performing signature-based identification for JPEG and PNG.

        """

        current\_cluster = start\_cluster

        while current\_cluster > 0 and current\_cluster < self.cluster\_count:

            # Get the FAT entry for the current cluster

            fat\_offset = (current\_cluster - 2) \* 2  # Assuming 16-bit FAT entries

            fat\_sector = self.fat\_start + int(fat\_offset / self.bytes\_per\_sector)

            fat\_entry\_offset = fat\_offset % self.bytes\_per\_sector

            fat\_data = self.get\_sector(fat\_sector)

            fat\_entry = struct.unpack('<H', fat\_data[fat\_entry\_offset:fat\_entry\_offset + 2])[0]

            # Extract data from the current cluster

            cluster\_data = self.get\_sector(self.first\_data\_sector + current\_cluster - 2)

            # Check for end-of-chain markers (0x0000 or 0xFFFF for FAT16)

            if fat\_entry in (0x0000, 0xFFFF):

                break

            # Data carving for JPEG and PNG signatures

            jpeg\_signature = b"\xFF\xD8"

            png\_signature = b"\x89PNG\r\n\x1A\n"

            for signature in [jpeg\_signature, png\_signature]:

                offset = 0

                while offset < len(cluster\_data):

                    signature\_pos = cluster\_data.find(signature, offset)

                    if signature\_pos != -1:

                        # Potential JPEG/PNG file found, extract data starting from signature

                        file\_data = cluster\_data[signature\_pos:]

                        # Continue extracting data from subsequent clusters until next signature or end-of-chain

                        next\_cluster = self.follow\_cluster\_chain(

                            current\_cluster + 1, recovered\_data, recovered\_files

                        )

                        if next\_cluster != 0xFFFF:  # Not end-of-chain

                            file\_data += recovered\_data.get(next\_cluster, b"")

                        offset = signature\_pos + len(signature)

                        # Identify potential file based on extension in directory entry

                        directory\_entry = recovered\_data.get(current\_cluster - 2, None)

                        if directory\_entry and self.is\_valid\_directory\_entry(directory\_entry):

                            filename = self.parse\_filename(directory\_entry)

                            extension = filename.split(".")[-1].lower()

                            if extension in ("jpg", "jpeg", "png", "txt", "pdf"):

                                recovered\_files.append((file\_data, filename))

                        else:

                            # Unlinked file - categorize later

                            recovered\_files.append((file\_data, f"UNLINKED\_{len(recovered\_files)}.BIN"))

                        break  # Found signature, break inner loop

                    else:

                        break  # No signature found in this cluster, move to next offset

            # Update recovered data for current cluster

            recovered\_data[current\_cluster - 2] = cluster\_data

            # Move to the next cluster in the chain

            current\_cluster = fat\_entry

        return current\_cluster  # Return the last cluster accessed

    def get\_directory\_structure(self, recovered\_data):

        """

        Reconstructs the directory structure based on valid directory entries.

        """

        directory\_structure = {}  # Dictionary to store directory paths and their subdirectories

        num\_files = 0  # Counter for identified files

        root\_sector = self.first\_data\_sector + self.root\_dir\_sectors - 1  # Last sector of the root directory

        for sector in range(self.first\_data\_sector, root\_sector + 1):

            sector\_data = self.get\_sector(sector)

            offset = 0

            while offset < len(sector\_data):

                directory\_entry = sector\_data[offset:offset + 32]  # Extract one directory entry

                if not self.is\_valid\_directory\_entry(directory\_entry):

                    break  # Reached end of directory entries or unused entries

                # Extract filename and check for directory attribute

                filename = self.parse\_filename(directory\_entry)

                is\_directory = (directory\_entry[11] & 0x10) != 0

                if is\_directory:

                    # Create directory path based on current cluster and parent directory structure

                    current\_cluster = (sector - self.first\_data\_sector) \* self.sectors\_per\_cluster + (offset // 32)

                    directory\_path = os.path.join(directory\_structure.get("", ""), filename)

                    directory\_structure.setdefault(directory\_path, {})

                else:

                    # Identified file (already categorized in follow\_cluster\_chain)

                    num\_files += 1

                offset += 32  # Move to the next directory entry

        return directory\_structure, num\_files

    def recover\_data(self):

        """

        Recovers data from the FAT16 image, including directory structure and potential files.

        """

        recovered\_data = {}  # Dictionary to store recovered cluster data (cluster number: data)

        recovered\_files = []  # List of tuples (data, filename) for recovered files

        directory\_structure, num\_files = self.get\_directory\_structure(recovered\_data)

        # Identify corruption based on directory structure inconsistencies (optional)

        # ...

        return recovered\_data, recovered\_files, directory\_structure, num\_files

    def create\_directory\_listing(self, recovered\_data, recovered\_files, directory\_structure, num\_files):

        """

        Creates the directory structure and recovered files based on the results.

        """

        os.makedirs("RecoveredFiles", exist\_ok=True)  # Create directory for recovered files

        # Write recovered files from recovered\_files list

        good\_files = {}  # Dictionary to store correctly sized files within directories

        for data, filename in recovered\_files:

            # Check if the file size matches the directory entry size (if available)

            # ... (optional - handle inconsistencies due to corruption)

            # Write the file to the appropriate location

            if filename in directory\_structure.get("", []):  # File belongs to root directory

                filepath = os.path.join("RecoveredFiles", filename)

            else:

                # Identify directory path based on directory structure

                directory\_path = ""

                for path, subdirectories in directory\_structure.items():

                    if filename in subdirectories:

                        directory\_path = path

                        break

                filepath = os.path.join("RecoveredFiles", directory\_path, filename)

            with open(filepath, "wb") as f:

                f.write(data)

            # Categorize as good file if size matches directory entry (optional)

            # ...

        # Create listing.txt file with summary report

        with open(os.path.join("RecoveredFiles", "listing.txt"), "w") as f:

            f.write(f"Number of Files Identified: {num\_files}\n")

            f.write("Recovered Directory Structure:\n")

            for directory, subdirectories in directory\_structure.items():

                f.write(f"- {directory}\n")

                if subdirectories:

                    for subdir in subdirectories:

                        f.write(f"  - {subdir}\n")

            f.write("\nRecovered Files:\n")

            for data, filename in recovered\_files:

                f.write(f"- {filename}\n")

if \_\_name\_\_ == "\_\_main\_\_":

    image\_filename =  r'C:\Users\hp\Desktop\fat16-36638740-39.img'  # Replace with your image filename

    fat = FAT(image\_filename)

    recovered\_data, recovered\_files, directory\_structure, num\_files = fat.recover\_data()

    fat.create\_directory\_listing(recovered\_data, recovered\_files, directory\_structure, num\_files)

    fat.close()

    print("Data recovery complete! Check the RecoveredFiles directory for results.")

code 2-

import os

import struct

class FATDisk:

    def \_\_init\_\_(self, disk\_image\_path):

        self.disk\_image\_path = disk\_image\_path

        self.cluster\_size = None  # To be determined from disk image

        self.fat\_entries = None   # To be populated

    def analyze\_disk(self):

        """

        Reads the disk image to determine cluster size and populate FAT entries.

        """

        try:

            with open(self.disk\_image\_path, "rb") as disk\_image:

                # Read boot sector to determine bytes per sector and sectors per cluster

                boot\_sector = disk\_image.read(512)

                bytes\_per\_sector = struct.unpack("<H", boot\_sector[0x0B:0x0D])[0]

                sectors\_per\_cluster = boot\_sector[0x0D]

                # Calculate cluster size

                self.cluster\_size = bytes\_per\_sector \* sectors\_per\_cluster

                # Read FAT table(s) based on FAT size

                # Determine FAT size based on boot sector information (e.g., number of sectors per FAT)

                # Read the FAT data from the appropriate location(s) on the disk

                # ... (Implementation details for FAT size determination and FAT table reading)

                # ...

                # Populate FAT entries (just a placeholder for now)

                self.fat\_entries = [0] \* 1000  # Example: Populate with zeros for testing

        except Exception as e:

            print(f"Error analyzing disk: {e}")

            return False

        return True

    def recover\_files(self, output\_dir):

        """

        Processes each cluster in the FAT table, recovering files and identifying directories.

        Args:

            output\_dir (str): Path to the output directory for recovered files.

        Returns:

            tuple: A tuple containing three elements:

                - recovered\_files (list): List of paths to recovered raw files.

                - unlinked\_files (list): List of paths to recovered (potentially trimmed) files not found in directories.

                - good\_files (dict): Dictionary mapping directory names to lists of file paths within those directories.

        """

        if self.fat\_entries is None:

            print("Error: FAT entries not populated. Please analyze the disk first.")

            return [], [], {}

        try:

            os.makedirs(output\_dir, exist\_ok=True)  # Create output directory if it doesn't exist

            recovered\_files = []

            unlinked\_files = []

            good\_files = {}

            for cluster\_num in range(len(self.fat\_entries)):

                file\_data = self.read\_cluster\_chain(cluster\_num)

                if file\_data:

                    file\_path = os.path.join(output\_dir, f"FILE{cluster\_num:04d}.BIN")

                    with open(file\_path, "wb") as f:

                        f.write(file\_data)

                        recovered\_files.append(file\_path)

                    # Attempt to identify file type and recover accordingly

                    if self.is\_directory(file\_data):

                        directory\_name, directory\_files = self.extract\_directory(file\_data)

                        good\_files[directory\_name] = directory\_files

                        self.recover\_directory(output\_dir, directory\_name, directory\_files)

                    elif self.is\_jpeg(file\_data):

                        self.recover\_image(output\_dir, cluster\_num, file\_data, "jpg")

                    elif self.is\_png(file\_data):

                        self.recover\_image(output\_dir, cluster\_num, file\_data, "png")

                    else:

                        # Optional: Trim recovered file by 5%

                        trimmed\_size = int(len(file\_data) \* 0.95)  # Trimmed to 95% of original size

                        trimmed\_file\_data = file\_data[:trimmed\_size]

                        # Write trimmed file

                        trimmed\_file\_path = os.path.join(output\_dir, f"FILE{cluster\_num:04d}.TXT")

                        with open(trimmed\_file\_path, "wb") as f:

                            f.write(trimmed\_file\_data)

                        unlinked\_files.append(trimmed\_file\_path)

        except Exception as e:

            print(f"Error recovering files: {e}")

        return recovered\_files, unlinked\_files, good\_files

    def recover\_directory(self, output\_dir, directory\_name, directory\_files):

        """

        Recovers a directory structure and its files.

        Args:

            output\_dir (str): Output directory.

            directory\_name (str): Name of the directory.

            directory\_files (dict): Dictionary mapping file names to file contents.

        """

        directory\_path = os.path.join(output\_dir, directory\_name)

        os.makedirs(directory\_path, exist\_ok=True)

        for filename, file\_content in directory\_files.items():

            file\_path = os.path.join(directory\_path, filename)

            with open(file\_path, "wb") as f:

                f.write(file\_content)

    def recover\_image(self, output\_dir, cluster\_num, file\_data, extension):

        """

        Recovers an image file.

        Args:

            output\_dir (str): Output directory.

            cluster\_num (int): Cluster number of the image file.

            file\_data (bytes): Image file data.

            extension (str): Image file extension (e.g., 'jpg', 'png').

        """

        file\_path = os.path.join(output\_dir, f"FILE{cluster\_num:04d}.{extension}")

        with open(file\_path, "wb") as f:

            f.write(file\_data)

    def read\_cluster\_chain(self, start\_cluster):

        """

        Reads the entire cluster chain starting from the specified cluster.

        Args:

            start\_cluster (int): The starting cluster number.

        Returns:

            bytes: The concatenated data from all clusters in the chain, or None if the chain is invalid.

        """

        if self.cluster\_size is None:

            print("Error: Cluster size not determined. Please analyze the disk first.")

            return None

        cluster\_data = b''

        cluster = start\_cluster

        try:

            while cluster != self.fat\_entries[cluster]:

                if cluster < 0 or cluster >= len(self.fat\_entries):

                    return None  # Invalid cluster chain

                cluster\_offset = cluster \* self.cluster\_size

                with open(self.disk\_image\_path, "rb") as disk\_image:

                    disk\_image.seek(cluster\_offset)

                    cluster\_data += disk\_image.read(self.cluster\_size)

                cluster = self.fat\_entries[cluster]

        except Exception as e:

            print(f"Error reading cluster chain: {e}")

            return None

        return cluster\_data

    def is\_directory(self, data):

        """

        Placeholder method to identify whether the given data represents a directory.

        Args:

            data (bytes): Data to be analyzed.

        Returns:

            bool: True if the data represents a directory, False otherwise.

        """

        # Implement directory identification logic here

        # For example, check for directory-specific markers or patterns in the data

        return False

    def extract\_directory(self, data):

        """

        Placeholder method to extract directory structure and files from the given data.

        Args:

            data (bytes): Data representing a directory.

        Returns:

            tuple: A tuple containing the directory name (str) and a dictionary mapping file names (str) to file contents (bytes).

        """

        # Implement directory extraction logic here

        # For example, parse directory entries and extract file names and contents

        directory\_name = "ExampleDirectory"

        directory\_files = {"ExampleFile.txt": b"Example file contents"}

        return directory\_name, directory\_files

    def is\_jpeg(self, data):

        """

        Placeholder method to identify whether the given data represents a JPEG image.

        Args:

            data (bytes): Data to be analyzed.

        Returns:

            bool: True if the data represents a JPEG image, False otherwise.

        """

        # Implement JPEG identification logic here

        # For example, check for JPEG file signatures (e.g., 0xFFD8FFE0)

        return False

    def is\_png(self, data):

        """

        Placeholder method to identify whether the given data represents a PNG image.

        Args:

            data (bytes): Data to be analyzed.

        Returns:

            bool: True if the data represents a PNG image, False otherwise.

        """

        # Implement PNG identification logic here

        # For example, check for PNG file signatures (e.g., 0x89504E47)

        return False

if \_\_name\_\_ == "\_\_main\_\_":

    disk\_image\_path = r'C:\Users\hp\Desktop\fat16-36638740-39.img'  # Update with the path to the FAT disk image

    output\_directory = 'RecoveredFiles'  # Update with the desired output directory

    fat\_disk = FATDisk(disk\_image\_path)

    if fat\_disk.analyze\_disk():

        recovered\_files, unlinked\_files, good\_files = fat\_disk.recover\_files(output\_directory)

        total\_recovered\_files = len(recovered\_files) + len(unlinked\_files)

        good\_files\_count = sum(len(files) for \_, files in good\_files.items())

        print(f"Summary Report:")

        print(f"- Number of files on disk: {total\_recovered\_files}")

        print(f"- Recovered Files: {total\_recovered\_files}")

        print(f"  - Recovered in directories (GoodFiles): {good\_files\_count}")

        print(f"  - Unlinked files: {len(unlinked\_files)}")

        print(f"- Potential data corruption identified (based on summary report)")

code 3- import os

import struct

import sys

class FATDisk:

    def \_\_init\_\_(self, disk\_image\_path):

        self.disk\_image\_path = disk\_image\_path

        self.cluster\_size = None  # To be determined from disk image

        self.fat\_entries = None   # To be populated

    def analyze\_disk(self):

        """

        Reads the disk image to determine cluster size and populate FAT entries.

        """

        try:

            with open(self.disk\_image\_path, "rb") as disk\_image:

                # Read boot sector to determine bytes per sector and sectors per cluster

                boot\_sector = disk\_image.read(512)

                bytes\_per\_sector = struct.unpack("<H", boot\_sector[0x0B:0x0D])[0]

                sectors\_per\_cluster = boot\_sector[0x0D]

                # Calculate cluster size

                self.cluster\_size = bytes\_per\_sector \* sectors\_per\_cluster

                # Read FAT table(s) based on FAT size

                # Determine FAT size based on boot sector information (e.g., number of sectors per FAT)

                # Read the FAT data from the appropriate location(s) on the disk

                # ... (Implementation details for FAT size determination and FAT table reading)

                # ...

                # Populate FAT entries (just a placeholder for now)

                self.fat\_entries = [0] \* 1000  # Example: Populate with zeros for testing

        except Exception as e:

            print(f"Error analyzing disk: {e}")

            return False

        return True

    def recover\_files(self, output\_dir):

        """

        Processes each cluster in the FAT table, recovering files and identifying directories.

        Args:

            output\_dir (str): Path to the output directory for recovered files.

        Returns:

            tuple: A tuple containing three elements:

                - recovered\_files (list): List of paths to recovered raw files.

                - unlinked\_files (list): List of paths to recovered (potentially trimmed) files not found in directories.

                - good\_files (dict): Dictionary mapping directory names to lists of file paths within those directories.

        """

        if self.fat\_entries is None:

            print("Error: FAT entries not populated. Please analyze the disk first.")

            return [], [], {}

        try:

            os.makedirs(output\_dir, exist\_ok=True)  # Create output directory if it doesn't exist

            recovered\_files = []

            unlinked\_files = []

            good\_files = {}

            for cluster\_num in range(len(self.fat\_entries)):

                file\_data = self.read\_cluster\_chain(cluster\_num)

                if file\_data:

                    file\_path = os.path.join(output\_dir, f"FILE{cluster\_num:04d}.BIN")

                    with open(file\_path, "wb") as f:

                        f.write(file\_data)

                        recovered\_files.append(file\_path)

                    # Attempt to identify file type and recover accordingly

                    if self.is\_directory(file\_data):

                        directory\_name, directory\_files = self.extract\_directory(file\_data)

                        good\_files[directory\_name] = directory\_files

                        self.recover\_directory(output\_dir, directory\_name, directory\_files)

                    elif self.is\_jpeg(file\_data):

                        self.recover\_image(output\_dir, cluster\_num, file\_data, "jpg")

                    elif self.is\_png(file\_data):

                        self.recover\_image(output\_dir, cluster\_num, file\_data, "png")

                    else:

                        # Optional: Trim recovered file by 5%

                        trimmed\_size = int(len(file\_data) \* 0.95)  # Trimmed to 95% of original size

                        trimmed\_file\_data = file\_data[:trimmed\_size]

                        # Write trimmed file

                        trimmed\_file\_path = os.path.join(output\_dir, f"FILE{cluster\_num:04d}.TXT")

                        with open(trimmed\_file\_path, "wb") as f:

                            f.write(trimmed\_file\_data)

                        unlinked\_files.append(trimmed\_file\_path)

        except Exception as e:

            print(f"Error recovering files: {e}")

        return recovered\_files, unlinked\_files, good\_files

    def recover\_directory(self, output\_dir, directory\_name, directory\_files):

        """

        Recovers a directory structure and its files.

        Args:

            output\_dir (str): Output directory.

            directory\_name (str): Name of the directory.

            directory\_files (dict): Dictionary mapping file names to file contents.

        """

        directory\_path = os.path.join(output\_dir, directory\_name)

        os.makedirs(directory\_path, exist\_ok=True)

        for filename, file\_content in directory\_files.items():

            file\_path = os.path.join(directory\_path, filename)

            with open(file\_path, "wb") as f:

                f.write(file\_content)

    def recover\_image(self, output\_dir, cluster\_num, file\_data, extension):

        """

        Recovers an image file.

        Args:

            output\_dir (str): Output directory.

            cluster\_num (int): Cluster number of the image file.

            file\_data (bytes): Image file data.

            extension (str): Image file extension (e.g., 'jpg', 'png').

        """

        file\_path = os.path.join(output\_dir, f"FILE{cluster\_num:04d}.{extension}")

        with open(file\_path, "wb") as f:

            f.write(file\_data)

    def read\_cluster\_chain(self, start\_cluster):

        """

        Reads the entire cluster chain starting from the specified cluster.

        Args:

            start\_cluster (int): The starting cluster number.

        Returns:

            bytes: The concatenated data from all clusters in the chain, or None if the chain is invalid.

        """

        if self.cluster\_size is None:

            print("Error: Cluster size not determined. Please analyze the disk first.")

            return None

        cluster\_data = b''

        cluster = start\_cluster

        try:

            while cluster != self.fat\_entries[cluster]:

                if cluster < 0 or cluster >= len(self.fat\_entries):

                    return None  # Invalid cluster chain

                cluster\_offset = cluster \* self.cluster\_size

                with open(self.disk\_image\_path, "rb") as disk\_image:

                    disk\_image.seek(cluster\_offset)

                    cluster\_data += disk\_image.read(self.cluster\_size)

                cluster = self.fat\_entries[cluster]

        except Exception as e:

            print(f"Error reading cluster chain: {e}")

            return None

        return cluster\_data

    def is\_directory(self, data):

        """

        Placeholder method to identify whether the given data represents a directory.

        Args:

            data (bytes): Data to be analyzed.

        Returns:

            bool: True if the data represents a directory, False otherwise.

        """

        # Implement directory identification logic here

        # For example, check for directory-specific markers or patterns in the data

        return False

    def extract\_directory(self, data):

        """

        Placeholder method to extract directory structure and files from the given data.

        Args:

            data (bytes): Data representing a directory.

        Returns:

            tuple: A tuple containing the directory name (str) and a dictionary mapping file names (str) to file contents (bytes).

        """

        # Implement directory extraction logic here

        # For example, parse directory entries and extract file names and contents

        directory\_name = "ExampleDirectory"

        directory\_files = {"ExampleFile.txt": b"Example file contents"}

        return directory\_name, directory\_files

    def is\_jpeg(self, data):

        """

        Placeholder method to identify whether the given data represents a JPEG image.

        Args:

            data (bytes): Data to be analyzed.

        Returns:

            bool: True if the data represents a JPEG image, False otherwise.

        """

        # Implement JPEG identification logic here

        # For example, check for JPEG file signatures (e.g., 0xFFD8FFE0)

        return False

    def is\_png(self, data):

        """

        Placeholder method to identify whether the given data represents a PNG image.

        Args:

            data (bytes): Data to be analyzed.

        Returns:

            bool: True if the data represents a PNG image, False otherwise.

        """

        # Implement PNG identification logic here

        # For example, check for PNG file signatures (e.g., 0x89504E47)

        return False

class FAT:

    def \_\_init\_\_(self, image\_filename):

        try:

            # Read the first sector (boot sector)

            block0 = self.get\_sector(0)

            # Unpack metadata (focus on relevant information)

            self.jmp\_boot, self.oem\_name, self.bytes\_per\_sector, self.sectors\_per\_cluster, \

            self.reserved\_sectors, self.num\_fats, self.root\_entries\_count, \

            self.total\_sectors\_16, self.media\_type, self.fat\_size\_16 = \

                struct.unpack('<3s8sHBHBHHBHHHLLL', block0[:40])

            # Calculate derived values

            self.root\_dir\_sectors = int((self.root\_entries\_count \* 32 + self.bytes\_per\_sector - 1) / self.bytes\_per\_sector)

            self.first\_data\_sector = self.reserved\_sectors + (self.num\_fats \* self.fat\_size\_16) + self.root\_dir\_sectors

            self.data\_sectors = self.total\_sectors\_16 - (self.reserved\_sectors + (self.num\_fats \* self.fat\_size\_16) + self.root\_dir\_sectors)

            self.cluster\_count = int(self.data\_sectors / self.sectors\_per\_cluster)

            self.fat\_start = self.reserved\_sectors  # Sector number of the first FAT table

        except struct.error as e:

            print(f"Error unpacking boot sector: {e}")

            sys.exit(1)

        self.image\_file = open(image\_filename, 'rb')

        self.bytes\_per\_sector = 512

    def get\_sector(self, sector\_number):

        """

        Reads a sector from the disk image.

        """

        self.image\_file.seek(sector\_number \* self.bytes\_per\_sector)

        return self.image\_file.read(self.bytes\_per\_sector)

    def close(self):

        self.image\_file.close()

    def is\_valid\_directory\_entry(self, directory\_entry):

        """

        Checks if a directory entry is valid (excluding unused and end-of-directory markers).

        """

        return (

            0 != directory\_entry[0] and  # First byte not zero (unused)

            directory\_entry[0] != 0xE5  # First byte not 0xE5 (end of directory)

        )

    def parse\_filename(self, directory\_entry):

        """

        Extracts the filename (8.3 format) from a directory entry.

        """

        filename = ""

        # Extract characters for filename (excluding spaces)

        for char\_offset in range(0, 8):

            char = directory\_entry[char\_offset]

            if char != 0x20:  # Exclude spaces

                filename += chr(char)

        # Extract characters for extension (excluding spaces)

        extension = ""

        for char\_offset in range(8, 11):

            char = directory\_entry[char\_offset]

            if char != 0x20:  # Exclude spaces

                extension += chr(char)

        # Combine filename and extension (if present)

        if extension:

            filename += "." + extension

        return filename

    def follow\_cluster\_chain(self, start\_cluster, recovered\_data, recovered\_files):

        """

        Follows a cluster chain, potentially recovering data, identifying end markers, detecting potential files,

        and performing signature-based identification for JPEG and PNG.

        """

        current\_cluster = start\_cluster

        while current\_cluster > 0 and current\_cluster < self.cluster\_count:

            # Get the FAT entry for the current cluster

            fat\_offset = (current\_cluster - 2) \* 2  # Assuming 16-bit FAT entries

            fat\_sector = self.fat\_start + int(fat\_offset / self.bytes\_per\_sector)

            fat\_entry\_offset = fat\_offset % self.bytes\_per\_sector

            fat\_data = self.get\_sector(fat\_sector)

            fat\_entry = struct.unpack('<H', fat\_data[fat\_entry\_offset:fat\_entry\_offset + 2])[0]

            # Extract data from the current cluster

            cluster\_data = self.get\_sector(self.first\_data\_sector + current\_cluster - 2)

            # Check for end-of-chain markers (0x0000 or 0xFFFF for FAT16)

            if fat\_entry in (0x0000, 0xFFFF):

                break

            # Data carving for JPEG and PNG signatures

            jpeg\_signature = b"\xFF\xD8"

            png\_signature = b"\x89PNG\r\n\x1A\n"

            for signature in [jpeg\_signature, png\_signature]:

                offset = 0

                while offset < len(cluster\_data):

                    signature\_pos = cluster\_data.find(signature, offset)

                    if signature\_pos != -1:

                        # Potential JPEG/PNG file found, extract data starting from signature

                        file\_data = cluster\_data[signature\_pos:]

                        # Continue extracting data from subsequent clusters until next signature or end-of-chain

                        next\_cluster = self.follow\_cluster\_chain(

                            current\_cluster + 1, recovered\_data, recovered\_files

                        )

                        if next\_cluster != 0xFFFF:  # Not end-of-chain

                            file\_data += recovered\_data.get(next\_cluster, b"")

                        offset = signature\_pos + len(signature)

                        # Identify potential file based on extension in directory entry

                        directory\_entry = recovered\_data.get(current\_cluster - 2, None)

                        if directory\_entry and self.is\_valid\_directory\_entry(directory\_entry):

                            filename = self.parse\_filename(directory\_entry)

                            extension = filename.split(".")[-1].lower()

                            if extension in ("jpg", "jpeg", "png", "txt", "pdf"):

                                recovered\_files.append((file\_data, filename))

                        else:

                            # Unlinked file - categorize later

                            recovered\_files.append((file\_data, f"UNLINKED\_{len(recovered\_files)}.BIN"))

                        break  # Found signature, break inner loop

                    else:

                        break  # No signature found in this cluster, move to next offset

            # Update recovered data for current cluster

            recovered\_data[current\_cluster - 2] = cluster\_data

            # Move to the next cluster in the chain

            current\_cluster = fat\_entry

        return current\_cluster  # Return the last cluster accessed

    def get\_directory\_structure(self, recovered\_data):

        """

        Reconstructs the directory structure based on valid directory entries.

        """

        directory\_structure = {}  # Dictionary to store directory paths and their subdirectories

        num\_files = 0  # Counter for identified files

        root\_sector = self.first\_data\_sector + self.root\_dir\_sectors - 1  # Last sector of the root directory

        for sector in range(self.first\_data\_sector, root\_sector + 1):

            sector\_data = self.get\_sector(sector)

            offset = 0

            while offset < len(sector\_data):

                directory\_entry = sector\_data[offset:offset + 32]  # Extract one directory entry

                if not self.is\_valid\_directory\_entry(directory\_entry):

                    break  # Reached end of directory entries or unused entries

                # Extract filename and check for directory attribute

                filename = self.parse\_filename(directory\_entry)

                is\_directory = (directory\_entry[11] & 0x10) != 0

                if is\_directory:

                    # Create directory path based on current cluster and parent directory structure

                    current\_cluster = (sector - self.first\_data\_sector) \* self.sectors\_per\_cluster + (offset // 32)

                    directory\_path = os.path.join(directory\_structure.get("", ""), filename)

                    directory\_structure.setdefault(directory\_path, {})

                else:

                    # Identified file (already categorized in follow\_cluster\_chain)

                    num\_files += 1

                offset += 32  # Move to the next directory entry

        return directory\_structure, num\_files

    def recover\_data(self):

        """

        Recovers data from the FAT16 image, including directory structure and potential files.

        """

        recovered\_data = {}  # Dictionary to store recovered cluster data (cluster number: data)

        recovered\_files = []  # List of tuples (data, filename) for recovered files

        directory\_structure, num\_files = self.get\_directory\_structure(recovered\_data)

        # Identify corruption based on directory structure inconsistencies (optional)

        # ...

        return recovered\_data, recovered\_files, directory\_structure, num\_files

    def create\_directory\_listing(self, recovered\_data, recovered\_files, directory\_structure, num\_files):

        """

        Creates the directory structure and recovered files based on the results.

        """

        os.makedirs("RecoveredFiles", exist\_ok=True)  # Create directory for recovered files

        # Write recovered files from recovered\_files list

        good\_files = {}  # Dictionary to store correctly sized files within directories

        for data, filename in recovered\_files:

            # Check if the file size matches the directory entry size (if available)

            # ... (optional - handle inconsistencies due to corruption)

            # Write the file to the appropriate location

            if filename in directory\_structure.get("", []):  # File belongs to root directory

                filepath = os.path.join("RecoveredFiles", filename)

            else:

                # Identify directory path based on directory structure

                directory\_path = ""

                for path, subdirectories in directory\_structure.items():

                    if filename in subdirectories:

                        directory\_path = path

                        break

                filepath = os.path.join("RecoveredFiles", directory\_path, filename)

            with open(filepath, "wb") as f:

                f.write(data)

            # Categorize as good file if size matches directory entry (optional)

            # ...

        # Create listing.txt file with summary report

        with open(os.path.join("RecoveredFiles", "listing.txt"), "w") as f:

            f.write(f"Number of Files Identified: {num\_files}\n")

            f.write("Recovered Directory Structure:\n")

            for directory, subdirectories in directory\_structure.items():

                f.write(f"- {directory}\n")

                if subdirectories:

                    for subdir in subdirectories:

                        f.write(f"  - {subdir}\n")

            f.write("\nRecovered Files:\n")

            for data, filename in recovered\_files:

                f.write(f"- {filename}\n")

if \_\_name\_\_ == "\_\_main\_\_":

    image\_filename = r'C:\Users\hp\Desktop\fat16-36638740-39.img'  # Replace with your image filename

    fat = FAT(image\_filename)

    recovered\_data, recovered\_files, directory\_structure, num\_files = fat.recover\_data()

    fat.create\_directory\_listing(recovered\_data, recovered\_files, directory\_structure, num\_files)

    fat.close()

    print("Data recovery complete! Check the RecoveredFiles directory for results.")