Hash Cracking Tool Report

George Ferres - 13252065
William Chaston - 13252354

Executive Summary

This project involved the development of a Python based tool designed to automate dictionary based password hash cracking, based on the case study outlined in the ICTPRG434 assessment. The tool was built to detect and crack common hash types by comparing them against a user provided dictionary file. It also includes optional case transformations (uppercase, lowercase, TitleCase) to increase the success rate and outputs all cracked matches to a structured CSV file.

The project was completed over a three week development period, with roles, goals, and responsibilities clearly defined from the start. All collaboration and project management were handled in person, with key decisions and task progress documented in weekly meeting minutes. GitHub was used for version control, following a single branch model with consistent commit practices. The tool met all core functional and user level requirements, performed reliably across test cases, and is well-documented for both users and future developers. While intentionally lightweight in scope, the program is designed to be a solid foundation for future enhancements such as multithreading, advanced hash formats, or CLI automation.

Project Overview

Introduction

This project is based on the *Dictionary Based Password Cracker* case study outlined in the assessment brief. In cybersecurity, password hashes show up constantly, in breach dumps, forensic investigations, or during red team engagements. They're irreversible by design, but given a good wordlist and enough time, weak passwords may as well be plaintext.

Why Automate It?

In the universe this tool exists in, it would seem everyone has been cracking hashes manually, checking formats, running transformations, and logging results by hand. That’s fine for a hobbyist hash cracker (of which there are many) but it’s not efficient when you’ve got thousands of hashes to work through. For this reason, the NSA contracted George Ferres and William Chaston to create a program of unimaginable complexity written in Python, a language few can even print "hello world" in. Python was decided to be the best fit out of the scripting options in the scope given to us, given our familiarity and ease of quickly prototyping/debugging. The speed of deployment is key given the 3 week deadline given to us by our employers, it seems they recently launched a tool called PRISM on what we can only assume to be vicious threat actors, and have a sudden need to crack 200,000 or so peoples hashes.

Automating the process means:

- One computer can do 100's of hours of work in minutes
- Large input files can be handled easily
- Trying uppercase/lowercase/titlecase variants can be done with ease

- Getting a structured CSV output is done instantaneously

The result is a simple, practical solution with direct applications in all sorts of settings.

Project Goals

Build a Python-based tool that:

- Prompts the user for a hash file and dictionary wordlist
- Detects the hash type automatically by length
- Optionally applies common word-case transformations
- Hashes and compares efficiently using the right algorithm
- Outputs cracked results to a CSV file and shows progress in real-time
- Keeps the interface simple enough for anyone without scripting knowledge to use

Team Structure and Roles

Given the small size of our team, responsibilities were shared flexibly, but each member took ownership of specific parts of the project to streamline progress and reduce overlap.

George

- Created and managed the GitHub repository
- Developed the main hash cracking logic
- Merged final pseudocode and python code elements into one file
- Managed meeting notes and development process documentation

Will

- Developed the dictionary file input script, including user prompts and file path validation
- Handled testing and refinement of input/output handling for cracked hashes
- Collaborated on debugging, result formatting, and verifying the program against test inputs
- Assisted with documentation planning and code cleanup

Overlapping Responsibility

- Creation of the algorithm/flowchart
- Commenting and pseudocode for the sections of code we wrote
- Contributing to the final project report

Problem Analysis and Requirements

Problem Analysis

The core challenge outlined in the case study is to streamline the process of dictionary-based password hash cracking. While the basic concept is straightforward, the actual implementation becomes increasingly complex when applied at scale or when multiple hash types and transformation options are involved.

As noted in the introduction, there is simply no existing automation for this type of task, meaning that cracking even a small set of hashes becomes a time intensive burden. Working out the correct hashing algorithm, applying multiple case variations for each dictionary entry, and tracking results

all require attention to detail and repetition, thus leaving too much room for human error and inconsistency, something our employers would not accept.

This project aims to eliminate that manual overhead. The problem is not just the volume of hashes, but the lack of tooling to manage the task start to finish in a clean, repeatable way.

Automation Requirements

To address the problem effectively, the solution must meet the following requirements:

- **Input Validation**: Prompt the user for a hash list and dictionary file path, and confirm the paths are valid before proceeding
- **Hash Type Detection**: Automatically determine the correct hashing algorithm based on hash length, supporting MD5, SHA-1, SHA-256 and SHA-512
- **Transformation Options**: Offer the ability to apply case transformations (UPPERCASE, lowercase, TitleCase) without modifying the dictionary file itself
- **Output Structure**: Record successful matches in a structured CSV format, appending to the file without overwriting existing results
- **Progress Feedback**: Provide a real-time progress indicator to help users understand the tool's activity and estimated runtime

Constraints

To stay within project scope and deadline constraints, the solution does not include support for:

- Salted hashes
- Advanced cracking techniques
- Parallel processing

It also avoids reliance on third-party APIs or external hash databases. Instead, the focus is on building a single, self contained Python tool that performs reliably and predictably for the specified use case.

Design Process

Conceptual Design

The initial approach to the problem focused on breaking the task into its core functional requirements: loading hash inputs, determining the hash type, comparing each hash to a set of dictionary words, and logging any successful matches. During our first brainstorming meeting, when constructing the algorithm we put a lot of thought into how to keep the solution as simple and reliable as possible, while still meeting all of our employers requirements.

From the outset, the concept was to design a tool that operates in a linear, user-guided flow. In where the program prompts for file paths, detects the type of each hash, applies optional transformations, and writes the output in a structured format. Rather than building a complex interface or advanced features, the focus was on getting a clean, repeatable cracking process working first. Once that structure was established, performance considerations and user quality-of-life features (like progress bars and CSV logging) were gradually introduced. The result was a tool that is both functional and maintainable, built on clear design priorities from the beginning.

Algorithm Design

The algorithm flowchart can be found in the user level documentation in the Readme.md file in the GitHub repository of this project.

```
// Authors: George Ferres, William Chaston
// We simplified some sections but core functionality is the same

// Import required standard libraries - or their equivelants
IMPORT os
IMPORT csv
IMPORT hashlib
IMPORT tqdm FROM tqdm
IMPORT Path FROM pathlib

// Declare global variables with default placeholder values
SET file_path TO "a"
SET dic_path  TO "a"
SET hash_count TO 0

// Prompt user for the hashes file path and validate it
FUNCTION whathash()
    PROMPT "Please enter the hashes file path: " AND STORE INPUT IN file_path
    IF Path(file_path) EXISTS THEN
        DISPLAY "The file path '" + file_path + "' is valid."
        SET file_name TO file_path
        SET file_extension TO Path(file_name).suffix
        DISPLAY "The file extension is: " + file_extension
        RETURN file_path
    ELSE
        DISPLAY "The file path '" + file_path + "' does not exist. Please check and
try again."
        RETURN CALL whathash()
    END IF
END FUNCTION

// Prompt user for the dictionary file path and validate it
FUNCTION whatdic()
    PROMPT "Please enter the dictionary file path: " AND STORE INPUT IN dic_path
    IF Path(dic_path) EXISTS THEN
        DISPLAY "The file path '" + dic_path + "' is valid."
        SET file_name TO dic_path
        SET file_extension TO Path(file_name).suffix
        DISPLAY "The file extension is: " + file_extension
        RETURN dic_path
    ELSE
        DISPLAY "The file path '" + dic_path + "' does not exist. Please check and
try again."
        RETURN CALL whatdic()
    END IF
```

```
END FUNCTION

// Determine suitable hash algorithm based on hash string length
FUNCTION detect_hash_type(hash_string)
    SET length TO LENGTH(hash_string)
    IF length == 32 THEN
        RETURN hashlib.md5
    ELIF length == 40 THEN
        RETURN hashlib.sha1
    ELIF length == 64 THEN
        RETURN hashlib.sha256
    ELIF length == 128 THEN
        RETURN hashlib.sha512
    ELSE
        RETURN null
    END IF
END FUNCTION

// Attempt case variation matches for a single dictionary word against target hash
FUNCTION try_variations(word, target_hash, hash_func, use_upper, use_lower,
use_title)
    INITIALISE attempts AS LIST OF (word, "original")
    IF use_upper IS TRUE THEN
        APPEND (UPPER(word), "upper") TO attempts
    END IF
    IF use_lower IS TRUE THEN
        APPEND (LOWER(word), "lower") TO attempts
    END IF
    IF use_title IS TRUE THEN
        APPEND (TITLE(word), "title") TO attempts
    END IF

    FOR EACH (variant, label) IN attempts DO
        SET word_hash TO hash_func(ENCODE(variant)).hexdigest()
        IF word_hash == target_hash THEN
            RETURN variant
        END IF
    END FOR
    RETURN null
END FUNCTION

// Compare each input hash against the dictionary to find matching word
FUNCTION check_hashes(input_list, hash_count)
    FOR EACH target_hash IN tqdm(input_list, desc="Cracking hashes") DO
        DISPLAY "Checking hash: " + target_hash
        SET hash_func TO CALL detect_hash_type(target_hash)
        IF hash_func IS null THEN
            DISPLAY "Unknown hash type"
            CONTINUE LOOP
```

```
        END IF

        SET use_upper TO use_upper_input
        SET use_lower TO use_lower_input
        SET use_title TO use_title_input

        SET found TO FALSE

        OPEN dic_path AS wordlist_file
            FOR EACH line IN wordlist_file DO
                SET word TO STRIP(line)
                SET match TO CALL try_variations(word, target_hash, hash_func,
use_upper, use_lower use_title)

                IF match IS NOT null THEN
                    DISPLAY "Match found: " + match
                    ADD 1 TO hash_count
                    OPEN "cracked_hashes.csv" IN append MODE AS csvfile
                        WRITE ROW [target_hash, match] TO csvfile
                        CLOSE csvfile
                SET found TO TRUE
                BREAK
                END IF
            END FOR
        CLOSE wordlist_file

        IF found IS FALSE THEN
            DISPLAY "No match found."
        END IF
    END FOR
    RETURN hash_count
END FUNCTION

// Program execution
SET file_path TO CALL whathash()
SET dic_path TO CALL whatdic()

// Request variation options from user
PROMPT "Check UPPERCASE versions? (y/n): " AND LOWER AND STORE BOOLEAN RESULT IN
use_upper_input
PROMPT "Check lowercase versions? (y/n): " AND LOWER AND STORE BOOLEAN RESULT IN
use_lower_input
PROMPT "Check TitleCase versions? (y/n): " AND LOWER AND STORE BOOLEAN RESULT IN
use_title_input

OPEN file_path AS input_file
    SET input_list TO LIST OF STRIP(line) FOR EACH line IN input_file

SET hash_count TO CALL check_hashes(input_list, hash_count)
```

```
DISPLAY "Cracked Hashes: " + hash_count
DISPLAY "Cracked Hash File: cracked_hashes.csv"
```

Software Design

Python was chosen as the implementation language due to its simplicity, portability, and suitability for rapid prototyping. Among the approved scripting options in scope (Python, Bash, PowerShell), Python provided the best balance of readability, string handling, and access to key built-in functions like cryptographic hashing via the `hashlib` library and progress bar via the `tdqm` library. Its cross-platform compatibility and minimal setup requirements also made it ideal for our contractors fast paced environment, where ease of use and fast development were priorities.

The software was designed with modularity and clarity in mind. Each major function such as input validation, hash detection, variation testing, and result logging is implemented independently, thus making the code easier to test, maintain, and extend. The program includes input checking, a simple user interface, and progress feedback, all aimed at improving reliability and user experience. A detailed breakdown of the program's internal structure and function responsibilities is provided in the developer level documentation.

Implementation

Code Development

Some sections did get reasonably more complex than we initially intended - We are happy to walk you through the program.

```
# Authors:
#   William Chaston
#   George Ferres
# --------------------------------

import os
import csv
import hashlib
from tqdm import tqdm
from pathlib import Path

file_path = 'a'
dic_path  = 'a'
hash_count = 0

def whathash(file_path): # Prompts the user to input the path to the file containing
hashed passwords
    file_path = input("Please enter the hashes file path: ")

    path = Path(file_path)
    if path.exists():
        print(f"The file path '{file_path}' is valid.")
        file_name = file_path
        file_extension = Path(file_name).suffix
        print(f"The file extension is: {file_extension}")
        return file_path
    else:
```

```
        print(f"The file path '{file_path}' does not exist. Please check and try
again.")
        whathash(file_path)

def whatdic(dic_path): # Prompts the user to input the path to the dictionary file
    dic_path = input("Please enter the dictionary file path: ")

    path = Path(dic_path)
    if path.exists():
        print(f"The file path '{dic_path}' is valid.")
        file_name = dic_path
        file_extension = Path(file_name).suffix
        print(f"The file extension is: {file_extension}")
        return dic_path
    else:
        print(f"The file path '{dic_path}' does not exist. Please check and try
again.")
        whatdic(dic_path)

def detect_hash_type(hash_string): # Detects the type of a hash based on its
character length
    length = len(hash_string)
    if length == 32:
        return hashlib.md5
    elif length == 40:
        return hashlib.sha1
    elif length == 64:
        return hashlib.sha256
    elif length == 128:
        return hashlib.sha512
    else:
        return None

def try_variations(word, target_hash, hash_func, use_upper, use_lower, use_title): #
Attempts to match various case variants of a word to a target hash
    attempts = [(word, 'original')] # Initialises the list of variations to try,
starting with the plain attempt e.g 'password', 'original'
    if use_upper: # Check if use_upper is True from the beginning of the program
        attempts.append((word.upper(), 'upper')) # Each variation is added to the
attempts list, with the identifier of what the variation is e.g 'PASSWORD', 'upper'
    if use_lower:
        attempts.append((word.lower(), 'lower'))
    if use_title:
        attempts.append((word.title(), 'title'))

    for variant, label in attempts:
        word_hash = hash_func(variant.encode()).hexdigest() # Hash variant
        if word_hash == target_hash: # Check variant against target hash
            return variant # Return the matching variant
    return None

def checkHashes(input_list, hash_count): # Compares each hash in the input list
against words in the dictionary file
    for target_hash in tqdm(input_list, desc="Cracking hashes"): # Initialises
progress bar
        print(f"\nChecking hash: {target_hash}")
        hash_func = detect_hash_type(target_hash)
```

```
        if not hash_func:
            print("Unknown hash type")
            continue

        # Reset variation flags for this hash
        use_upper = use_upper_input
        use_lower = use_lower_input
        use_title = use_title_input

        found = False

        # Open the dictionary file and iterate through each word
        with open(dic_path, "r", encoding="latin-1") as wordlist_file: # "latin-1" is
needed because rockyou has non UTF-8 characters
            for line in wordlist_file:
                word = line.strip()
                match = try_variations(word, target_hash, hash_func,
                                       use_upper, use_lower, use_title)
                if match:
                    print(f"Match found: {match}")
                    hash_count += 1 # Keep track of number of cracked hashes
                    # Append the cracked pair to CSV
                    with open("cracked_hashes.csv", "a", newline="") as csvfile:
                        writer = csv.writer(csvfile)
                        writer.writerow([target_hash, match])
                    found = True
                    break

        if not found:
            print("No match found.")
    return hash_count

# Prompt the user for file paths
file_path = whathash(file_path)
dic_path  = whatdic(dic_path)

# Prompt for variations to test
use_upper_input = input("Check UPPERCASE versions? (y/n): ").lower() == 'y' # "==
'y'" turn the answer into a boolean True/False statement
use_lower_input = input("Check lowercase versions? (y/n): ").lower() == 'y'
use_title_input = input("Check TitleCase versions? (y/n): ").lower() == 'y'

# Load hash list from the path entered by user
with open(file_path, "r") as input_file:
    input_list = [line.strip() for line in input_file]  # Gets each line of hash file

# Run the cracker
hash_count = checkHashes(input_list, hash_count)
print(f"Cracked Hashes: {hash_count}\nCracked Hash File: cracked_hashes.csv")
```

Testing and Debugging

Given the small scale of the project and the need to rapidly deliver a working application, the testing process focused primarily on functional validation rather than formal unit testing. The majority of debugging was performed during development using `print()` statements to inspect variable values, control flow, and identify where logic failed to produce expected results. This

approach helped track issues like incorrect hash-function selection, wordlist processing errors, or unexpected user input handling.

Test Cases and Coverage

Testing was done manually using a set of sample input files that included:

- Hash lists with known plaintext values (for MD5, SHA-1, SHA-256, and SHA-512)
- The `rockyou.txt` dictionary file
- Files with deliberate formatting issues (extra spaces, blank lines, unsupported hash lengths)

Test scenarios included:

- Confirming each supported hash length maps to the correct algorithm
- Verifying correct handling of UPPERCASE, lowercase, and TitleCase variations
- Ensuring the program exits or loops appropriately on invalid file paths
- Checking that output is correctly written to `cracked_hashes.csv` and does not overwrite existing results
- Verifying that the progress bar reflects actual cracking progress

Debugging Notes

Key bugs encountered during development included:

- Incorrect case transformation logic causing missed matches
- Output not appending correctly to the CSV file on some runs
- Handling of encoding issues with non-UTF8 characters in the dictionary file

These were addressed iteratively using print based debugging, followed by re-running the full cracking loop with controlled inputs to confirm fixes. While no formal testing framework was used, coverage of all major functional elements was ensured through structured manual testing and intentional misuse scenarios.

Collaboration and Project Management

Meeting Notes

Week 1 Review Meeting

Date: 4 June 2025
People: George & Will

What we did
- Create pre-code algorithm together
- Created repo and added a README with algorithm
- Decided on using Python programming language
- Researched Python libraries needed for program functionality
- Assigned our roles on GitHub (George = Owner, Will = Collaborator)
- Agreed on simple workflow : push straight to main with tidy commit messages
- Mapped the next two weeks:
  - Write the core Python in Week 2
  - Polish and add pseudocode in Week 3
  - Documentation throughout and report in late Week 3, outside of class

To-dos
- Will: scaffold the input-file prompts script
- George: draft the main cracking script structure

Week 2 Review Meeting

Date: 11 June 2025
People: George & Will

What we did

- Will wrote the input file handling script
- George wrote the main cracking logic
- Both starting rough notes for user guide - not pushed to repo yet
- Tested core functionality - everything cracking quickly and inputs are working
- Confirmed commit messages are clear and regular
- All feedback still being done face-to-face after each push

GitHub protocol check-in

- Still pushing straight to main, no branches
- Good commit hygiene - messages are short and descriptive
- Both contributors on 2FA, with appropriate repo roles
- No version control issues so far

To-dos

- Will: Expand input file checks to catch edge cases
- George: Writing main cracking logic
- George: Clean up variable names and add hash count to output
- Both: Testing functionality as we go
- Both: Providing feedback on each others code as we work together in class

Week 3 Review Meeting

Date: 18 June 2025
People: George & Will

What we did

```
- Wrote Pseudocode to describe the cracking logic and input handling
- Cleaned up comments in program
- Finalised local documentation drafts - not yet pushed to the repo
- Reviewed GitHub activity and captured screenshots for assessment
- Reviewed reporting requirements for assessment

#### GitHub protocol check-in

- Workflow kept simple: all commits to main, no merge conflict
- Good tracking of changes via commit history
- Feedback loop has stayed as a face-to-face protocol
- Repo is now public, both users on 2FA, permissions still appropriate


#### To-dos

- George: Finalise and push README example section
- Both: Push documentation files to GitHub and prep submission portfolio
- Both: Start on delegated sections of report
```

Project Management Tools and Techniques

Overview of the tools and methodologies used for project management and collaboration.

This project was managed using a practical, face-to-face collaboration model, supported by version control via GitHub. Rather than using formal task tracking tools or online collaboration platforms, all project planning and coordination was handled through in person discussions during class, with key decisions and task tracking recorded in meeting minutes.

Weekly review meetings were held across the three week development sprint. During these meetings, the team (George and Will) discussed progress, assigned new tasks, and made design and implementation decisions. Notes from these sessions were maintained as part of the project documentation.

All feedback on code, functionality, and structure was delivered face to face, either during these weekly meetings or during collaborative work sessions. This informal but effective communication style allowed the team to work quickly without the overhead of more complex tooling.

Version Control

GitHub was used to manage the project repository. A single-branch model was adopted for simplicity, in where all contributions were pushed directly to the `main` branch with descriptive, tidy commit messages.

While there were no formal pull requests or branching strategies, the team followed consistent version control practices:

- Commits with clear messages
- No merge conflicts due to working on features in our own files
- Regular code reviews conducted in person after each push

This approach allowed for fast iteration and clear visibility of progress, while maintaining a clean and well documented repository history.

Documentation

Developer Documentation

Code Structure

Main Components

- User Input Functions
  - `whathash()` and `whatdic()` prompt the user to enter the file paths for the hash list and the dictionary, respectively
  - They both check whether the given path is valid. If not, the function calls itself again until a valid path is provided
- Hash Type Detection
  - `detect_hash_type(hash_string)` determines which hash function to use based on the length of the hash string
  - It returns the corresponding `hashlib` function for MD5 (32 chars), SHA1 (40), SHA256 (64), or SHA512 (128). If the length does not match any of these, it returns `None`
- Variation Matching
  - `try_variations()` takes a single word and attempts to hash and compare different case variants (original, uppercase, lowercase, title case) with a target hash
  - If any variant matches, it returns the matching word; otherwise, it returns `None`
- Main Cracking Loop
  - `checkHashes()` iterates over each hash in the input list
  - For each hash, it uses the dictionary file to try variations of every word until it finds a match or exhausts the list
  - Progress is shown using `tqdm` library
  - Matching results are appended to `cracked_hashes.csv` in the format `[hash, plaintext]`
- Main Execution Block
  - Prompts the user for file paths and case variation options
  - Reads the input hash file into a list
  - Calls the main cracking function
  - Outputs the number of cracked hashes at the end

Key Files and Outputs

- Input
  - A plain text file containing hashes (one per line)
    - 12 Files to test provided in the repository
  - A dictionary file of possible passwords (one per line)
    - Intended to be used with rockyou.txt but any similar wordlists should work
- Output
  - Cracked hashes are stored in `cracked_hashes.csv`
  - Console output includes match status and overall progress via progress bar

Dependencies

- `hashlib`: Used for hashing (standard library)

- `csv`: Used to write results to a CSV file (standard library)
- `pathlib`: Used to check and manipulate file paths (standard library)
- `tqdm`: Displays a progress bar. Install via `pip install tqdm` if not already available

Implementation Notes

- The script reads dictionary files using `latin-1` encoding
  - This is necessary for compatibility with common password lists like `rockyou.txt`, which include extended characters
- File validation is recursive, ensuring the program cannot continue with invalid paths
- The program does not support salted hashes or custom formats

User Documentation

Overview

This tool is designed to crack password hashes using a dictionary attack. It supports common hash types (MD5, SHA-1, SHA-256, SHA-512) and allows users to apply case variations to increase the chances of a successful match.

Requirements

- Python 3.8 or newer
- tqdm package installed (used for progress display)

To install `tqdm`, run:

```
`pip install tqdm`
```

How to Use

Step 1: Prepare Input Files

You will need:

- A hash list file (`.txt`) with one hash per line
- A dictionary file (`.txt`) with one word per line (e.g. `rockyou.txt`)

Step 2: Run the Program

From a terminal or command prompt, navigate to the folder containing the script and run:

```
python main.py
```

Step 3: Follow the Prompts

The program will ask you for:

1. The path to your hash file
   - Example: `C:\Users\YourName\Documents\hashes.txt`

2. The path to your dictionary file
    - Example: `C:\Users\YourName\Documents\rockyou.txt`
3. Whether to check **UPPERCASE** versions of each dictionary word (`y` or `n`)
4. Whether to check **lowercase** versions (`y` or `n`)
5. Whether to check **TitleCase** versions (`y` or `n`)

Once options are selected, the cracking process begins. A progress bar will display progress in real time.

Output

The program outputs results to a file called: `cracked_hashes.csv`

Each line in the file contains: `<original_hash>,<matching_password>`

- For example: `5f4dcc3b5aa765d61d8327deb882cf99,password`

The CSV file will be updated each time the program finds a match, and results are appended, not overwritten.

Notes and Tips

- The program identifies hash types automatically based on length
- Dictionary files must be in plain text and encoded in a compatible format (the script reads using `"latin-1"` to support extended characters)
- If a file path is incorrect or cannot be read, you will be prompted to re-enter it
- If a hash does not match any dictionary entry, it will be skipped, and you will be notified in the terminal output

Example Session

```
Please enter the hashes file path: hashes.txt
The file path 'hashes.txt' is valid.
The file extension is: .txt

Please enter the dictionary file path: rockyou.txt
The file path 'rockyou.txt' is valid.
The file extension is: .txt

Check UPPERCASE versions? (y/n): y
Check lowercase versions? (y/n): y
Check TitleCase versions? (y/n): n

Cracking hashes: 100%|████████████████████████████████████| 5/5 [00:01<00:00,
4.91it/s]
Match found: password

No match found.

Match found: 123456
```

```
Cracked Hashes: 3
Cracked Hash File: cracked_hashes.csv
```

Reflection and Conclusion

One of the main challenges in this project was managing time effectively while building a working, user friendly tool that met both functional and documentation requirements. We kept our workflow simple, with in person check-ins and a weekly plan that allowed us to focus on one key stage at a time. Debugging was done mostly through print statements during development, which proved sufficient for spotting logical issues and tracking variable states. Input handling required particular attention especially around encoding and malformed files, but we resolved these by explicitly setting the encoding to `"latin-1"` and building in basic validation.

Throughout the project, we improved our skills in Python scripting, using library's and methods we had not yet explored, and working with real world data formats. We also developed collaborative habits around version control, using GitHub with consistent commit practices and live feedback. The tool met all defined objectives, including automatic hash type detection, case transformation options, structured CSV output, and live progress feedback. If extended in future, the project could hugely benefit from multithreading, particularly if working with large sets of hashes, and even support for handling of more advanced hash formats. Overall, the project delivered a solid, functional result, now to turn it over to our trustworthy employer.