

The creation of a password auditing tool utilising ASCON implementation

Drew Wandless (c2034671)

I195 BSc Computer Science
(Cyber Security)

Word Count: 14915

Supervisor: Carlton Shepherd

Abstract

Lightweight cryptographic algorithms such as ASCON are increasingly relevant for use in resource-constrained environments. However, support for these algorithms remains limited in the domain of password auditing tools. This project addresses this gap by developing a proof-of-concept password auditing tool that integrates ASCON's hashing functionality. Following the waterfall model, the development process involved conducting background research, establishing system requirements, designing the system as well as implementing and testing the tool. The resulting tool successfully meets the outlined objectives and demonstrates the feasibility of incorporating such lightweight cryptosystems into password auditing applications. However, while implementation was achievable, doing so without introducing performance overheads presents a separate challenge. As such, there remains considerable room for optimisation, with several areas identified for future improvement and extended functionality.

Declaration

I hereby declare that the work conducted within this report presents my own work and findings, unless otherwise mentioned.

Acknowledgements

I would like to express my sincere gratitude to the University of Newcastle's Computer Science Department, and to my supervisor, Carlton Shepherd, for their invaluable support, guidance, and encouragement throughout the duration of this project. I am also deeply thankful to my parents for their unwavering support and inspiration, not only during this project but throughout my entire university journey.

Table of Contents

Chapter 1 – Introduction.....	1
1.1 – Introduction.....	1
1.2 – Purpose & Motivation.....	1
1.3 – Aims & Objectives.....	2
1.3.1 – <i>Aim</i>	2
1.3.2 – <i>Objectives</i>	2
1.4 – Project Structure & Ethical Considerations.....	3
1.4.1 – <i>Report Structure</i>	3
1.4.2 – <i>Project Plan Explanation</i>	4
1.4.3 – <i>Ethical Considerations</i>	5
Chapter 2 – Related Work.....	6
2.1 – Introduction.....	6
2.2 – Understanding Brute-Force.....	6
2.3 – Similar Systems.....	7
2.3.1 – <i>John the Ripper</i>	7
2.3.2 – <i>Hashcat</i>	8
2.3.3 – <i>Benchmarks</i>	8
2.4 – Understanding Password Strength.....	9
2.4.1 – <i>Shannon Password Entropy</i>	9
2.4.2 – <i>Password Quality Indicator</i>	10
2.4.3 – <i>Zxcvbn Rating</i>	11
2.5 – ASCON Hashing Functions.....	11
2.6 – Summary.....	13
Chapter 3 – Methodology.....	14
3.1 – Introduction.....	14
3.2 – Design Requirements.....	14
3.2.1 – <i>Functional Design Requirements</i>	14
3.2.2 – <i>Non-Functional Design Requirements</i>	16
3.3 – Logic Flowchart.....	17
3.4 – User Interface Designs.....	18
3.4.1 – <i>Dashboard Page Wireframe</i>	18
3.4.2 – <i>Run Page Wireframe</i>	19

3.4.3 – Results Page Wireframe.....	20
3.4.4 – Log Page Wireframe.....	21
3.4.5 – Help Page Wireframe.....	22
3.5 – Tools & Technologies Used.....	23
3.5.1 – Python.....	23
3.5.2 – Custom Tkinter.....	23
3.5.3 – Hashlib.....	24
3.5.4 – Matplotlib.....	25
3.5.5 – Pandas.....	26
3.5.6 – Zxcvbn Implementation.....	27
3.5.7 – ASCON Implementation.....	28
3.6 – Development.....	29
3.6.1 – User Inputs & Uploading Files.....	29
3.6.2 – Wordlist Audit.....	30
3.6.3 – Brute Force Audit.....	31
3.6.4 – Hash Types.....	32
3.6.5 – System Utilisation Logging.....	32
3.6.6 – Results Visualisations.....	33
3.7 – Testing.....	38
3.7.1 – Hash String Function.....	38
3.7.2 – Wordlist Function.....	39
3.7.3 – Brute force Function.....	40
3.8 – Optimisation	40
3.8.1 – Threading & Multiprocessing.....	40
3.8.2 – Implementing Additional Hashes.....	41
3.8.3 – Performance Improvement.....	42
3.9 – Summary.....	42
Chapter 4 – Results & Evaluation	43
4.1 – Introduction.....	43
4.2 – Design Methodology.....	43
4.3 – Datasets.....	44
4.3.1 – Rockyou.txt.....	44
4.3.2 – SecLists Xato-net	45
4.3.3 – Pwned Password Top 100k List	45
4.3.4 – Dataset Summary.....	46

4.4 – Performance Test Results.....	47
4.4.1 – Hash String Performance Results.....	47
4.4.2 – Wordlist Performance Results.....	47
4.4.3 – Brute force Performance Results.....	48
4.4.4 – Optimised Function Results.....	49
4.4.5 – Optimisation Summary.....	50
4.5 – Requirement Fulfilment.....	53
4.5.1 – Functional Requirement Fulfilment.....	53
4.5.2 – Non-Functional Requirement Fulfilment.....	54
4.6 – Limitations.....	55
Chapter 5 – Discussion & Conclusion.....	56
5.1 – Introduction.....	56
5.2 – Aim & Objective Fulfilment.....	56
5.2.1 – Objective 1.....	56
5.2.2 – Objective 2.....	56
5.2.3 – Objective 3.....	57
5.2.4 – Objective 4.....	57
5.3 – Personal Reflection.....	58
5.4 – Future Work.....	59
5.5 – Summary.....	60

Table of Figures

Figure 1 – Project Gantt Chart.....	5
Figure 2 – Benchmark Hash Rate Table.....	9
Figure 3 – ASCON Hash-256 Flowchart.....	12
Figure 4 – ASCON XO/CXOF Flowchart.....	13
Figure 5 – Program Logic Flowchart.....	17
Figure 6 – Dashboard Page Wireframe.....	18
Figure 7 – Run Page Wireframe.....	19
Figure 8 – Results Page Wireframe.....	20
Figure 9 – log Page Wireframe.....	21
Figure 10 – Help Page Wireframe.....	22

Figure 11 – Custom Tkinter Code Snippet.....	24
Figure 12 – Hashlib Code Snippet.....	25
Figure 13 – Matplotlib Code Snippet.....	26
Figure 14 – Pandas Code Snippet.....	27
Figure 15 – Zxcvbn Code Snippet.....	27
Figure 16 – ASCON Code Snippet.....	28
Figure 17 – Run page Screenshot.....	29
Figure 18 – Wordlist Function Code Snippet.....	30
Figure 19 – Brute Force Function Code Snippet.....	31
Figure 20 – Utilisation Logging Function Code Snippet.....	33
Figure 21 – Character Frequency Graph Screenshot.....	34
Figure 22 – Password Quality Indicator Graph Screenshot.....	35
Figure 23 – Zxcvbn Graph Screenshot.....	35
Figure 24 – Utilisation Graph Screenshot.....	36
Figure 25 – Password Length Distribution Screenshot.....	37
Figure 26 – Success Rate Pie Chart Screenshot.....	37
Figure 27 – Hash String Functional Testing Results.....	39
Figure 28 – Multiprocessing Incorporation Screenshot.....	41
Figure 29 – Newly added Hashes Wordlist Test Results.....	42
Figure 30 – Newly added Hashes Brute Force Test Results.....	42
Figure 31 – Rockyou.txt Performance Test Results.....	44
Figure 32 – Xato-net Performance Test Results.....	45
Figure 33 – Pwned 100k Performance Test Results.....	45
Figure 34 – Password List Comparison Graph.....	46
Figure 35 – Hash String Performance Test Results.....	47
Figure 36 – Wordlist Performance Test Results.....	48
Figure 37 – Brute Force Performance Test Results.....	48
Figure 38 – Optimised Wordlist Performance Test Results.....	49
Figure 39 – Optimised Brute Force Performance Test Results.....	49
Figure 40 – Wordlist Hash Rate Improvement Graph.....	50
Figure 41 – Brute Force Hash Rate Improvement Graph.....	50
Figure 42 – CPU Utilisation Graph.....	51
Figure 43 – Memory Complexity Graph.....	52

Chapter 1 – Introduction

1.1 – Introduction

Within this chapter is the introduction to the project. This will include an explanation of the purpose and motivation behind the project, the definition of the aim and objectives that the project should achieve, as well as any other relevant information such as the structure of the project as well as that of this report.

1.2 – Purpose & Motivation

In recent years, both demand and reliance on lightweight cryptosystems for resource constrained devices has significantly increased (McKay et al., 2017). With this widespread adoption comes with the regard for their security. One prominent system in the field of lightweight encryption is ASCON (Dobraunig et al., 2019), already recognised by the National Institute of Standards and Technology (NIST) for its efficiency and ease of implementation (Computer Security Division, 2023). This project aims at integrating ASCON into a password auditing tool to evaluate its performance. Additionally, the tool will serve as an educational resource.

More specifically ASCON is *“a family of lightweight cryptographic algorithms designed for efficiency and ease of implementation”* (Eichlseder, 2019). And In 2023, it was selected as NIST’s standard for lightweight encryption and hashing, marking a significant milestone in the adoption of streamlined cryptographic solutions. While most modern cryptosystems are becoming *“increasingly complex”* to meet the demand for *“quantum-safe cryptography”* (NCSC, 2023), ASCON stands out with its lightweight nature. It is highly efficient in terms of computational power and memory usage, making it ideal for resource-constrained devices. This efficiency is due to its construction, which *“relies on standardised and well-analysed primitives”* (Dobraunig et al, 2021), including substitution, permutation, and bit-interleaving layers.

The recent emergence of ASCON and lightweight cryptography raises important questions about their resilience. As Kodirov et al. (2024) highlight, *“addressing vulnerabilities in lightweight cryptosystems is crucial.”* Because of this ASCON’s behaviour in practical environments remains underexplored. This project therefore seeks to integrate ASCON into a password auditing tool, to observe and study its operation and characteristics in a real-world scenario.

To achieve this, the project aim's in developing a graphical-based password auditing tool from the ground up, ensuring full customisation, flexibility, and creative control. The tool will provide data-driven insights through visualisations that analyse password strength, character frequency distributions, and common password patterns. This approach will enhance the evaluation of such cryptosystems while differentiating the tool from already existing solutions.

By integrating ASCON into this system, the project aims to support both users who rely on it for encryption and security researchers interested in its practical behaviour. While much of the existing research focuses on ASCON's theoretical security and performance, this project takes a more applied approach by incorporating ASCON into the password auditing tool. This integration is intended to facilitate exploration and guide further research on ASCON and its real-world use.

1.3 – Aims & Objectives

1.3.1 – Aim

The overall aim of the project is to develop a password auditing tool from the ground up that supports the integration of the ASCON hashing function (on top of pre-existing broken hashing methods), specifically a tool that makes use of brute force algorithms and password lists then compares them against a target hash. Furthermore, and uniquely the auditing tool will make use of data visualisation techniques via a graphical user interface (GUI) which will provide a data-driven approach to password security analysis.

1.3.2 – Objectives

- 1) Integrate ASCON's hashing functionality into the auditing tool to assess its performance compared to traditional hashing algorithms – This involves implementing both wordlist and brute-force attack methods, along with GUI-based visualisations to display key metrics such as success rate, password length, character patterns, password strength, execution speed, hash rate, CPU usage, and memory consumption. The primary focus is on integrating ASCON and studying how it performs in real-world scenarios (by using said visuals), particularly in comparison to established (and already broken) hashing algorithms under identical attack conditions.

- 2) Perform testing and debugging on the password auditing tool itself to ensure its accuracy, efficiency, and reliability – This includes unit testing, performance profiling, and debugging to resolve at least 90% of critical issues prior to deployment. Testing will verify that all measurements (including those related to ASCON) are correctly captured and displayed, and that the tool behaves consistently across different hashing algorithms. As ensuring the tool's internal reliability is essential for drawing meaningful insights from the data it produces.
- 3) Analyse and compare two examples of already existing password auditing tools against my tool – Aim to functionally analyse and conduct equivalent testing as done on my own tool (see objective two) for two existing password auditing tools, John the Ripper (Openwall, 2019) and Hashcat (Hashcat, 2018), to understand their features which make them standardised within the industry. By studying available documentation, testing both tools, and analysing their strengths and weaknesses, I will understand said features that can inform the development of my own tool. This will additionally help me establish a benchmark for comparative evaluation and testing using both measurable performance metrics and qualitative assessments.
- 4) Produce a comprehensive documentation report that serves both as a user guide and a final analysis of the project – This report will summarise and evaluate the effectiveness of the password auditing tool, offering clear, detailed instructions for users and valuable insights. It will also critically assess the tool's functionality, serving as a key evaluation of the project's success. Beyond this the document will be an essential portfolio piece for me (as a personal objective). Designed to be both an educational resource and a testament to my work, it will hopefully support future development within the industry while highlighting my contributions to the field.

1.4 – Project Structure & Ethical Considerations

1.4.1 – Report Structure

This Report is divided into 5 main chapters. Chapter One provides insight into the motivation and rationale behind the project, offering perspective on its significance. Additionally, it serves as a reference point for evaluating the project's success by addressing its aims and objectives that should be met.

Chapter Two is particularly important as it presents an all-encompassing review of existing relevant research that has already been conducted. It explores thematically similar studies and identifies gaps in knowledge that this project seeks to address.

Chapter Three outlines the methodology, focusing on the design, development and testing processes of the waterfall methodology utilised and seen in figure 1. Therefore, this chapter is divided into three sections. The first, the Design section, documents the processes undertaken before implementation and highlights key considerations made during development. The second section, Implementation, details the process of building the system, including a consideration of the tools used and coding practices employed. The final section, Testing, covers the types of tests conducted, their execution, an analysis of the results, and the optimisations made from said tests.

Chapter Four expands on the findings from the testing phase, analysing the insights gained and considering their implications. It also incorporates evaluation of the methodology employed, testing results, and requirements met. These findings contribute to the overall evaluation of the project.

Chapter Five serves as a conclusive chapter, offering a reflection on the project. It discusses what aspects were successful, whether the original aim was achieved and potential areas for improvement. Additionally, it explores opportunities for future research and development that could build upon this work.

1.4.2 – Project Plan Explanation

The structure of this report is closely linked to the project plan (figure 1) itself. As previously mentioned, the project follows the waterfall methodology, which organises tasks sequentially into distinct stages: planning, design, implementation, and testing. This approach is well-suited for the project, as each stage builds upon the previous one, allowing for effective compartmentalisation of tasks.

Chapters One and Two correspond to the planning stage, while Chapter Three encompasses the design, implementation, and testing stages. However, the report of this project deviates slightly from the original plan. In the plan, both implementation and testing stages were divided into front-end and back-end development, whereas in practice, these stages were more interconnected within the report.

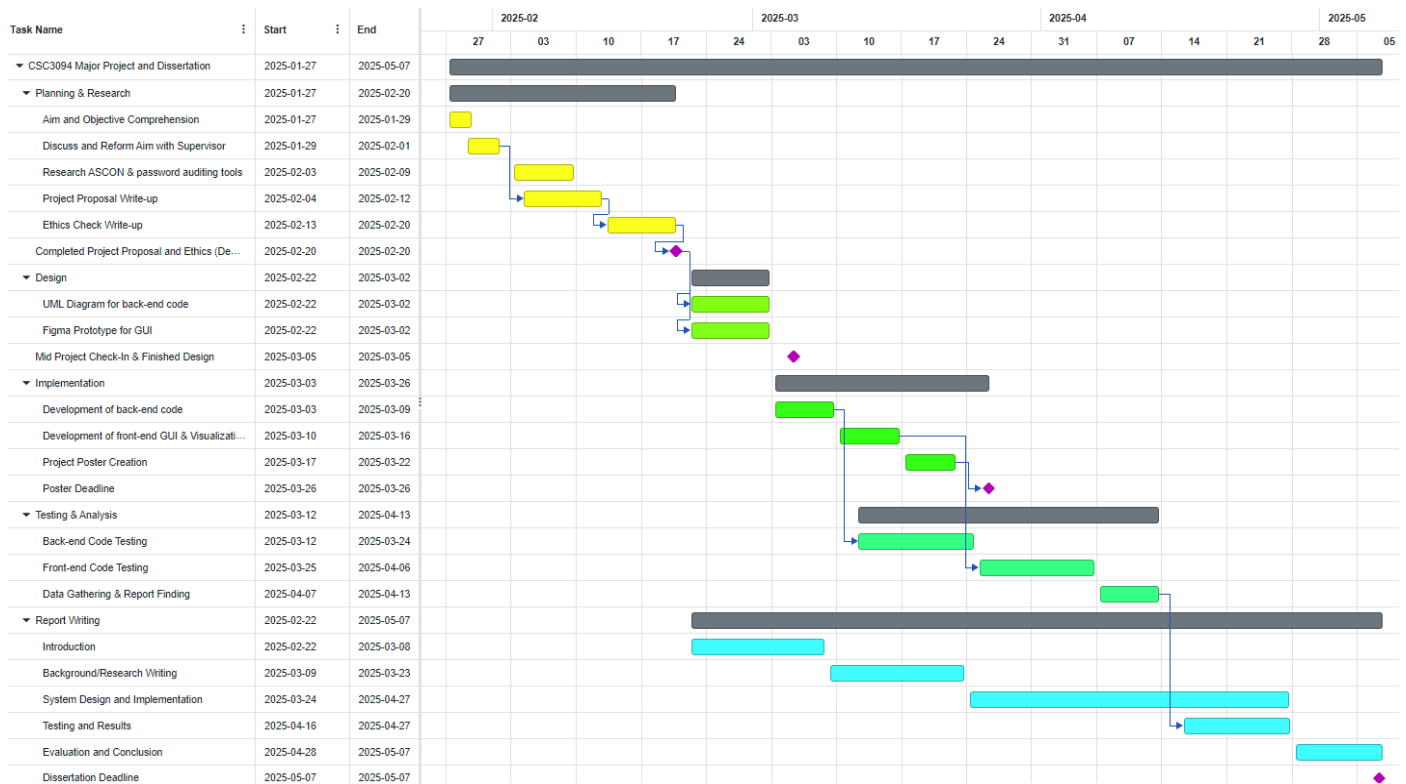


Figure 1 – A Gantt chart illustrating the project timeline, structured according to the waterfall methodology.

1.4.3 – Ethical Considerations

The development of any cybersecurity tool demands careful ethical reflection, not only to recognise the potential for misuse, but also to evaluate whether the tool’s intended benefits justify its creation. In this case, the tool is designed for educational and research purposes, specifically to support the exploration of lightweight cryptography such as ASCON. Its potential to contribute to the academic field is deemed to outweigh the risks of improper use.

Although the tool will be publicly accessible, its development was guided by standards from organisations such as NIST, to ensure responsible design and adherence to ethical expectations. The use of the tool should remain strictly within research and educational settings, and it must only be deployed with explicit consent on designated test environments. Any unauthorised or malicious use is explicitly discouraged and is contrary to the project’s aim.

Importantly to note, this project does not attempt to attack or compromise ASCON. Instead, it aims to complement the ongoing research by integrating ASCON into a practical auditing tool, allowing its behaviour to be observed in controlled, ethical scenarios. All analysis will be carried out objectively, with full respect for the expertise behind ASCON’s development.

Chapter 2 – Related Work

2.1 – Introduction

This chapter serves as a guide to help define the key concepts relevant to this project. It not only clarifies but also highlights their advantages and importance within the context of the work. It begins with an overview of password audits, examining their mechanics, variations, as well as the systems that employ such methodologies. This is followed by an analysis of password strength, the factors that influence it and the ways of measuring it. Finally, the chapter introduces ASCON, with a specific focus on its hashing functionality, which plays a central role in the project's design.

2.2 – Understanding Brute Force

As defined by The MITRE Corporation (2017), brute force is a method where *"adversaries may use to gain access to accounts when passwords are unknown, but their hashes have been obtained"*, typically achieved by *"systematically guessing passwords using a repetitive or iterative mechanism."* Brute-force audits work by taking advantage of large amounts of computational resources to guess weak or poorly chosen passwords. In today's digital landscape, where high-performance computing resources are widely accessible, such methods have become increasingly practical and efficient.

Given the range of methods available to facilitate password guessing through iterative attempts, brute-force audits can be subdivided into several sub-techniques. The MITRE Corporation (2017) identifies four major sub-techniques: *"Password Guessing, Password Cracking, Password Spraying, and Credential Stuffing"*. This project primarily focuses on *"password cracking"*, with supplementary use of dictionary attacks to enhance efficiency.

"Password cracking" represents a systematic and targeted form of brute-force audit. It typically involves obtaining password hashes from a credential leak and attempting to break them in an external environment. As noted by The MITRE Corporation (2017), *"This process is usually done on adversary-controlled systems, outside of a target network."* This approach is particularly effective, as adversaries are not hindered by typical security mechanisms such as rate-limiting or account lockout policies, allowing unrestricted guessing attempts.

In contrast, dictionary attacks (a subset of password cracking) rely on precompiled lists of commonly used passwords rather than generating random combinations. These wordlists often contain popular passwords, phrases, and combinations, increasing the likelihood of a successful guess. Well-known wordlists used in such audits include `rockyou.txt` (Breault, 2017) and Crackstation's wordlist (Crackstation's, 2021). By using these curated lists, attackers can focus on more probable passwords, making dictionary attacks highly effective against weak and common password choices.

2.3 – Similar Systems

Although the system proposed in this project aims to be unique in its visualisations and ASCON integration, it is essential to examine existing tools that serve similar functions and are widely adopted in the industry. This section explores some of the most prominent password auditing tools, examining their capabilities, performance, and impact. Studying these tools not only acknowledges the technologies that have inspired this work but also provides valuable insights into established and effective techniques, offering opportunities to improve and refine the proposed system during development.

2.3.1 – John the Ripper

As described by Openwall (2019), "*John the Ripper is an open-source password security auditing and password recovery tool available for many operating systems*" and is notable for "*supporting hundreds of hash and cipher types.*" Its cross-platform support and ability to handle a vast array of hash types, including niche formats, is one of the many reasons John the Ripper (John) is respected as versatile and widely used tool in the password auditing community.

One of John's primary advantages is its CPU-centric design, written in C and Assembly, enabling low-level hardware optimisation and efficient resource usage. By making use of multi-threading and vectorised instruction sets (e.g., AVX, AVX512), John can perform parallelised password guessing (guessing multiple hashes at the same time), significantly improving its performance, especially when dealing with complex hashes or large datasets. This makes John particularly suitable for systems without dedicated GPUs or when targeting hashing algorithms that are more vulnerable to CPU-based attacks.

2.3.2 – Hashcat

Alternatively, Hashcat is another widely used and actively maintained password auditing tool. Although it may appear like John at first glance, it is fundamentally different in many ways. Both tools support CPU and GPU cracking, but while John is primarily optimised for CPU attacks, Hashcat is designed from the ground up to exploit GPU acceleration, making it significantly faster when appropriate hardware is available. However, performance is highly system-dependent, as on systems without powerful GPUs, John may outperform Hashcat due to its CPU efficiency.

One of Hashcat's most powerful and distinctive features is its mask attack functionality. Although mask attacks are not exclusive to Hashcat, the degree of customisation and flexibility it offers is unparalleled. A mask attack enables users to define known structural elements of a password, narrowing the key space and focusing guessing efforts on likely patterns. Masks consist of placeholders representing different character sets, and Hashcat uniquely supports custom character sets, allowing the combination of multiple predefined and user-defined sets. This level of control makes Hashcat extremely effective when an adversary may have partial knowledge of password formats, such as required lengths or specific character patterns.

2.3.3 – Benchmarks

Both Hashcat and Openwall provide extensive lists of benchmarks online which is publicly accessible and encourages others to contribute their own results. While these offers a broad overview of each tool's performance across various systems and supported hardware's, their crowdsourced nature, along with the lack of comparative benchmarking between each other lead to the conducting of personal performance benchmarks on both Hashcat and John the Ripper (Figure 2) with the aim to obtain more consistent and specific results that will be able to be used comparably against each other and the tool aiming to be developed in this project .

Figure 2 presents benchmarks for eight common hash types, measured in hashes per second (H/s). Hashes per second is a metric used to measure the rate of throughput of hashes attempted against a given target hash, this reflects both the computational power, but more importantly the efficiency of the auditing software. As for the benchmarks, they reflect the "real" performance (actual observed speed during the test). For each of the algorithms there is a result from both John and Hashcat.

Hash Type	John the Ripper Speed	Hashcat Speed
MD5	64,045,000 H/s	578,500,000 H/s
SHA1	37,334,000 H/s	320,400,000 H/s
SHA256	13,369,000 H/s	169,900,000 H/s
SHA512	1,695,000 H/s	65,741,500 H/s
NTLM	46,640,000 H/s	1,276,400,000 H/s
LM	96,703,000 H/s	85,357,800 H/s
bcrypt	6,141 H/s	2,054 H/s
scrypt	245 H/s	6 H/s

Figure 2 – A table depicting a list of common hash types along with their performance benchmarks for both systems (John the Ripper and Hashcat) measured in Hashes per second.

It is important to note that these benchmarks are not part of the application testing that will be covered later in the Methodology section. Instead, they serve as purely observational reference points made for performance benchmarks, helping to contextualise potential performance targets that the system being developed should aim to meet.

All benchmarks and future tests are conducted on the same machine and Linux environment to ensure consistency. The system used is equipped with an Intel 11th Gen i5-1135G7 processor (2.40GHz, 2319MHz) featuring 4 cores and 8 logical processors, along with 16GB of RAM.

2.4 – Understanding Password Strength

Conventionally, Shannon entropy (Shannon, 1948) is the metric most associated with password strength. However, the aim of this section is to demonstrate that this is not necessarily the most accurate or practical measure, and that instead there are alternative, more effective methods for metricising password strength.

2.4.1 – Shannon Password Entropy

Simply put Shannon Entropy is the measure of how unpredictable a password is and is computed as follows: $\text{Entropy} = \log_2(C^L) = L \times \log_2(C)$ where C is the character set size and L is the password length. While this mathematical approach provides a useful

theoretical baseline, it operates under the assumption that passwords are generated completely randomly. As a result, Shannon Entropy fails to account for predictable patterns, common letter structures, or user tendencies in password creation, limiting its accuracy in real-world scenarios.

2.4.2 – Password Quality Indicator

As previously mentioned, password strength is often mistaken with password entropy. Therefore, entropy is frequently assumed to correlate directly with the effectiveness of password auditing tools. As noted by Ma et al. (2010), "*lower entropy suggests a weaker or less secure password*".

However, this assumed direct relationship between password strength and entropy is "*loosely defined*" and, in fact is described as, "*an inadequate indicator of password quality*" (Ma et al., 2010), particularly when assessing resistance to real-world attacks. To address this limitation, Ma et al. (2010) proposed an alternative approach: the Password Quality Indicator (PQI), a more comprehensive measure of password robustness.

Ma et al. (2010) explain that "*we can measure the quality of a password by how different it is from dictionary words, how long it is, and how large the password's character set is.*" In practice, PQI is based on two key components: D, the Levenshtein editing distance, and L, the effective password length. Together, these metrics capture a range of factors, including password length, character variety, and pattern analysis. Ultimately, PQI provides a method to model real-world attacks by estimating the number of guesses an adversary would need to successfully break a specific password. More specifically, and the way intended implementation within this tool, "D", the Levenshtein editing distance, is calculated by comparing the target password to each entry within a given dictionary, with the smallest distance (that is, the most similar password) taken as D. For example, if the target password is "*password123*" and the dictionary contains "*password*", then $D = 3$, as three characters differentiate the two. The effective password length, denoted as L, is calculated as $L = (\text{length of password}) \times \log_2 (\text{total character set size})$. The total character set size reflects the range of unique characters present in the password; for instance, the password "*Password123*" includes uppercase letters (A–Z), lowercase letters (a–z), and digits (0–9), resulting in a total possible character set size of 62.

2.4.3 – Zxcvbn Rating

In contrast to the academic methods previously discussed, Zxcvbn is a widely adopted password strength estimation framework used in real-world applications. Developed by Dropbox engineer, Daniel Wheeler, Zxcvbn is described as “*a realistic password strength estimation*” (Wheeler, 2012), designed to evaluate passwords in a way that reflects how adversaries would attempt to crack them.

As outlined by Wheeler (2012), “*Zxcvbn process for estimating password strength involves three key stages: match, score, and search*”. In the matching phase, the system “*enumerates all the (possibly overlapping) patterns it can detect*”, identifying elements such as dictionary words, names, dates, and other common patterns. During the scoring phase, Zxcvbn “*calculates an entropy for each matched pattern*”, estimating the strength of each element. Finally, in the search stage, Zxcvbn optimises its detection of the most predictable, low-entropy patterns between those found.

Zxcvbn benefits from available Python implementations and a standardised scoring system that rates password strength on a scale from 0 to 4, making it an accessible and practical tool. Unlike PQI, Zxcvbn emphasises pattern recognition and attack modelling, offering a more applied approach suited for real-time password assessments. However, Zxcvbn is not without limitations. As Wheeler (2012) notes, “*Zxcvbn currently only supports English words*”, and its ability to detect common phrases is limited. Additionally, Wheeler highlights that “*the goal isn’t accuracy, though. The goal is to give sound password advice*”, underscoring that Zxcvbn prioritises practical advice over theoretical precision.

2.5 – ASCON Hashing Functions

As a cryptosystem itself ASCON’s functionalities are wide ranging, as for this project specifically it makes use of ASCON’s hashing functionality. That being the hash functions it provides, where a hash function is an algorithm that deterministically turns an arbitrary length input into a fixed-length output. Usually of which these are collision resistant and pre-image resistant (one-way). ASCON themselves “*offers 2 important hash functions: ASCON-HASH and ASCON-XOF*” (Yu et al., 2023).

Both models are built using “*a sponge-based hash function*” (Yu et al., 2023), which differs from the traditional Merkle-Damgård construction in “*favour of greater simplicity*”

and efficiency” (Bertoni et al., 2007). The sponge construction works by breaking the input data into small chunks, “absorbing” them into an internal state, and applying permutation. After padding is applied, the hash output is produced by “squeezing” data from this state. This approach is considered simple because it relies on a single core permutation, and efficient because it can handle inputs of any length while producing outputs of desired lengths.

ASCON-HASH, the first mode and makes use of this sponge construction. More specifically it has a fixed internal state of 320-bits, “The hashing modes absorbs the message (M as seen in figure 3) in 64-bit blocks” (Eichlseder, 2024) each block is then absorbed into this internal state then a 12-round permutation is applied (ASCON p-12), then the block is padded if needed. Once this has been done it is then “squeezed the hash value in 64-bit blocks” (Eichlseder, 2024), ultimately making a 256-bit output, which “provides 128-bit security” (Eichlseder, 2024).

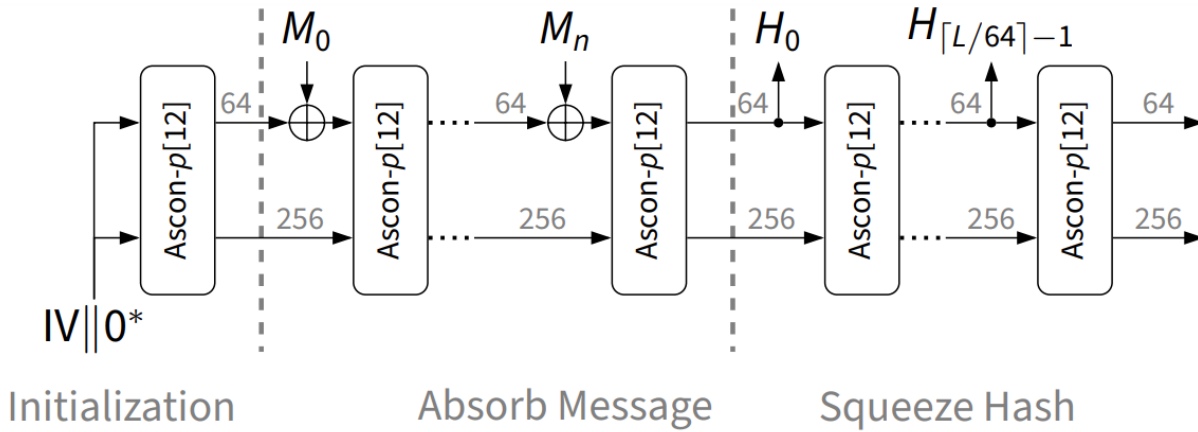


Figure 3 – A flowchart showing how ASCON-HASH operates (Eichlseder, 2024)

On the other hand, ASCON-XOF (Extendable output function), the second mode, differentiates itself in one keyway, it can take any-length input and produce an arbitrary-length output from that, meaning that it is flexible in terms of the output length required, whereas ASCON-HASH is fixed at 256-bits. The process itself is very similar to ASCON-HASH although it alternates when squeezing the processed input as it continues for as long as needed, applying permutations and reading output blocks. The same can be said for CXOF (Customisable XOF) which differs in the fact that it has built-in context-separation allowing for different use cases as it “additionally absorbs a customization string Z (as seen in figure 4) in 64-bit blocks before the message” (Eichlseder, 2024).

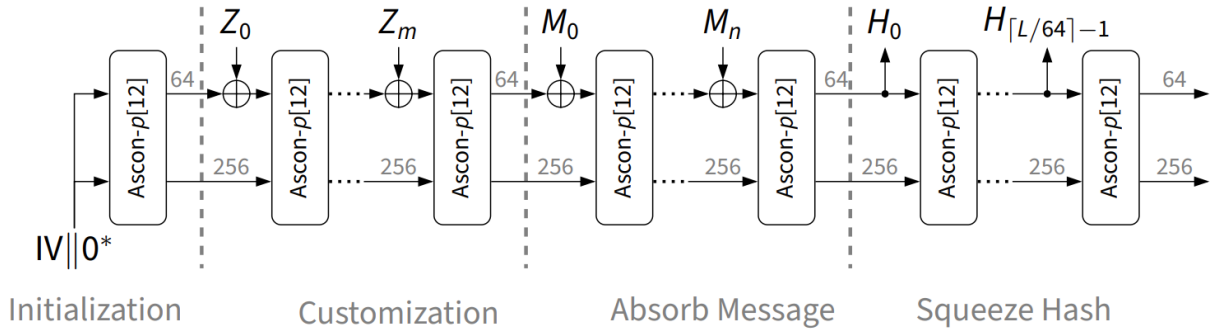


Figure 4 – A flowchart showing how ASCON-XOF/CXOF operates (Eichlseder, 2024)

2.6 – Summary

Specifically, the chapter has outlined what a brute-force attack is, including its sub-techniques, and identified the methods I intend to utilise, namely password cracking and dictionary attacks. Additionally, it presents two case studies of similar systems, analysing their most notable features and benchmarks for comparative analysis later in the project.

The chapter also concludes an evaluation of the concept of password strength, recognising the limitations of Shannon entropy as a metric. Therefore, within this project, references to "password strength" will rely on either the Password Quality Indicator (PQI) or Zxcvbn rating.

Finally, ASCON hashing functions are examined in detail, discussing functionality. In which this evaluation aims to build a foundational understanding and establish essential knowledge of such a hash function for the remainder of the project.

Chapter 3 – Methodology

3.1 – Introduction

This chapter is informally divided into three key sections, each serving a unique purpose. The first is the Design section, which presents a structured and comprehensive summary of the design decisions made throughout the system's development. These choices are informed by insights from previous chapters, particularly the Related Work chapter. Additionally, each design decision is carefully evaluated, with its advantages and disadvantages weighed to justify why it was selected.

The second section focuses on the Implementation phase, detailing the key methods and approaches used to bring the design to life. This includes an analysis of the tools and techniques employed, as well as an exploration of development process of the system. Furthermore, it highlights the design decisions and ethical insights that were directly incorporated into the implementation and examines which had the most significant impact.

The third and final section in chapter 3 is testing followed with optimisation, which focuses on the strategy and execution of system testing. It covers the types of tests conducted, execution details, results and changes made. Additionally, it discusses how these findings inform the next chapter and contribute to the overall evaluation of the system.

3.2 – Design Requirements

3.2.1 – Functional Design Requirements

Below is a concise list of the functional design requirements that the tool must fulfil. This includes both essential functionalities and required features. These requirements are strongly informed by the objectives outlined in Chapter 1 (Introduction), as well as by the concepts and insights discussed in Chapter 2 (Related Work).

No.	Description	Priority
1	Accept user inputs from both text based and txt file uploads. This should be done via a GUI interface which	High

	specifically allows for both wordlist and target hashes to be uploaded as a txt file, as well as a method for selecting which hashing algorithm a user wishes to use.	
2	Support at least two alternative methods for conducting a password audit.	High
3	Support the ASCON hashing algorithm including its three different hash modes, as mentioned in Chapter 2 these are the ASCON-256, XOF128 and CXOF128.	High
4	On top of ASCON support alternative popular hashing algorithms and have already been considered broken specifically MD5, SHA1, SHA256, SHA512, NTLM, LM, Bcrypt and Scrypt.	Medium
5	Output and display results once an audit has been completed, after auditing is processed the system should generate both multiple visualisations for monitoring performance, password strength as well presenting general findings from the data collected	Medium
6	Generate reports and maintain them in the form of logs, upon completion present a report of findings and key details, this should have the ability to be saved in the application but also downloadable and exportable	Medium
7	Ensure that the GUI is presents an accurate and up to date feed of the status of a given audit, this includes generally presenting feedback in response to users' interaction with the program.	Low
8	Cater for errors specifically via error handling, including the validation of user inputs but also any potential resource consumption issues that lead to unexpected crashes.	Medium
9	Make use of multi-processing such as threading and enable GPU processing to better the performance of the program	Medium

3.2.2 – Non-Functional Design Requirements

The following is a list of non-functional design requirements. Unlike functional requirements, which describe specific functions of the system, non-functional requirements define baseline benchmarks of attributes the system must meet. These include aspects such as performance, efficiency, and overall quality standards.

No.	Description	Priority
1	The tool should be cross compatible upon any system that supports the use of python specifically on Linux, MacOS and Windows.	Medium
2	The system should have response times for operations under 5 seconds for processes that aren't running an audit. And a time for when running an audit to ensure the program doesn't continue running for a very long time.	High
3	General Hashing speeds for both audit methods and across all algorithms should be an average of at least 10,000 Hashes per second	High
4	The program should make use of caps of system utilisation to ensure its not being overused, with a limit of 75% on both CPU and memory utilisation. This will hamper performance but increase reliability.	Medium
5	Generated report logs should be saved locally to the machine and accessible for 30 days before being deleted by the application. Additionally identifiable data should be removed from them.	Medium
6	Text file inputs should be scalable allowing for very large txt files to be uploaded without performance degradation	Medium
7	The code should be written in a user friendly and easily understandable manor with lots of comments to enable accessible code and invite open-source collaboration	Low
8	Live data measuring should be read into a txt file and then later read from a txt file instead of being directly interacted with by the application as processing such a vast amount would degrade performance.	Medium

3.3 – Logic Flowchart

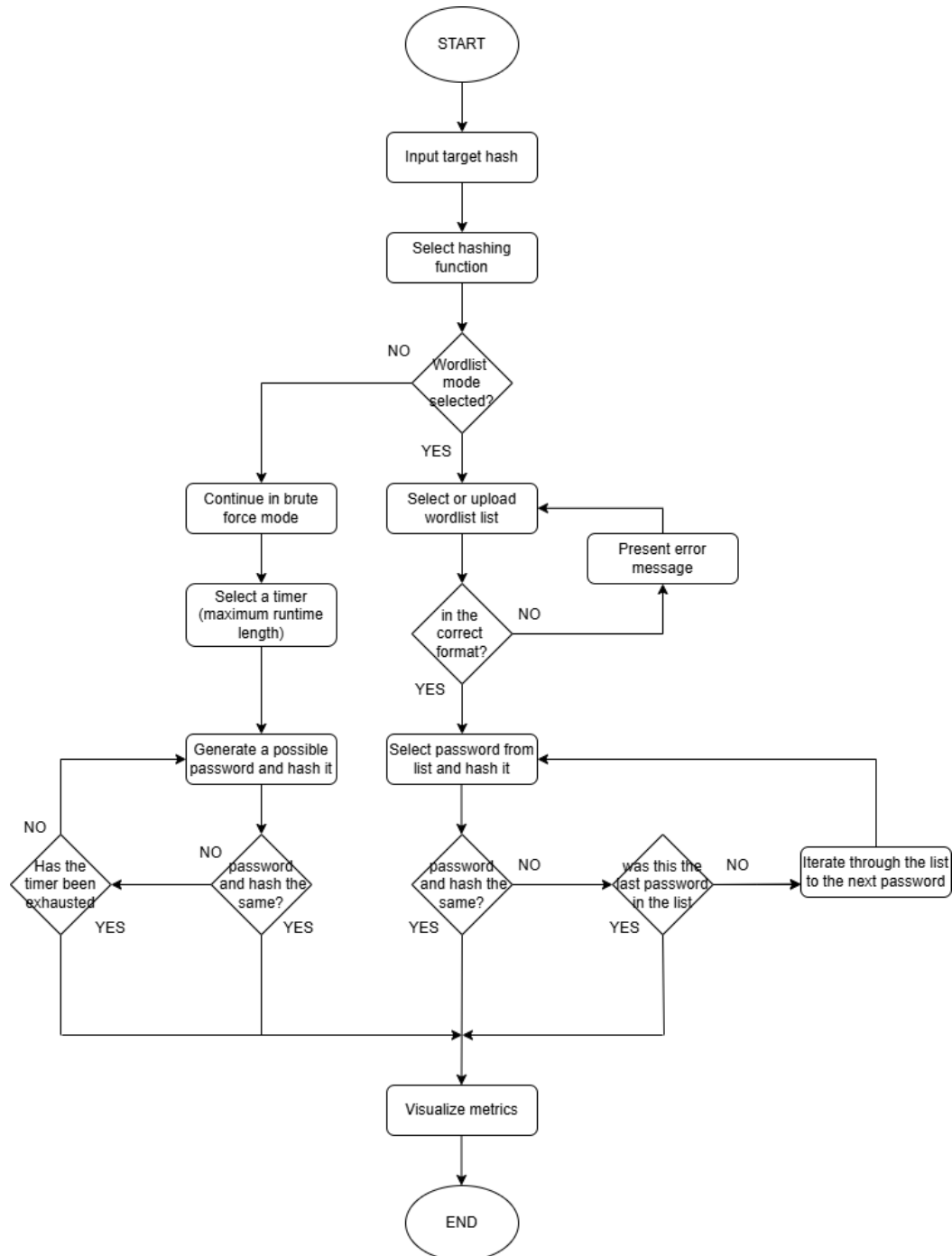


Figure 5 – A flowchart design of the intended logic of the program.

Figure 5 represents a breakdown of the core password auditing process using either brute force or a wordlist-based approach. It begins by inputting a target hash and selecting a hashing function. If wordlist mode is chosen, a list is uploaded and verified; otherwise, brute force mode is used with a runtime limit. The system hashes possible passwords and compares them to the target hash until the list is exhausted or the timer runs out.

3.4 – User Interface Designs

3.4.1 – Dashboard Page Wireframe

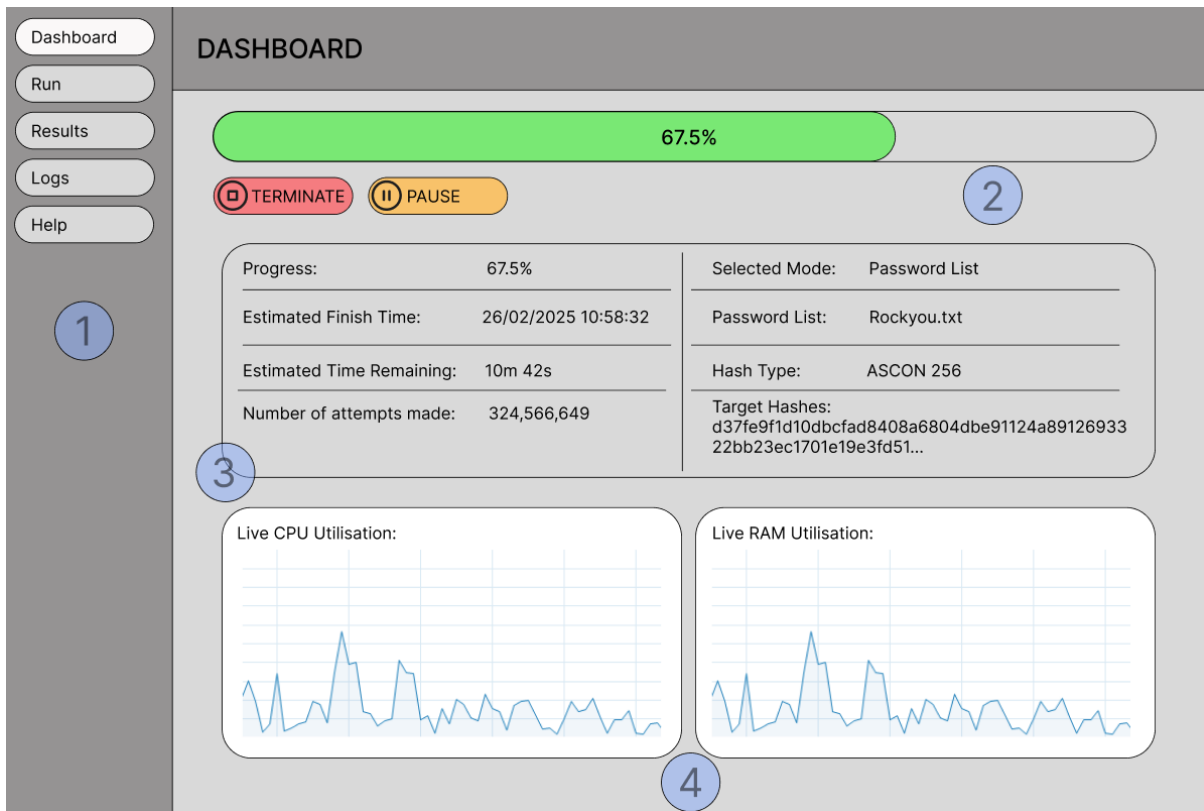


Figure 6 – A wireframe design of the dashboard page

The dashboard page itself will act as a main hub for users to access the most necessary information as fast as possible whilst also being able to assess the performance and time-taken in real time, it includes:

1. A side-oriented navigation bar, that allows users to navigate the pages of the application.
2. A Main progress bar alongside quick access operation buttons including a way of terminating, pausing and continuing the processing, the bar itself visualises the estimated time left.
3. A summary of the key details about the current process that is being run, this includes progress, estimated time to finish (and remaining), number of attempts, selected mode, selected wordlist and the hash type.
4. Two live visualisation feeds of the current CPU and RAM utilisation that updates in real time.

3.4.2 – Run Page Wireframe

The wireframe shows a web application interface for a 'Run' operation. On the left is a sidebar with buttons: 'Dashboard', 'Run' (highlighted), 'Results', 'Logs', and 'Help'. The main area is titled 'RUN'. It contains several input fields and buttons, numbered 1 through 5 for reference:

- 1**: 'Select Mode*:' with two buttons: 'Password List' and 'Brute Force'.
- 2**: 'Select Hash Type*:' with a dropdown menu showing 'ASCON 256'.
- 3**: 'Select Termination Time:' with a text input field showing '26/02/2025 13:00:00'.
- 4**: 'Upload Target Hashes*:' section. It has two options separated by 'OR':
 - Option 1: 'Upload a txt file containing the hashes (each hash on a new line):' with an upload icon (upward arrow in a box).
 - Option 2: 'Copy and paste your target hash into here (each hash on a new line):' with a large text area and a save icon (floppy disk).
- 5**: 'Upload Password List:' section. It also has two options separated by 'OR':
 - Option 1: 'Upload a txt file as as wordlist (each word should be on a new line)' with an upload icon.
 - Option 2: 'Paste a list of words into here (each word on a new line):' with a large text area and a save icon.

At the bottom center is a large green button labeled 'RUN'.

Figure 7 – A wireframe design of the run page

The run page is the main method of input from a user's perspective, this is where all key operations are chosen and selected to be able to provide an audit that produces a successful outcome, this page includes:

1. Select mode input is a required choice between password list or brute force. If the password list is selected, then the option to “Upload password list” will additionally be presented.
2. Hash type is the required selection for which the user is assuming the target hash to be the type of. This will be a dropdown selection box.
3. An optional input is the time a user wants the code to terminate at (assuming it is still running by then). This ensures that even if a user does not manually terminate the process, then it can be automatically done by a timer.
4. The upload target hash is the input where either a user can copy and paste multiple hashes into the textbox or upload a txt file of their own (assuming each hash is on a new line).
5. Like uploading a hash, the user can upload a password list up paste one in, this is assuming that the password list mode is selected.

3.4.3 – Results Page Wireframe

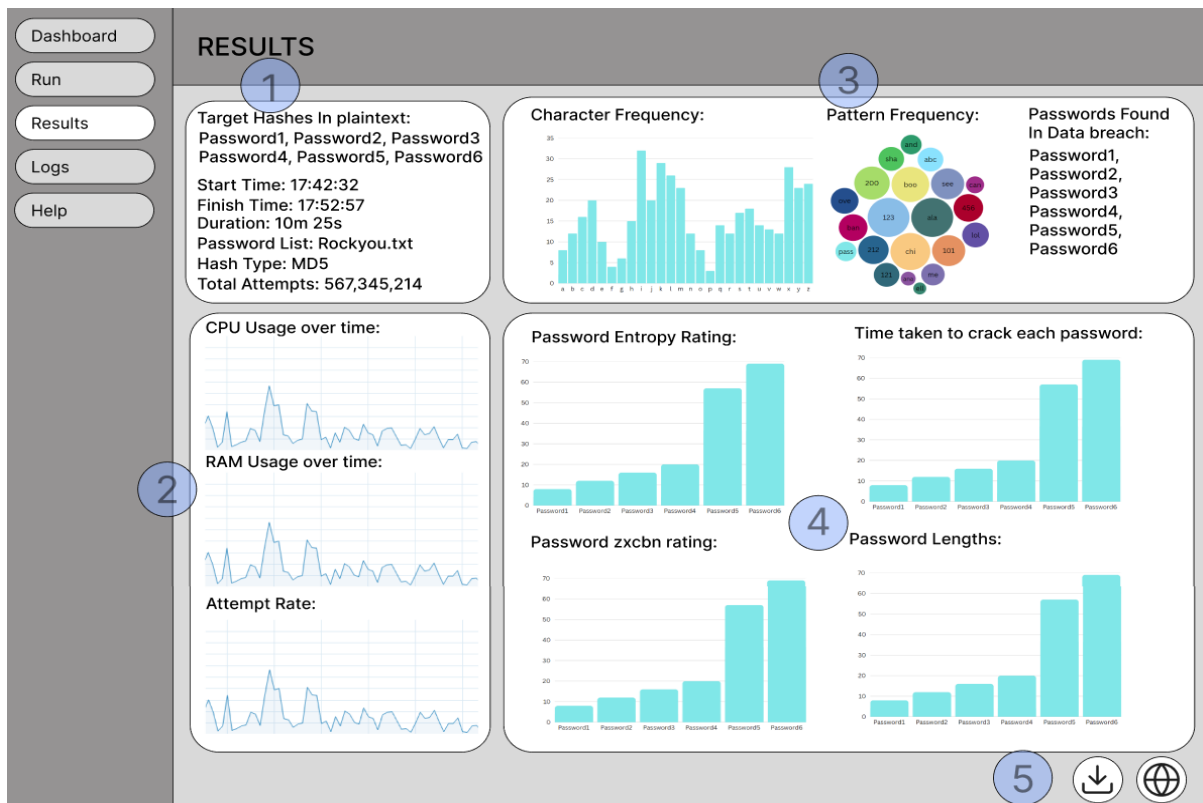


Figure 8 – A wireframe design of the results page

The results page is the key page that users will interact with to view the output of the operations taken place. Whilst also providing the ability to view overall data and export results. This page includes the following:

1. A box representing all the data present on the dashboard page and provides basic quantitative measurements such as start/finish times, duration, hash type, password list used and total attempts.
2. Three graphs that visualise CPU usage, RAM usage and hash rate over time, in combination they can be used to inform a user of the performance of an audit.
3. Two frequency-based graphs which show the character frequency in all the passwords found, as well as a pattern frequency word bank which displays if there are similar patterns between characters.
4. The other four graphs assess each of the cracked passwords individually; each password is provided with an entropy rating, the specific time taken to crack that password, a Zxcvbn rating as well as measuring the password length. These four combined helps build a picture of the strengths and weaknesses of each password that is found within the audit.

5. The two buttons at the bottom of the screen let users either export the current report or switch view to a more general visualisation (in which they change) to depict the overall results

3.4.4 – Log Page Wireframe

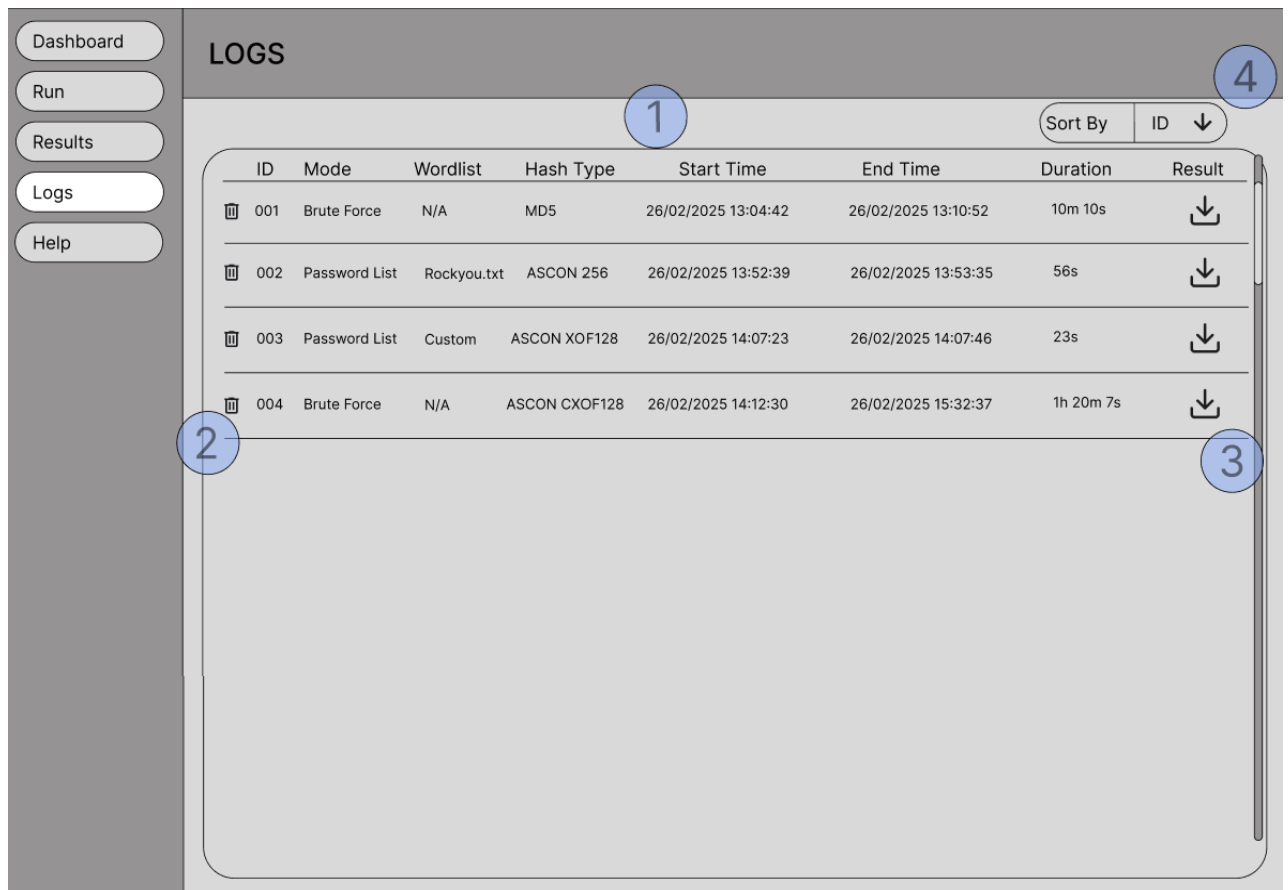


Figure 9 – A wireframe design of the log page

The logs page acts as a databank to access all prior audits that have been conducted. These will be saved locally as well as the option to be able to export them as different file formats. This page specifically includes:

1. A list that shows all audits conducted as well as provides important information about them such as ID number, mode, wordlist, hash type, times and duration.
2. On each log there is an option to delete the log, removing it from the locally saved file.
3. Similarly, there is a download button for each which allows the user to download/export a version of the log file.
4. Finally, there is a sort of button at the top which will enable a user to sort all logs by different conditions (newest first, oldest first, mode, hash type, etc.)

3.4.5 – Help Page Wireframe

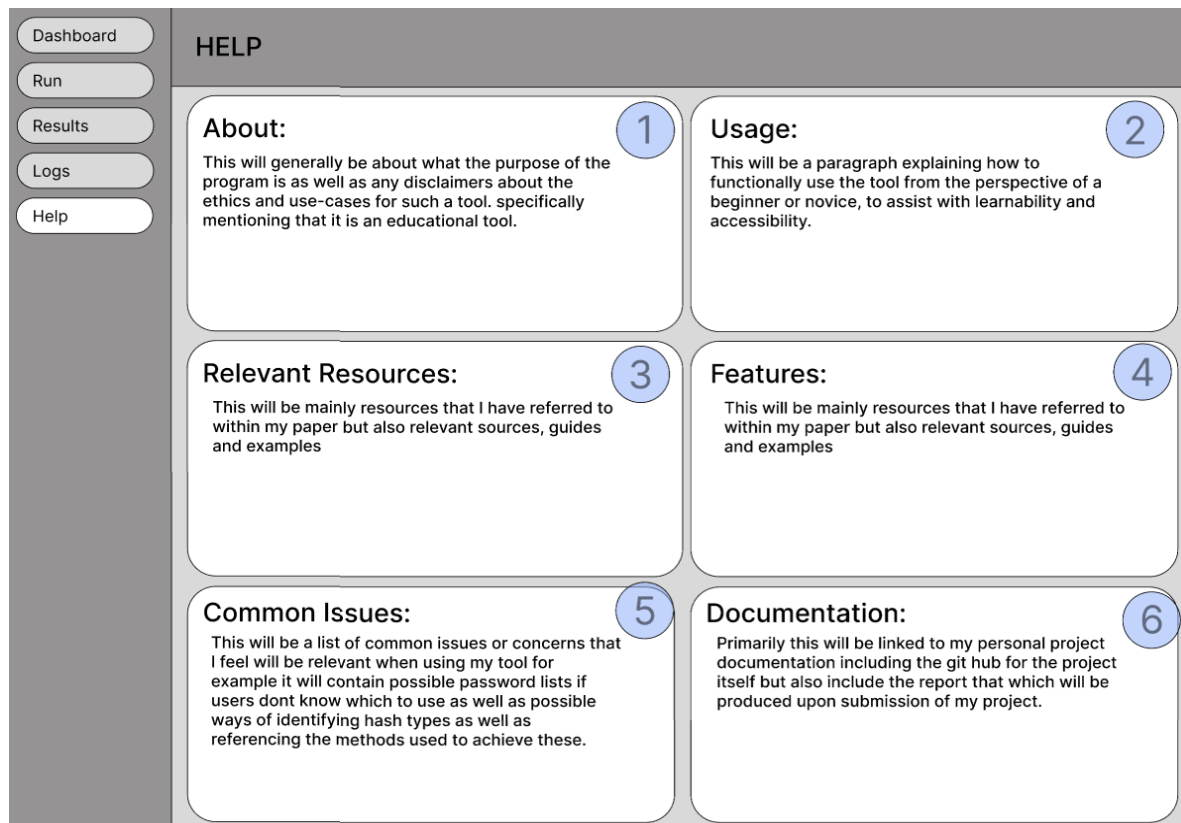


Figure 10 – A wireframe design of the log page

The help page is an all-in-one guidebook to using the tool. Its main goal is to provide the user with the knowledge to use the application with the secondary objective of guiding users to this report, it includes:

1. The about section will describe the purpose of the tool and included disclaimers about the use-cases for it.
2. The usage section functionally teaches and guides the user on how to use the tool, it also discusses the backend features of the tool.
3. The relevant resources will link to resources, guides and examples of password auditing tools and will generally include tool's that benefit the use of this tool.
4. The features section will advertise the key disparities between this tool and already existing ones and explain what makes this one unique.
5. The common issues section covers common or relevant issues a user may come across when using the tool.
6. The documentation section will primarily link to this very report, as well as this GitHub to provide extra context to the tool.

3.5 – Tools & Technologies Used

This section outlines the tools and technologies implemented in the program. Each technology is accompanied by a brief description, a justification for its selection (including its advantages and disadvantages), and examples of its application.

3.5.1 – Python

Python is a “*high-level, easy-to-learn programming language*” widely used for various applications. It features built-in “*support for modules and packages*”, promoting program modularity (Python Software Foundation, 2024).

Specifically, Python was chosen primarily due to its extensive library support, particularly in data science, data visualisation, and cryptography. Additionally, it offers a pre-existing implementation of the ASCON hashing algorithm, which was crucial given the project’s scope and time constraints. By making use of these existing packages as well having a personal familiarity with this language allowed for a more efficient development process.

However, the use of a high-level language like Python did present some challenges. Drawing from the experiences with systems like John the Ripper and Hashcat, which are built with lower-level languages, it’s evident that lower-level programming can provide better optimisation and finer control over memory. This, in turn, results in inherent enhanced system efficiency and faster execution speeds, that isn’t otherwise possible with python.

3.5.2 – Custom Tkinter

CustomTkinter is an extension built on the popular Python GUI framework Tkinter, designed to create enhanced interfaces with modernised styling for a more visually appealing user experience, an essential requirement to project such as this one.

The decision to use CustomTkinter stems from its ability to address the visual limitations of Tkinter while preserving its lightweight nature. It offers a powerful yet approachable solution, making it ideal even for developers without a background in frontend development. Additionally given the smaller scope of the project, the performance drawbacks commonly associated with larger projects and CustomTkinter are not an issue, making CustomTkinter an optimal choice for the scale of the project.

```

import customtkinter as ctk

class Help(ctk.CTkFrame):
    def __init__(self, parent, controller):
        super().__init__(parent)
        self.controller = controller

        # Scrollable frame
        scrollable_frame = ctk.CTkScrollableFrame(self, width=800, height=600)
        scrollable_frame.pack(expand=True, fill="both", padx=20, pady=20)

        # Main Heading
        title_label = ctk.CTkLabel(scrollable_frame, text="Help", font=("Arial", 26, "bold"))
        title_label.pack(pady=(10, 5), anchor="center")

        # Intro paragraph
        description = ctk.CTkLabel(scrollable_frame, text="Welcome to the Help Page", font=("Arial", 16))
        description.pack(pady=(0, 20), anchor="center")

```

Figure 11 – Code snippet from help page, an example of Custom Tkinter’s usage.

3.5.3 – Hashlib

Another useful Python module is hashlib, which “*provides functions for efficiently hashing files*” (Python, 2024). In this case, it is specifically used to support a variety of common hash types, including MD5, SHA-1, SHA-256, and SHA-512 (seen in figure 12). Within this specific implementation the module hashes strings, which are then compared against the target hash later in the program.

The primary reason for using Hashlib is its fast performance and optimisation, which allows it to efficiently hash and handle large amounts of data per second. This makes it perfectly suited for the needs of this project. Additionally, its popularity can be attributed to the wide range of supported hash algorithms, many of which are used in this project. In fact, Hashlib handles more than half of the hashes supported in the application with the other’s being either ASCON or supported by Passlib.

However, it’s not without its limitations. Hashlib does not support lesser-known hashes (such as ASCON) and it lacks native support for adding salt’s (random data) to hashes, a common security practice, to reduce the deterministic nature of hash functions. This therefore inherently means Hashlib doesn’t incorporate more complex hashing algorithms either.


```

def hash_string(hash_type, string, hash_length=32): # function that hashes string by chosen hash type
    if hash_type == "MD5":
        return hashlib.md5(string.encode()).hexdigest() # hash to md5
    elif hash_type == "SHA-1":
        return hashlib.sha1(string.encode()).hexdigest() # hash to sha1
    elif hash_type == "SHA-256":
        return hashlib.sha256(string.encode()).hexdigest() # hash to sha256
    elif hash_type == "SHA-512":
        return hashlib.sha512(string.encode()).hexdigest() # hash to sha512
    elif hash_type == "Ascon-Hash256":
        return ascon_hash(message=string.encode(), variant="Ascon-Hash256", hashlength=32).hex() # hash to ascon 256
    elif hash_type == "Ascon-XOF128":
        return ascon_hash(message=string.encode(), variant="Ascon-XOF128", hashlength=hash_length // 2).hex() # hash to ascon xof128
    elif hash_type == "Ascon-CXOF128":
        return ascon_hash(message=string.encode(), variant="Ascon-CXOF128", hashlength=hash_length // 2).hex() # hash to ascon cxof128
    elif hash_type == "NTLM":
        return hashlib.new('md4', string.encode('utf-16le')).hexdigest().upper() # hash to ntlm
    elif hash_type == "LM":
        return lmhash.hash(string).upper() # hash to ntlm
    else:
        raise ValueError("Unsupported hash type")

```

Figure 12 – Code snippet from backend, an example of Hashlib’s usage, hashing multiple types of hashes.

3.5.4 – Matplotlib

Matplotlib is a powerful library that helps to enable data visualisation by generating graphs and figures from data sources. It plays a crucial role throughout the application, providing clear, data-driven insights to users. Specifically, it has enabled the creation of various visualisations, including a character frequency distribution, performance metrics over time (such as CPU and RAM utilisation), a success rate pie chart, a password length histogram, and password strength charts using both PQI and Zxcvbn. These visuals help to achieve the previously set out objectives of providing data focused insights after an audit is conducted.

Its general popularity and adoption within my project can be attributed in part to its seamless integration with other libraries, such as its smooth compatibility with Custom Tkinter. Additionally, the range of plotting techniques, visualisations, and customisation options it offers is impressive and well-documented. The only drawback encountered in this project was that Matplotlib is primarily designed for static plots, which led to the use of other libraries like Mplcursors and Pyplot to enable interactive elements within the interface.

```

def display_success_rate_pie_chart(self, results):
    # Define success and failure counts
    success_count = sum(1 for data in results['results'].values() if data['password'] != "Password not found.")
    failure_count = len(results['results']) - success_count

    # Data for the pie chart
    labels = ['Success', 'Failure']
    sizes = [success_count, failure_count]
    colors = ['#4CAF50', '#F44336'] # Green and red

    # Set background color to match application dark theme (#2b2b2b)
    fig, ax = plt.subplots(figsize=(10, 5))
    fig.patch.set_facecolor('#2b2b2b') # Set figure background color
    ax.set_facecolor('#2b2b2b') # Set axes background color

    # Create the pie chart
    ax.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90, colors=colors,
           textprops={'color': 'white'}, wedgeprops={'edgecolor': 'black'})

    # Set title
    ax.set_title('Success Rate', color='white')

    # Display the graph in the tkinter window
    self.graph_canvas = FigureCanvasTkAgg(fig, self.graph_frame)
    self.graph_canvas.get_tk_widget().pack()
    self.graph_canvas.draw()

    # Close the figure after rendering it
    plt.close(fig) # Prevents the figure from staying open in memory

```

Figure 13 – Code snippet from the results page, an example of Matplotlib’s (Plt) usage with a pie chart.

3.5.5 – Pandas

Pandas is a Python library that played a crucial role in processing the large volumes of data generated by the program. Specifically, it was used to analyse and manage the numerous text files the program relies on for both input and output. One of its key applications is logging CPU and memory utilisation throughout the runtime, as logs are created every 0.0001 seconds, resulting in a substantial amount of data.

The advantages of this library lie in its powerful data structures and seamless integration with other libraries (such as Matplotlib). This makes it highly efficient and optimised for handling large datasets, which is essential for this application. However, one equally important consideration is the lack of native parallelisation and multi-threading support in Pandas, which requires the use of additional libraries to handle those tasks.

```
def visualize_system_usage(self, log_file="usage_log.txt"):
    # Read the timestamp, cpu usage and ram usage data into a DataFrame
    df = pd.read_csv(log_file, names=["timestamp", "cpu_usage", "ram_usage"])

    # Convert timestamps to readable format
    df['timestamp'] = pd.to_datetime(df['timestamp'], unit='s')
```

Figure 14 – Code snippet from the results page, an example of reading the log file using pandas and processing the data to a dataframe (df), to be used in a visualisation.

3.5.6 – Zxcvbn Implementation

As the name suggests, this library provides a Python implementation of the Zxcvbn password scoring system, which had an analysis, and its advantages considered within background research section (Chapter 2). Its primary function is to evaluate the strength of passwords by running them through the Zxcvbn algorithm. Each password assessed after the audit is assigned a score ranging from 0 to 4 (shown in figure 15), reflecting its strength based on the Zxcvbn scale.

```
def display_password_zxcvbn_graph(self, results):
    passwords = [data['password'] for data in results['results'].values() if
                  data['password'] != "Password not found."]

    # Use zxcvbn to calculate password strength score (0-4)
    strength_scores = {pwd: zxcvbn.zxcvbn(pwd)['score'] for pwd in passwords}
    sorted_strength = dict(sorted(strength_scores.items(), key=lambda x: x[1]))

    # Set background color to match application dark theme (#2b2b2b)
    fig, ax = plt.subplots(figsize=(10, 5))
    fig.patch.set_facecolor('#2b2b2b') # Set figure background color
    ax.set_facecolor('#2b2b2b') # Set axes background color
    bars = ax.barh(list(sorted_strength.keys()), list(sorted_strength.values()), color='royalblue') # Blue bars
    ax.set_xlabel('Password Strength (0-4)', color='white')
    ax.set_title('Password Strength for Each Password', color='white')
    ax.tick_params(axis='both', labelcolor='white') # Set tick label color to white

    # Add tooltips with mplcursors
    mplcursors.cursor(bars, hover=True).connect(
        "add", lambda sel: sel.annotation.set_text(f'Password: {list(sorted_strength.keys())[sel.index]}\n'
                                                    f'Strength: {list(sorted_strength.values())[sel.index]}'))

    self.graph_canvas = FigureCanvasTkAgg(fig, self.graph_frame)
    self.graph_canvas.get_tk_widget().pack()
    self.graph_canvas.draw()

    # Close the figure after rendering it
    plt.close(fig)
```

Figure 15 – Code snippet showing an example of Zxcvbn used to score passwords before visualisation

3.5.7 – ASCON Implementation

One of the key components of this project, crucial to achieving the set goals and objectives, is the Python implementation of ASCON (Eichlseder, 2023). At its core, this is the ASCON Hashing algorithm (figure 16), adapted as a Python function that can hash strings in any of the three previously discussed modes. I have integrated this functionality into the tool, enabling it to hash strings based on the selected mode and compare them against the target hash. This feature is a vital part of the project, and, given its significance, its advantages are self-evident.

```
def ascon_hash(message, variant="Ascon-Hash256", hashlength=32):

    versions = {"Ascon-Hash256": 2,
                "Ascon-XOF128": 3,
                "Ascon-CXOF128": 4}
    assert variant in versions.keys()
    if variant == "Ascon-Hash256": assert (hashlength == 32)
    a = b = 12 # rounds
    rate = 8 # bytes
    taglen = 256 if variant == "Ascon-Hash256" else 0

    # Initialization
    iv = to_bytes([versions[variant], 0, (b << 4) + a]) + int_to_bytes(taglen, 2) + to_bytes([rate, 0, 0])
    S = bytes_to_state(iv + zero_bytes(32))
    if debug: printstate(S, "initial value:")

    ascon_permutation(S, 12)
    if debug: printstate(S, "initialization:")

    # Message Processing (Absorbing)
    m_padding = to_bytes([0x01]) + zero_bytes(rate - (len(message) % rate) - 1)
    m_padded = message + m_padding

    # message blocks 0,...,n
    for block in range(0, len(m_padded), rate):
        S[0] ^= bytes_to_int(m_padded[block:block + rate])
        ascon_permutation(S, 12)
    if debug: printstate(S, "process message:")

    # Finalization (Squeezing)
    H = b""
    while len(H) < hashlength:
        H += int_to_bytes(S[0], rate)
        ascon_permutation(S, 12)
    if debug: printstate(S, "finalization:")
    return H[:hashlength]
```

Figure 16– Code snippet showing the of the ASCON hashing function as implemented in python (Eichlseder, 2023), that is used in the previously witnessed hash string function (figure 12)

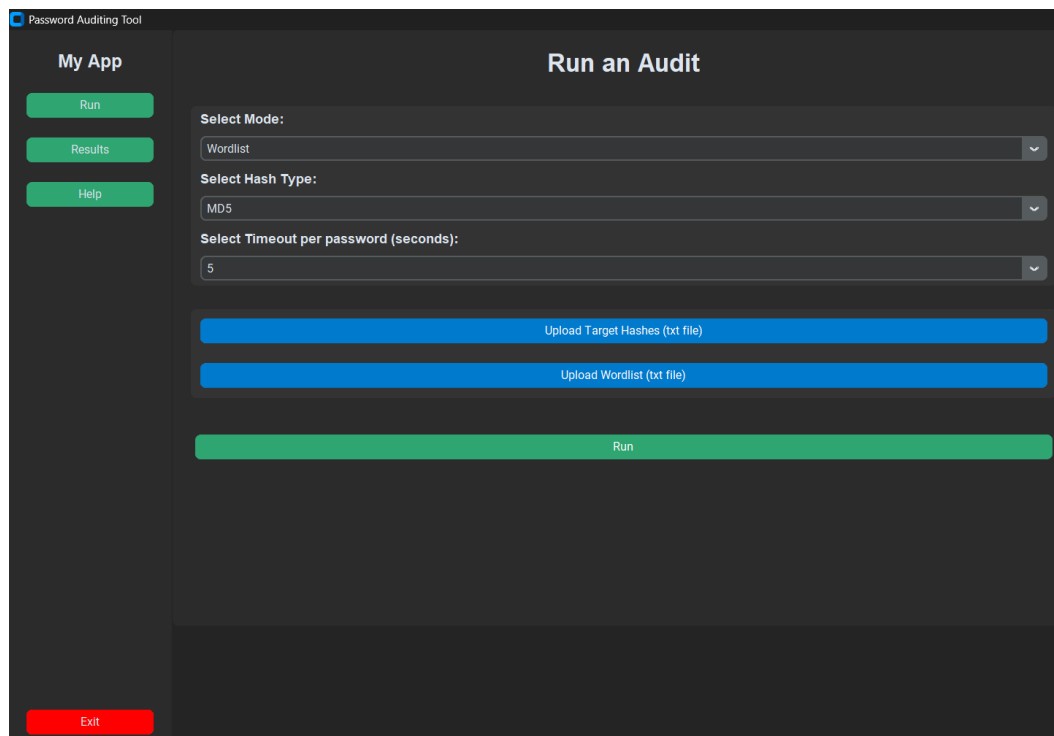
3.6 – Development

This section covers the actual development process carried out during the implementation stage of the waterfall model that was employed during the project's lifecycle. Not only does that act as a walkthrough of the system and its main functionalities. But also, a reflection of the process including design considerations made.

3.6.1 – User Inputs & Uploading Files

One of the core features introduced early in the development of the application was the file upload functionality, designed to simplify content input for processing. This feature allows users to upload a target hash (the hash they wish to reverse into plaintext) and, optionally, a wordlist for comparison. Additionally, the application requires the selection of a hash type, a timer, and an option to choose between wordlist or brute force modes.

For ease of use, the modes, hash type selection, and timer are all presented through simple dropdown selections (see figure 17). However, the handling of target hash and wordlist inputs posed a more significant challenge. Initially, the application allowed users to copy and paste values into a text box. This approach, though simple, was reconsidered in favour of supporting file uploads, specifically .txt files.



The screenshot displays the 'Run an Audit' interface of the Password Auditing Tool. On the left, a sidebar titled 'My App' contains buttons for 'Run', 'Results', 'Help', and 'Exit'. The main area features a 'Run an Audit' section with three dropdown menus: 'Select Mode:' (set to 'Wordlist'), 'Select Hash Type:' (set to 'MD5'), and 'Select Timeout per password (seconds):' (set to '5'). Below these are three file upload buttons: 'Upload Target Hashes (txt file)', 'Upload Wordlist (txt file)', and a large green 'Run' button.

Figure 17 – The frontend for the Run page, showcasing the input types and methods.

This shift to file uploads was a strategic decision that addressed several key issues. Copying and pasting large wordlists into a text box is not only error-prone but also cumbersome, often encountering formatting issues, particularly when dealing with sizable datasets or complex wordlists. Furthermore, the performance of the application can be negatively affected when handling large amounts of data pasted directly into a text box. By supporting file uploads, where each line is automatically stripped and sanitised, the application can handle larger and more complex datasets seamlessly. This method also provides the added benefit of enabling users to upload multiple target hashes in a single .txt file, streamlining the process even further.

3.6.2 – Wordlist Audit

After being able to gather all the required data to conduct an audit the program now needed a mode of auditing, this first mode chosen to do so (and the most recommended to use) is the wordlist mode.

As a general overview it is automatically ran in the background once selected on the run page and when the “Run button is pressed”, this function as described previously will consist of hashing every single word in the wordlist and then compare it to the target hash. If a match occurs then the algorithm has found the plaintext version of the target hash, if not (the timer runs out) it will return “Password not Found” On top of this analytical data is measured such as the time taken to find the hashes plaintext equivalent as well as the number of attempts made.

```
def wordlist_crack(target_hash, hash_type, wordlist_path, timeout=None):
    start_time = time.time()
    guesses = 0

    with open(wordlist_path, 'r', encoding='utf-8', errors='ignore') as file: # read wordlist file
        for line in file: # for each word in wordlist
            word = line.strip() # strip the line
            guesses += 1 # add +1 guess counter
            if hash_string(hash_type, word, len(target_hash)) == target_hash: # run hash the word and compare it to the target
                elapsed = time.time() - start_time # if it's the same stop the timer
                return word, elapsed, guesses # return the plaintext, total time and number of guesses
            if time.time() - start_time > timeout: # if the timeout is surpassed
                return None, time.time() - start_time, guesses # only return the time and the number of guesses made
    return None, time.time() - start_time, guesses
```

Figure 18 – A code snippet of commented code from the backend algorithm for conducting a wordlist audit

3.6.3 – Brute Force Audit

Alternatively, the other mode for operation which isn't as efficient as the wordlist audit (something that will be considered in more detail later) is the brute force mode. This is where the algorithm attempts every single character combination (requiring a significantly large amount of processing power).

This like the wordlist audit is ran in the background once chosen and the Run button is pressed. Although implemented this function still is unstable in the way that it can cause serious crashes and significantly slow the run time of the program, even with introducing multiprocessing and a maximum character limit, this is in part to the nature of such the mode as it requires lots of processing power. This in turn is a highlight and one of the key areas of improvement. And one which is involved within the following testing and optimisation processes. As for its functionality it can be witnessed in figure 19 and is like that of the wordlist audit.

```
def brute_force_worker(args):
    target_hash, hash_type, attempt = args
    attempt_str = ''.join(attempt) # turns character set into string
    if hash_string(hash_type, attempt_str, len(target_hash)) == target_hash: # compares hashes characters against target hash
        return attempt_str # if they are equal return the plaintext
    return None

def brute_force_crack(target_hash, hash_type, max_length=6, timeout=None):
    chars = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789" # all possible characters (no special chars)
    start_time = time.time() # begin timer
    guesses = 0 # start attempt counter
    with multiprocessing.Pool(processes=multiprocessing.cpu_count()) as pool: # create multiprocessing worker pool
        for length in range(1, max_length + 1): # iterate over all possible string lengths (max 6)
            attempts = itertools.product(chars, repeat=length)
            args = ((target_hash, hash_type, attempt) for attempt in attempts) # create args for worker function
            for result in pool.imap_unordered(brute_force_worker, args, chunksize=1000): # distribute work to worker
                guesses += 1 # each time add to guesses
                if result is not None: # if results is found
                    elapsed = time.time() - start_time # finish timer
                    return result, elapsed, guesses # return results, total time and total guesses
            if time.time() - start_time > timeout: # finish timer
                return None, time.time() - start_time, guesses # return only time and guesses
    return None, time.time() - start_time, guesses
```

Figure 19 – A code snippet of commented code from the backend algorithm for conducting a brute force audit, split into two functions.

3.6.4 – Hash Types

Once the text files are uploaded and audit mode is selected, the next step is for the user to choose a hash type. This selection depends on which hashing algorithm is believed to have been used to generate the target hash. Identifying the correct hashing family can be complex, as it often requires specialised tools. A recommended tool for this purpose is "Hash Identifier" (Zion3R, 2024).

As shown in Figure 12, the supported hash types include MD5, SHA-1, SHA-256, SHA-512, LM, NTLM, and ASCON. These were selected due to their support in existing hashing libraries, lightweight nature, popularity, and the fact that each is already considered broken. ASCON was specifically included to align with the project's aims and objectives.

On the other hand, certain hash types like scrypt and bcrypt were not included. As mentioned in the tools & technologies section, these algorithms are not natively supported by hashlib due to their use of salts, which hashlib does not support. Additionally, incorporating these algorithms would have been challenging due to their complexity, which would negatively impact the program's runtime performance. However, expanding the suite of supported algorithms is a key improvement area, and it remains a topic for future development before and during the testing and optimisation processes.

3.6.5 – System Utilisation Logging

Once all the input fields are completed, the user clicks the "Run" button to initiate the process. This action triggers a backend function that performs multiple tasks, including verification, execution of the audit algorithms, and the monitoring process.

Initially, the monitoring process was considered a minor aspect of the project, which is why it was not explicitly mentioned in the requirements and objectives. However, as the project progressed, it became clear that monitoring was both significant and crucial, as it was indirectly linked to several key requirements. The monitoring process tracks various data points, such as CPU usage, memory usage, time taken (both overall and per hash), start/finish times, and the number of attempts made. These statistics are measured by multiple functions, with one of the primary functions responsible for system utilisation being illustrated in Figure 20.


```

def monitor_system_usage(): # function takes a reading of CPU and RAM Usage
    cpu_usage = psutil.cpu_percent()
    ram_usage = psutil.virtual_memory().percent
    return cpu_usage, ram_usage # returns both

def monitor_usage_periodically(log_file="usage_log.txt"):
    with open(log_file, 'a') as file: # Open in append mode
        # Log an initial entry before the loop starts
        timestamp = time.time() # take time measurement
        cpu_usage, ram_usage = monitor_system_usage() # take cpu and memory measurement
        file.write(f"{timestamp},{cpu_usage},{ram_usage}\n")
        file.flush() # immediately write contents

    while True:
        timestamp = time.time() # takes time measurement
        cpu_usage, ram_usage = monitor_system_usage() # take cpu and memory measurement
        file.write(f"{timestamp},{cpu_usage},{ram_usage}\n")
        file.flush() # immediately write contents
        time.sleep(0.0001) # interval between next log, still need to tweak this

```

Figure 20 – A code snippet of commented code from the backend file of the program, specifically a function for measuring CPU and Memory utilisation.

3.6.6 – Results Visualisations

Once the user has entered all required data, executed the program, and the audit has been successfully completed (verified, monitored, and finalised) the results of the audit are now ready for review.

The user is automatically redirected to the results page (figure 21), which may have been previously accessed but would either display no data or results from a prior audit. On this page, the user can review all relevant audit data, including the hashes and their corresponding plaintext equivalents (if discovered). Additional details about the audit are also provided, such as the hash type and mode used, the total duration of the audit, the number of attempts made, the average attempts per second, the time taken per hash, the wordlist employed, and the respective start and finish times.

Moreover, the results include several visualisations that leverage the audit data to offer deeper insights. These visualisations, which will be explored in greater detail, encompass the following: the character frequency distribution, the Password Quality Index graph, the Zxcvbn score graph, the system usage graph, the password length histogram, and the success rate pie chart.



Figure 21 – A screenshot of the frontend results page, each hash and its plaintext on the left, on the right an overview of results and below that buttons to view each graph.

Beginning with the first graph, as seen in figure 21, this is a graph which displays the character frequency distribution of all the characters within the plaintext passwords (that have been found). Displaying in a sorted order from characters least present to most present. The use case for this graph it to help find patterns within the results data which otherwise wouldn't be so visible, as is can provide insights into what characters that specific dataset of hashes leans on the most.

The next graph, figure 22, visualises the PQI (Password Quality Index) scores of all successfully cracked passwords. As previously discussed, PQI is a scoring system designed to quantify the strength of a password, the higher the PQI, the stronger the password. In this graph, passwords are sorted from highest to lowest PQI. By representing password strength in a clear, metric-based format, the visualisation enables a deeper understanding of what constitutes a "strong" password. It also highlights weak passwords, making it easy to compare the full spectrum of quality. Furthermore, helping to identify patterns in weak choices, recognise common traits among stronger ones, and exposes the disparity between the two.

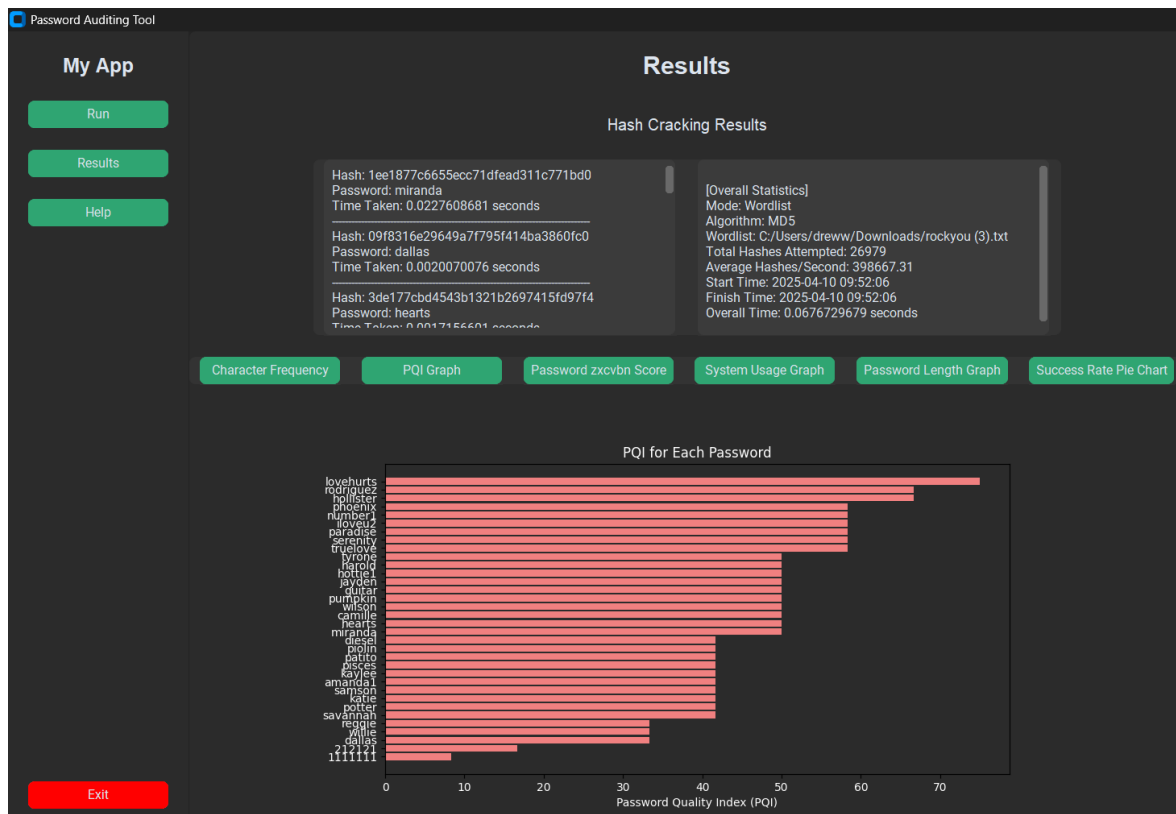


Figure 22 – A screenshot of the frontend results page, showing the PQL bar chart visualisation, measuring each password found based on PQL score.



The next visualisation seen in figure 23 also focuses on evaluating password strength, but through an alternative method: the Zxcvbn algorithm. Unlike the PQI system, which uses its own proprietary scale, Zxcvbn rates password strength on a scale from 0 to 4 (hence why each visualisation is separate, as they use different scaling). This comparison offers valuable insight, as the two systems often produce differing results when assessing the same password. These discrepancies not only highlight the varying criteria and assumptions behind each model but also shed light on potential biases in how password strength is quantified. By visualising these differences, it can help to better understand the strengths and limitations of each approach, whilst generally help providing perspective on what passwords may be the overall strongest.

The system utilisation graphs shown in Figure 24, previously discussed in detail, are valuable tools for monitoring real-time system performance. These graphs (one displaying CPU usage and the other RAM usage) offer a direct visual representation of how intensively the system is being used during program execution, providing insight into program efficiency and aids in diagnosing performance issues. Due to high utilisation levels often correlating with slower execution speeds, as the system must allocate more resources to handle the workload.



Figure 24 – A screenshot of the frontend results page, showing the system utilisation graphs (the top being CPU utilisation the latter memory utilisation)

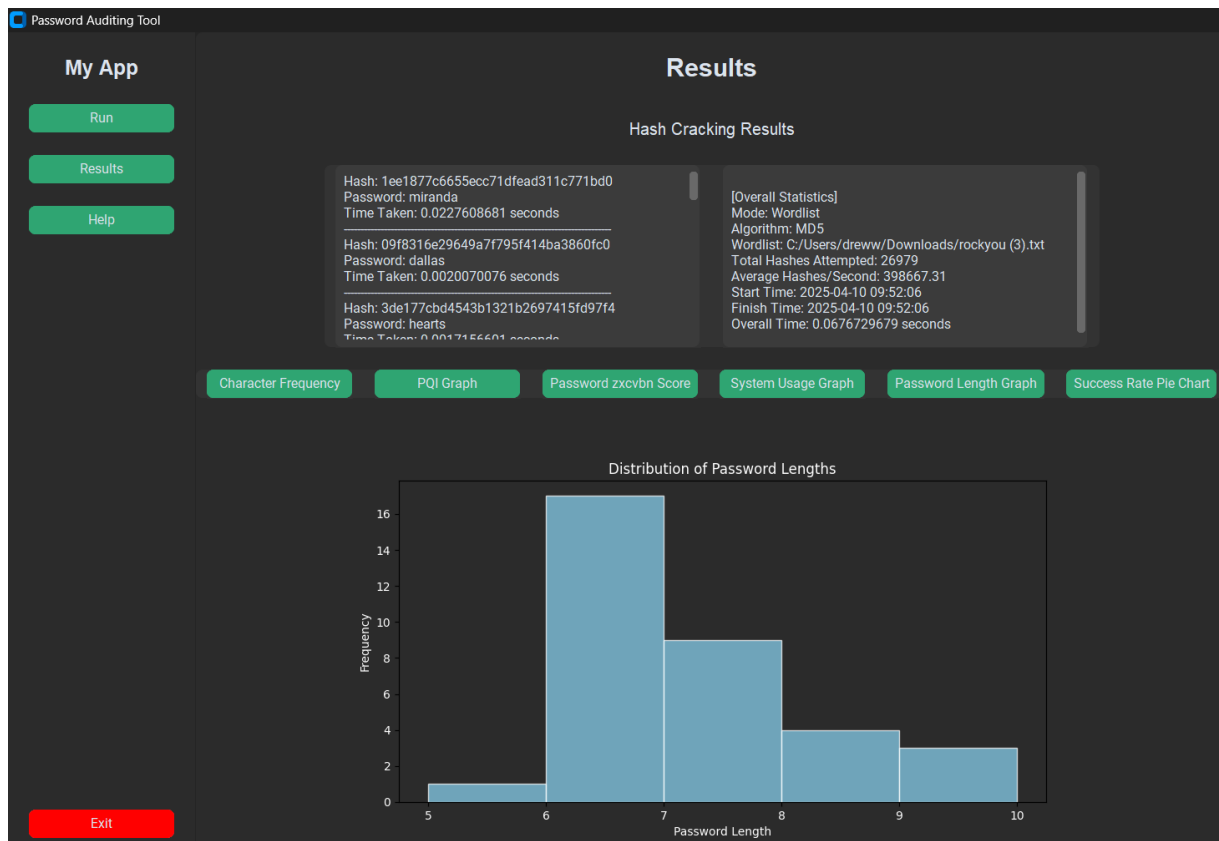


Figure 25 – A screenshot of the frontend results page, showing a histogram of password lengths of the passwords that have been found.

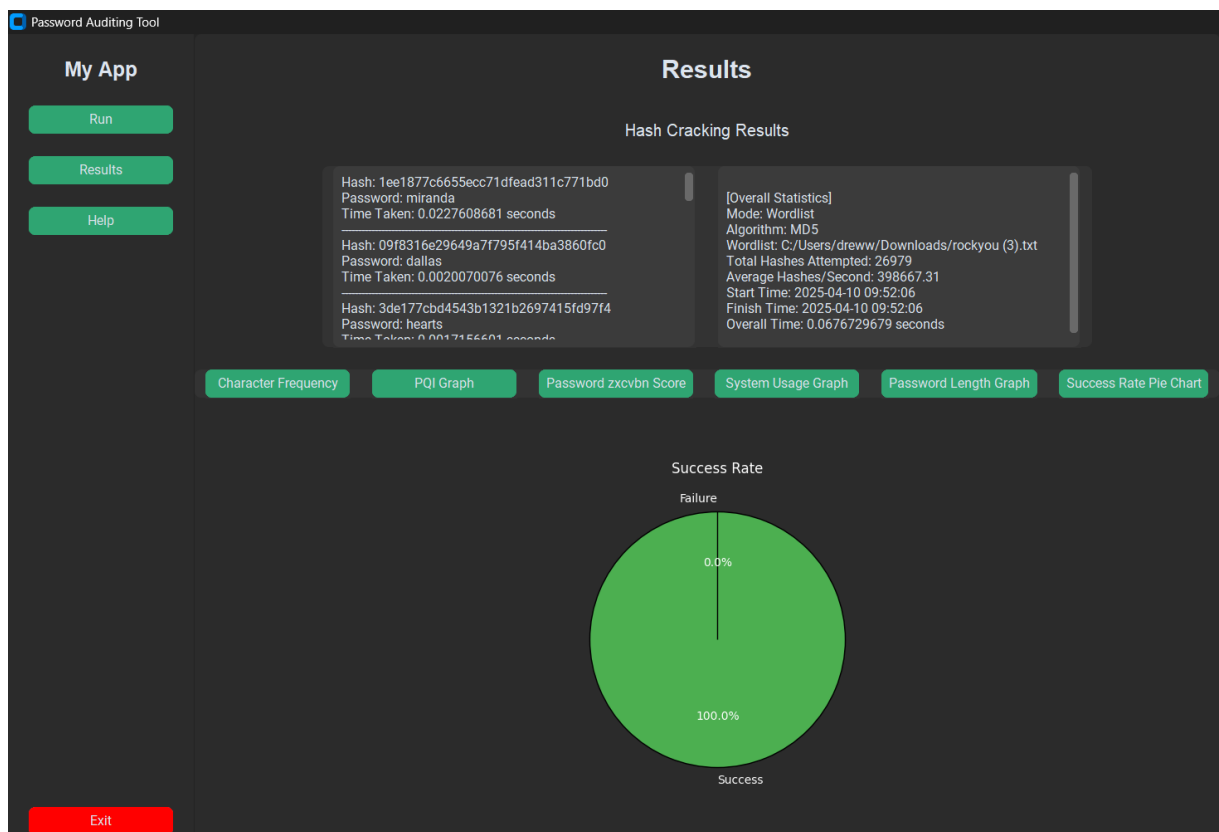


Figure 26 – A screenshot of the frontend results page, showing a simple pie chart which displays the success rate of passwords found and those that haven't been.

The final two graphs, shown in Figures 25 and 26 respectively, consist of a password length histogram and a pie chart illustrating the success rate of recovered passwords. The histogram provides a clear visualisation of the average, minimum, and maximum password lengths within the dataset. When used alongside the password strength graphs, it offers valuable insights into the correlation between password length and overall strength. The pie chart, on the other hand, serves as a simple yet effective way to display the success rate of password recovery, highlighting how easy or difficult it may be to audit a given set of password hashes.

3.7 – Testing

This section outlines the tests that were conducted, a brief description, and how they may influence the optimisation process. As emphasised in the introduction, testing is a critical component and objective of this project. It not only reinforces the validity of the application, and the claims presented in this report but also serves to help align the tool with the standards and benchmarks of similar systems discussed within the related work section.

Most of the testing focused on the backend functions, which also underwent the most significant changes during this process. Each function was reviewed in detail, assigned its own dedicated testing file, and tested using two main approaches. The first being unit testing to ensure the functionality of the backend was correct. This was then followed with performance testing to understand and reflect on any performance increases that may occur.

3.7.1 – Hash String Function

Unit testing the first function, hash string (responsible for hashing the attempted strings) quickly revealed three distinct issues. The first two stemmed from improper handling of hash lengths. As mentioned earlier, Ascon-Hash256 requires a fixed length of 32, while other Ascon modes can use arbitrary lengths (but they must still be explicitly specified). The function was failing to apply the correct hash length when comparing strings to a target hash, leading to inaccurate results in which the ascon hashes were half their intended length. The third issue arose from the use of MD4, which is not supported by the Hashlib library. To resolve this, the implementation was switched to use the Passlib library, mirroring the approach used for the LM hash.

```
===== 3 failed, 6 passed in 0.23s =====
PASSED [ 11%]PASSED [ 22%]PASSED [ 33%]PASSED [ 44%]PASSED [ 55%]FAILED [ 66%]
hash_string_tests.py:6 (test_known_hashes[Ascon-XOF128-test-85130352d6c207646d091b46d2bb0df6fe2189a7be71ac5826a4ce62e8262e8e])
'85130352d6c207646d091b46d2bb0df6' != '85130352d6c207646d091b46d2bb0df6fe2189a7be71ac5826a4ce62e8262e8e'

Expected : '85130352d6c207646d091b46d2bb0df6fe2189a7be71ac5826a4ce62e8262e8e'
Actual   : '85130352d6c207646d091b46d2bb0df6'
<Click to see difference>

FAILED [ 77%]
hash_string_tests.py:6 (test_known_hashes[Ascon-CXOF128-test-8a054af4c5f8b03e2085c68595da50f95516717ba787e5b217455d1608a5fe25])
'8a054af4c5f8b03e2085c68595da50f9' != '8a054af4c5f8b03e2085c68595da50f95516717ba787e5b217455d1608a5fe25'

Expected : '8a054af4c5f8b03e2085c68595da50f95516717ba787e5b217455d1608a5fe25'
Actual   : '8a054af4c5f8b03e2085c68595da50f9'
<Click to see difference>
```

Figure 27 – A set of two screenshots showing the issues encountered during the unit testing of the backend hash string function specifically for hashing ASCON-XOF, ASCON CXOF.

In contrast to the unit tests, the performance testing of the hash string function produced results that closely aligned with expectations (section 4.4.1). Each hashing algorithm was evaluated individually, with metrics recorded for hashes per second, total execution time, and CPU and memory usage. Overall, the results reflected the maximum achievable hash rates of the program, as the function directly calls the underlying hashing libraries. The only notable issue observed was the significant performance gap between the ASCON functions and the other hashing algorithms, a discrepancy that will be examined in more detail during the evaluation.

3.7.2 – Wordlist Function

The next function selected for testing was the wordlist function. As one of the program's more prominent and essential components, it underwent a more rigorous and comprehensive testing process, particularly in terms of performance evaluation.

I began with unit testing. Like the approach taken with the previous function, a variety of input scenarios were used. These included switching between different hashing algorithms, introducing invalid or non-existent target hashes, using empty wordlists, and deliberately triggering the function's built-in timeout mechanism. As expected, all unit tests were successfully passed, confirming that the wordlist function behaved as intended.

Performance testing was conducted by creating an extremely large wordlist file then measuring the speed at which a specific target word was found. This was a critical part

of testing process as it anticipated that performance trends observed in the hash string function would also appear here, it was reassuring to see this confirmed. On the one hand, the results validated that the wordlist function itself was not the source of the performance bottleneck. On the other hand, the data reinforced the earlier observation, a pronounced disparity between the ASCON algorithms and the others remained, indicating that the issue originates from the hash string function rather than the wordlist.

3.7.3 – Brute Force Function

Equally important as the wordlist function (if not more so, given its inherent performance issues) the brute-force function was tested in a similar format: unit tests followed by performance benchmarks. While the unit tests didn't lead to major changes, they did reveal a key quirk of the LM hash, which ignores case sensitivity. As a result, uppercase variations are treated as valid matches, indirectly helping with the performance of the function.

Besides this where the performance tests, as expected, their results reflected that of the previous two function. They followed the same trend, where the Hashlib-based functions consistently outperformed the custom ASCON implementations, which remain inherently slower. This reinforces the conclusion that, regardless of how optimised a hashing function might be, the computational overhead introduced by ASCON's design significantly hampers the overall performance. Beyond this confirmation, a few other noteworthy observations emerged. That being the CPU utilisation during the brute-force tests was noticeably lower compared to the wordlist-based function.

3.8 – Optimisation

Now that the three primary functions of the program have been examined and tested it was now time for changes to be made to them both in response to the results of the tests (section 4.4), but also in response to my general findings whilst using this "development build" of the application.

3.8.1 – Threading & Multiprocessing

One of the key changes made during this process which directly addressed the performance issues incurred from both algorithms was the impact upon the frontend GUI. Prior to changes made, running both algorithms would either freeze or crash the

frontend application. After further debugging it was found that the auditing process itself was being called by the Run function. Unintentionally blocking the main thread and freezing the GUI, therefore changing this (seen in Figure 28) and offloading these processes to a background thread allowed the GUI to maintain its responsiveness. Additionally adding visual confirmation of actions such as a loading icon appear once the run button has been pressed helped further aided with usability of the tool.

```
# Function to run the cracking process in the background thread
def run_crack(self):
    mode = self.select_mode.get()
    hash_type = self.select_hash_type.get()
    timeout = int(self.timeout_input.get())

    if not self.target_hash_path: # If no target hash is uploaded, produce an error
        self.controller.show_error("Error: Please upload a target hash file.")
        return

    if mode == "Wordlist" and not self.wordlist_path: # If mode is wordlist and no wordlist, produce error
        self.controller.show_error("Error: Please upload a wordlist file.")
        return

    # Show the loading indicator
    self.loading_label.pack(pady=10)
    self.loading_text = "Loading." # Start with one dot

    # Start the dot animation
    self.animate_loading_dots()

    # Run cracking process in the background
    threading.Thread(target=self.run_crack_in_background, args=(mode, hash_type, timeout)).start()

# Function that runs the cracker in the background and updates GUI once done
def run_crack_in_background(self, mode, hash_type, timeout):
    result = run_cracker(mode, hash_type, self.target_hash_path, self.wordlist_path, timeout)

    # Update GUI safely after processing is done
    self.after(0, self.display_results, result)
```

Figure 28 – A screenshot showing edited code so that the audit is being ran in a background thread instead of the main one.

3.8.2 – Implementing Additional Hashes

As part of the optimisation process, six additional hashing algorithms were introduced to improve the tool's flexibility and real-world applicability. These included SHA-224, SHA-384, BLAKE2b, BLAKE2s, SHA3-256, and SHA3-512. They were chosen for their widespread use, lightweight and auditable designs, and modern characteristics as alternatives to the existing options. Since all are natively supported by Python's Hashlib library, integration was seamless. Each was also tested to verify correct functionality and comparable performance.

```

===== 6 passed in 1.24s =====
PASSED [ 16%]SHA-224      : 772,515.15 H/s | Average CPU: 58.40% | Average Mem: 62.10807 MB | Total Time: 0.12945s
PASSED [ 33%]SHA-384      : 663,201.66 H/s | Average CPU: 33.25% | Average Mem: 61.35938 MB | Total Time: 0.15078s
PASSED [ 50%]BLAKE2b      : 882,462.21 H/s | Average CPU: 37.20% | Average Mem: 62.16406 MB | Total Time: 0.11332s
PASSED [ 66%]BLAKE2s      : 934,864.27 H/s | Average CPU: 43.00% | Average Mem: 61.39453 MB | Total Time: 0.10697s
PASSED [ 83%]SHA3-256     : 628,384.64 H/s | Average CPU: 0.00% | Average Mem: 62.17969 MB | Total Time: 0.15914s
PASSED [100%]SHA3-512     : 576,742.84 H/s | Average CPU: 66.73% | Average Mem: 61.47786 MB | Total Time: 0.17339s

```

Figure 29 – Wordlist mode test on new algorithms showing they functionally work

```

===== 6 passed in 56.04s =====
PASSED [ 16%]SHA-224      : 28,011.42 H/s | Average CPU: 3.73% | Average Mem: 61.54908 MB | Total Time: 8.79049s
PASSED [ 33%]SHA-384      : 27,744.35 H/s | Average CPU: 4.04% | Average Mem: 61.61675 MB | Total Time: 8.87373s
PASSED [ 50%]BLAKE2b      : 28,462.58 H/s | Average CPU: 4.45% | Average Mem: 61.64739 MB | Total Time: 8.64988s
PASSED [ 66%]BLAKE2s      : 24,886.85 H/s | Average CPU: 7.50% | Average Mem: 61.66382 MB | Total Time: 9.88992s
PASSED [ 83%]SHA3-256     : 25,217.51 H/s | Average CPU: 5.36% | Average Mem: 61.67163 MB | Total Time: 9.76175s
PASSED [100%]SHA3-512     : 25,093.18 H/s | Average CPU: 5.45% | Average Mem: 61.67188 MB | Total Time: 9.81143s

```

Figure 30 – Brute force mode test on new algorithms showing they functionally work

3.8.3 – Performance Improvements

Additional changes were made to the backend after the testing phase, aimed at improving overall performance. For the most part, these adjustments were successful. Optimising time delays, switching to “perf_counter” for more precise timing, and refining the way files were read and split all contributed to noticeable speed improvements across most functions. However, as observed previously, while the ASCON functions did show some performance gains, the gap between their execution times and those of other algorithms remained significant. This suggests that a deeper review of the ASCON’s implementation (within the evaluation) necessary to address the disparity and achieve further noticeable performance gains from each of the three functions.

3.9 – Summary

Overall, the methodology effectively delivers a high-level overview of the system’s development across design, implementation, and testing, including justifications for the chosen methods, tools, and changes made. Both design and implementation sections offer a chronological account of the entire process, from initial planning to final deployment. Whilst the testing phase successfully addressed the program’s main issues (particularly ASCON’s performance behaviour), although this required more time than initially expected. A deep insight into these limitations, along with a broader evaluation of the application’s success against its original aims and objectives, will be discussed further in the Evaluation and Conclusion chapters.

Chapter 4 – Results & Evaluation

4.1 – Introduction

This chapter provides a comprehensive summary of the project's lifecycle, including a detailed evaluation of the program. The evaluation is structured around four key areas.

First, it reviews the design methodology used throughout the project, specifically, the waterfall approach. After this there is a consideration of datasets and their effectiveness. Then, there is an in-depth comparison of the testing results outlined in Chapter 3 (both before and after optimisation). The fourth section assesses whether the requirements defined in the design stage were successfully met. Finally, the chapter concludes with the highlighting of the potential limitations of the work produced.

4.2 – Design Methodology

As previously mentioned, this project followed the waterfall software development lifecycle model, a decision made during the project planning phase. This choice shaped not only the development process but also the structure of this report. Given the project's time constraints, the waterfall model seemed an ideal fit. Its phase-based structure allowed for strict deadlines, aiding time management and ensuring a logical progression between stages.

The first two stages, Research & Planning (covered in Chapters 1 and 2) and Design (in Chapter 3) were completed smoothly and on schedule. However, the implementation and testing stages, which make up the remainder of Chapter 3, encountered delays. These setbacks led to the omission of both Functional Requirement 6, as well as non-functional Requirements 4 and 5. That being the logging system had to be removed as well as the efficiency improvements were not fully realised due to time spent addressing performance overheads caused by ASCON.

Once testing fell behind schedule, the rigid nature of the waterfall model caused delays in all subsequent phases, including this chapter and the subsequent conclusion. While the predefined milestones and deadlines helped maintain focus early on within the project, in hindsight, a more flexible development methodology (such as Agile) may have been more appropriate, as it could have better accommodated the unforeseen challenges and time delays incurred in the later stages of the project.

4.3 – Datasets

An important aspect of the evaluation involves assessing how the application performs when using different wordlists. This section focuses on testing the wordlist function with a variety of datasets to determine whether it inherently favours certain wordlists or if some offer better performance than others. The results presented here will highlight the impact each wordlist has on the function's effectiveness.

It is important to compare these results against one another within this section only. They should not be directly compared to the results in Section 4.4, as that section uses a significantly larger dataset in which the last word of the list is always the target word. With that being for performance testing reasons to establish a benchmark unlike the following tests which are assessing the effects of different wordlists.

For testing purposes, a consistent target password, “Password123” was selected. This choice ensures compatibility across all wordlists, while also providing a realistic mid-range benchmark, as it typically appears around the middle of most lists. Each test was conducted ten times, with the average (mean) result calculated to provide a reliable metric.

4.3.1 – Rockyou.txt

Rockyou.txt (Breault, 2017) is one of the most widely used password lists in the cybersecurity community and played a central role during the functionality testing and development of this project. Its popularity is largely due to its foundation in real-world data (containing approximately 14 million passwords) and its efficiency, offering strong performance even on lower-end systems thanks to its relatively compact file size.

```
===== 15 passed in 448.57s (0:07:28) =====
PASSED [ 6%]MD5 : Mean H/s: 794,090.54 | Avg CPU: 39.87% | Avg Mem: 60.91406 MB | Mean Time: 0.04226s
PASSED [ 13%]SHA-1 : Mean H/s: 706,812.21 | Avg CPU: 43.44% | Avg Mem: 60.94141 MB | Mean Time: 0.04755s
PASSED [ 20%]SHA-256 : Mean H/s: 371,388.74 | Avg CPU: 43.88% | Avg Mem: 61.00000 MB | Mean Time: 0.09045s
PASSED [ 26%]SHA-512 : Mean H/s: 371,998.89 | Avg CPU: 61.10% | Avg Mem: 61.01953 MB | Mean Time: 0.09099s
PASSED [ 33%]Ascon-Hash256 : Mean H/s: 1,983.36 | Avg CPU: 98.08% | Avg Mem: 61.06883 MB | Mean Time: 17.04289s
PASSED [ 40%]Ascon-XOF128 : Mean H/s: 3,229.08 | Avg CPU: 99.07% | Avg Mem: 61.06250 MB | Mean Time: 10.39251s
PASSED [ 46%]Ascon-CXOF128 : Mean H/s: 2,765.96 | Avg CPU: 98.60% | Avg Mem: 61.10602 MB | Mean Time: 12.22561s
PASSED [ 53%]NTLM : Mean H/s: 26,031.41 | Avg CPU: 97.03% | Avg Mem: 61.08203 MB | Mean Time: 1.31719s
PASSED [ 60%]LM : Mean H/s: 5,737.69 | Avg CPU: 93.89% | Avg Mem: 61.10581 MB | Mean Time: 0.24177s
PASSED [ 66%]SHA-224 : Mean H/s: 653,526.04 | Avg CPU: 50.14% | Avg Mem: 61.12109 MB | Mean Time: 0.05183s
PASSED [ 73%]SHA-384 : Mean H/s: 447,040.90 | Avg CPU: 48.26% | Avg Mem: 61.12070 MB | Mean Time: 0.07582s
PASSED [ 80%]BLAKE2b : Mean H/s: 590,096.76 | Avg CPU: 53.19% | Avg Mem: 61.13281 MB | Mean Time: 0.05688s
PASSED [ 86%]BLAKE2s : Mean H/s: 831,119.64 | Avg CPU: 50.68% | Avg Mem: 61.14062 MB | Mean Time: 0.04036s
PASSED [ 93%]SHA3-256 : Mean H/s: 501,164.59 | Avg CPU: 62.75% | Avg Mem: 61.14453 MB | Mean Time: 0.06694s
PASSED [100%]SHA3-512 : Mean H/s: 496,725.14 | Avg CPU: 64.20% | Avg Mem: 61.14453 MB | Mean Time: 0.06756s
```

Figure 31 – Performance results from using the rockyou.txt wordlist

4.3.2 – SecLists Xato-net

The SecLists password databank (Miessler, 2019) is an archive containing a wide range of credentials designed for security testing and research. As described by Miessler (2019), it includes “*usernames, passwords, URLs, sensitive data patterns, fuzzing payloads, web shells, and many more.*” One of the notable lists sourced from this archive is the “xato-net-10-million-passwords”, which despite being smaller in scale than rockyou.txt at just 10 million entries was selected for testing to observe how a smaller list performs under identical conditions.

```
===== 15 passed in 62.79s (0:01:02) =====
PASSED [ 6%]MD5 : Mean H/s: 941,504.91 | Avg CPU: 0.00% | Avg Mem: 60.86250 MB | Mean Time: 0.00805s
PASSED [ 13%]SHA-1 : Mean H/s: 927,232.70 | Avg CPU: 0.00% | Avg Mem: 60.88242 MB | Mean Time: 0.00823s
PASSED [ 20%]SHA-256 : Mean H/s: 927,877.56 | Avg CPU: 0.00% | Avg Mem: 60.89023 MB | Mean Time: 0.00817s
PASSED [ 26%]SHA-512 : Mean H/s: 744,141.38 | Avg CPU: 0.00% | Avg Mem: 60.90977 MB | Mean Time: 0.01019s
PASSED [ 33%]Ascon-Hash256 : Mean H/s: 3,135.77 | Avg CPU: 99.12% | Avg Mem: 60.91396 MB | Mean Time: 2.41640s
PASSED [ 40%]Ascon-XOF128 : Mean H/s: 4,560.13 | Avg CPU: 98.76% | Avg Mem: 60.91797 MB | Mean Time: 1.66123s
PASSED [ 46%]Ascon-CXOF128 : Mean H/s: 4,579.49 | Avg CPU: 99.62% | Avg Mem: 60.91797 MB | Mean Time: 1.65375s
PASSED [ 53%]NTLM : Mean H/s: 43,218.58 | Avg CPU: 90.12% | Avg Mem: 60.87472 MB | Mean Time: 0.17526s
PASSED [ 60%]LM : Mean H/s: 11,495.38 | Avg CPU: 91.36% | Avg Mem: 60.88666 MB | Mean Time: 0.09463s
PASSED [ 66%]SHA-224 : Mean H/s: 931,613.11 | Avg CPU: 0.00% | Avg Mem: 60.89062 MB | Mean Time: 0.00814s
PASSED [ 73%]SHA-384 : Mean H/s: 783,833.15 | Avg CPU: 0.00% | Avg Mem: 60.89062 MB | Mean Time: 0.00968s
PASSED [ 80%]BLAKE2b : Mean H/s: 969,164.86 | Avg CPU: 0.00% | Avg Mem: 60.90625 MB | Mean Time: 0.00784s
PASSED [ 86%]BLAKE2s : Mean H/s: 1,114,613.87 | Avg CPU: 0.00% | Avg Mem: 60.91367 MB | Mean Time: 0.00680s
PASSED [ 93%]SHA3-256 : Mean H/s: 678,663.76 | Avg CPU: 0.00% | Avg Mem: 60.91758 MB | Mean Time: 0.01116s
PASSED [100%]SHA3-512 : Mean H/s: 646,742.71 | Avg CPU: 4.88% | Avg Mem: 60.91797 MB | Mean Time: 0.01173s
```

Figure 32 – Performance results from using the Xato-net 10 million wordlist

4.3.3 – Pwned Password Top 100k List

The final list, and the smallest of the three, is derived from the popular service HavelBeenPwned (HavelBeenPwned, 2022). This list is a subset of a larger compilation curated by the National Cyber Security Centre (NCSC, 2012). It was chosen to assess whether the trend observed in the previous tests (where smaller wordlists, such as the Xato-net wordlist, led to performance improvements) would hold true.

```
===== 15 passed in 40.38s =====
PASSED [ 6%]MD5 : Mean H/s: 916,371.75 | Avg CPU: 0.00% | Avg Mem: 60.59336 MB | Mean Time: 0.00534s
PASSED [ 13%]SHA-1 : Mean H/s: 925,826.73 | Avg CPU: 0.00% | Avg Mem: 60.61641 MB | Mean Time: 0.00532s
PASSED [ 20%]SHA-256 : Mean H/s: 917,374.55 | Avg CPU: 0.00% | Avg Mem: 60.63203 MB | Mean Time: 0.00533s
PASSED [ 26%]SHA-512 : Mean H/s: 759,756.04 | Avg CPU: 0.00% | Avg Mem: 60.65234 MB | Mean Time: 0.00643s
PASSED [ 33%]Ascon-Hash256 : Mean H/s: 3,138.42 | Avg CPU: 98.48% | Avg Mem: 60.65609 MB | Mean Time: 1.55677s
PASSED [ 40%]Ascon-XOF128 : Mean H/s: 4,630.55 | Avg CPU: 99.02% | Avg Mem: 60.66016 MB | Mean Time: 1.05480s
PASSED [ 46%]Ascon-CXOF128 : Mean H/s: 4,411.77 | Avg CPU: 98.77% | Avg Mem: 60.66016 MB | Mean Time: 1.10859s
PASSED [ 53%]NTLM : Mean H/s: 41,901.16 | Avg CPU: 77.59% | Avg Mem: 60.66367 MB | Mean Time: 0.11662s
PASSED [ 60%]LM : Mean H/s: 11,229.35 | Avg CPU: 58.12% | Avg Mem: 60.69909 MB | Mean Time: 0.04178s
PASSED [ 66%]SHA-224 : Mean H/s: 886,272.70 | Avg CPU: 0.00% | Avg Mem: 60.69922 MB | Mean Time: 0.00552s
PASSED [ 73%]SHA-384 : Mean H/s: 751,794.31 | Avg CPU: 0.00% | Avg Mem: 60.69922 MB | Mean Time: 0.00650s
PASSED [ 80%]BLAKE2b : Mean H/s: 935,015.43 | Avg CPU: 0.00% | Avg Mem: 60.71484 MB | Mean Time: 0.00523s
PASSED [ 86%]BLAKE2s : Mean H/s: 1,006,551.74 | Avg CPU: 0.00% | Avg Mem: 60.72266 MB | Mean Time: 0.00486s
PASSED [ 93%]SHA3-256 : Mean H/s: 647,510.52 | Avg CPU: 0.00% | Avg Mem: 60.72656 MB | Mean Time: 0.00755s
PASSED [100%]SHA3-512 : Mean H/s: 647,344.79 | Avg CPU: 0.00% | Avg Mem: 60.72695 MB | Mean Time: 0.00755s
```

Figure 33 – Performance results from using the “HavelBeenPwned 100k” wordlist

4.3.4 – Dataset Summary

The results from the nine initially implemented hashing algorithms clearly show that the tool's performance is strongly affected by the size of the wordlist used. Across all three tested datasets, smaller wordlists consistently yielded faster hash rates, shorter execution times, and lower CPU usage. In contrast, larger wordlists (Rockyou) led to slower hash rates, longer execution times, and higher CPU usage. However, it's important to recognise that this trade-off comes with the benefit of improved password coverage and detection accuracy. Notably, performance gains begin to plateau beyond a certain point (there's little difference between the Xato and Pwned Efficiency although Xato is ten times bigger).

Each of the selected offers a practical compromise between speed and comprehensiveness. Their inclusion demonstrates how varying dataset sizes impact performance, offering insight into selecting the most suitable wordlist based on the desired balance of speed and accuracy. Overall, the Xato-net list stood out as the best performer, delivering strong efficiency despite maintaining a relatively large size. That said, all three wordlists are well-suited to the tool's intended research and educational purposes and are recommended for use in this context.

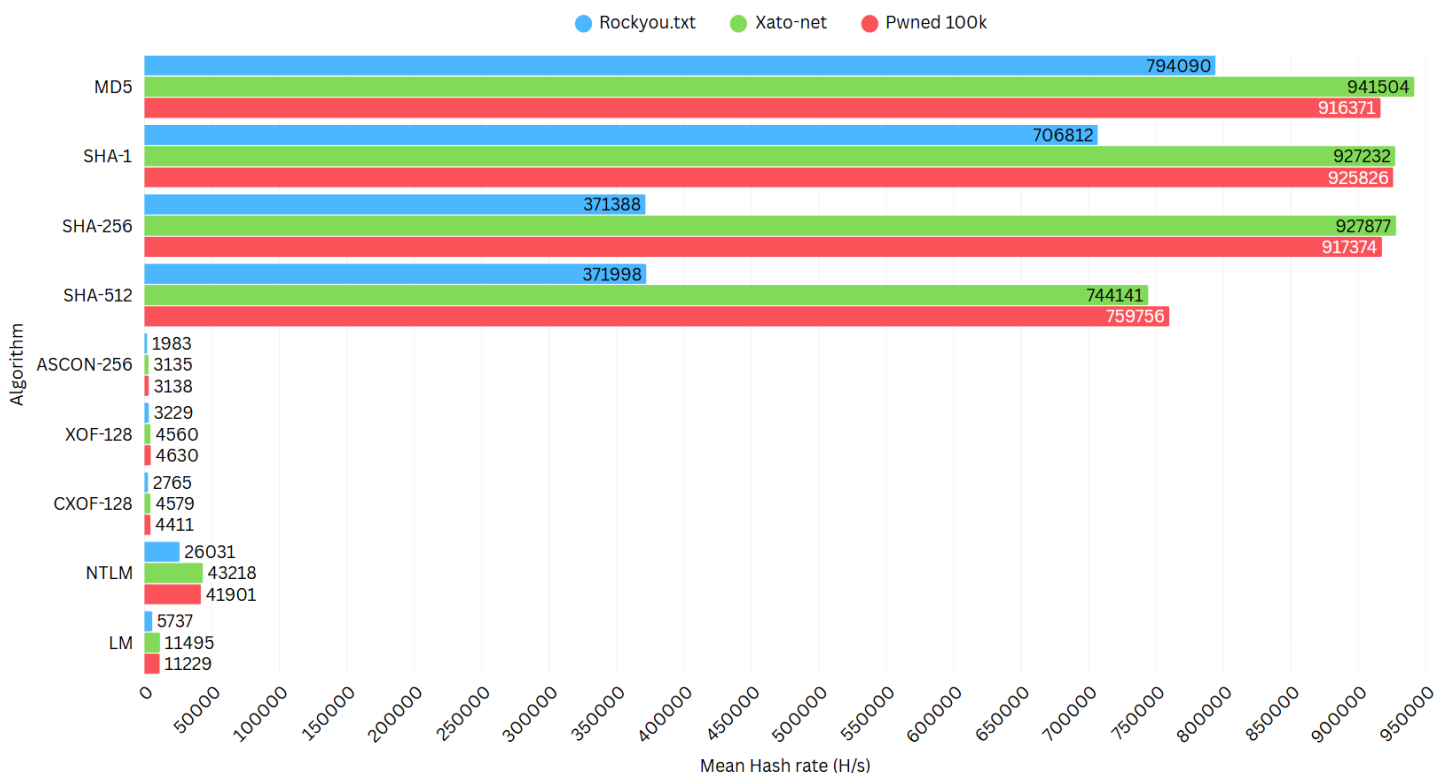


Figure 34 – Cross comparison of hash rates for the three different wordlists across the original nine hashing algorithms

4.4 – Performance Test Results

This section presents the actual performance results from the tests outlined in Section 3.7. Importantly, the three sets of results from test conducted prior to the optimisation process (namely, 4.4.1, 4.4.2, and 4.4.3) were each based on a single run. This approach was taken because the functions that they test are linear and exhibit minimal variation across runs. Additionally, this was for efficiency of time and the fact that these results also act as benchmarks for section 4.4.4 which, in contrast, was executed ten times to provide a more reliable assessment of performance gains.

4.4.1 – Hash String Function Results

As discussed in Section 3.7.1, the results from this function represent the maximum hash rate supported by the underlying hashing libraries, and by extension, the highest performance the program itself can achieve. As shown in Figure 35, most algorithms, particularly those supported by Python’s Hashlib (MD5, SHA-1, SHA-256, and SHA-512) performed well, delivering high hash rates with minimal CPU and memory usage. However, also noted in Section 3.7.1, a key issue identified was the significant performance gap between the ASCON family algorithms (Ascon-Hash256, Ascon-XOF128, and Ascon-CXOF128) and the others. These algorithms were noticeably slower and placed a much heavier load on CPU resources. This disparity is important to consider when interpreting the following test results, especially since this function operates at the highest level and is called by the others, potentially creating a performance bottleneck.

```
hash_string_tests.py::test_hash_string_performance PASSED [100%]
[Hash String Performance Report]
MD5           : 1,336,719.60 H/s | Average CPU: 0.00% | Average Mem: 60.53906 MB | Total Time: 0.00075s
SHA-1         : 366,018.82 H/s | Average CPU: 0.00% | Average Mem: 60.55078 MB | Total Time: 0.00273s
SHA-256       : 357,474.80 H/s | Average CPU: 0.00% | Average Mem: 60.56250 MB | Total Time: 0.00280s
SHA-512       : 321,471.05 H/s | Average CPU: 0.00% | Average Mem: 60.57031 MB | Total Time: 0.00311s
Ascon-Hash256 : 1,562.42 H/s | Average CPU: 88.62% | Average Mem: 60.58984 MB | Total Time: 0.64003s
Ascon-XOF128  : 1,977.94 H/s | Average CPU: 88.89% | Average Mem: 60.58984 MB | Total Time: 0.50558s
Ascon-CXOF128 : 1,967.59 H/s | Average CPU: 88.89% | Average Mem: 60.58984 MB | Total Time: 0.50824s
NTLM         : 21,780.85 H/s | Average CPU: 0.00% | Average Mem: 60.58984 MB | Total Time: 0.04591s
LM           : 9,632.74 H/s | Average CPU: 49.60% | Average Mem: 60.62305 MB | Total Time: 0.10381s
```

Figure 35 – A screenshot showing results from the performance test on the hash string function

4.4.2 – Wordlist Function Results

As previously mentioned in Section 3.7.2, the wordlist function closely mirrors the behaviour of the hash string function, reinforcing the conclusion that the performance

issues stem from the ASCON hashing algorithms or their implementation, rather than the function's internal logic. The same three ASCON variants, Hash256, XOF128, and CXOF128 were affected. As shown in Figure 36, these variants collectively added 87.08 seconds to the total execution time, a substantial delay compared to the other algorithms. This performance gap was further reflected in their hash rates, which were, in some cases, over 100 times slower than those of the Hashlib algorithms, highlighting the significant impact this overhead can have on the application's performance.

```

===== 9 passed in 99.11s (0:01:39) =====
PASSED [ 11%]MD5      : 776,830.56 H/s | Average CPU: 49.85% | Average Mem: 61.10547 MB | Total Time: 0.12873s
PASSED [ 22%]SHA-1    : 801,128.63 H/s | Average CPU: 0.00% | Average Mem: 61.64062 MB | Total Time: 0.12482s
PASSED [ 33%]SHA-256  : 805,423.40 H/s | Average CPU: 43.75% | Average Mem: 61.21484 MB | Total Time: 0.12416s
PASSED [ 44%]SHA-512  : 662,049.53 H/s | Average CPU: 50.20% | Average Mem: 61.24219 MB | Total Time: 0.15105s
PASSED [ 55%]Ascon-Hash256 : 2,155.59 H/s | Average CPU: 98.52% | Average Mem: 61.26791 MB | Total Time: 46.39107s
PASSED [ 66%]Ascon-XOF128 : 4,887.58 H/s | Average CPU: 99.79% | Average Mem: 61.21248 MB | Total Time: 20.46003s
PASSED [ 77%]Ascon-CXOF128 : 4,944.19 H/s | Average CPU: 99.47% | Average Mem: 61.17899 MB | Total Time: 20.22578s
PASSED [ 88%]NTLM     : 42,467.19 H/s | Average CPU: 96.86% | Average Mem: 61.18602 MB | Total Time: 2.35476s
PASSED [100%]LM       : 11,654.38 H/s | Average CPU: 98.73% | Average Mem: 61.28398 MB | Total Time: 8.58046s

```

Figure 36 – A screenshot showing results from the performance test on the wordlist function

4.4.3 – Brute Force Function Results

As with the previous tests, Figure 37 (brute-force test results) shows that ASCON remains slower than the Hashlib algorithms, though the performance gap is noticeably smaller than in the wordlist test. In fact, ASCON achieves a higher hash rate and lower CPU usage here, suggesting it may be better suited for brute-force scenarios. With this said, it doesn't imply that the brute-force function outperforms the wordlist function overall. As the wordlist approach demonstrated a significantly higher average hash rate 345,726.67 H/s compared to just 12,591.50 H/s for brute-force. This contrast was to be expected and reducing it was one of the key goals of the optimisation process.

```

===== 9 passed in 204.74s (0:03:24) =====
PASSED [ 11%]MD5      : 10,952.87 H/s | Average CPU: 11.22% | Average Mem: 61.27357 MB | Total Time: 22.41065s
PASSED [ 22%]SHA-1    : 10,657.74 H/s | Average CPU: 11.52% | Average Mem: 61.34360 MB | Total Time: 23.04214s
PASSED [ 33%]SHA-256  : 11,833.19 H/s | Average CPU: 11.49% | Average Mem: 61.33111 MB | Total Time: 20.74826s
PASSED [ 44%]SHA-512  : 18,481.29 H/s | Average CPU: 7.33% | Average Mem: 61.31612 MB | Total Time: 13.25822s
PASSED [ 55%]Ascon-Hash256 : 5,702.68 H/s | Average CPU: 4.47% | Average Mem: 61.32968 MB | Total Time: 42.89632s
PASSED [ 66%]Ascon-XOF128 : 9,112.13 H/s | Average CPU: 4.59% | Average Mem: 61.34375 MB | Total Time: 26.96580s
PASSED [ 77%]Ascon-CXOF128 : 8,684.20 H/s | Average CPU: 5.91% | Average Mem: 61.35136 MB | Total Time: 28.28712s
PASSED [ 88%]NTLM     : 22,141.82 H/s | Average CPU: 4.43% | Average Mem: 61.35918 MB | Total Time: 11.11964s
PASSED [100%]LM       : 15,757.62 H/s | Average CPU: 6.13% | Average Mem: 61.39194 MB | Total Time: 15.61099s

```

Figure 37 – A screenshot showing the results from the performance test on the brute force function.

4.4.4 – Optimised Function Results

Although Section 3.8 outlines the optimisation process, it does not directly present the results from the post-optimisation tests. These tests were identical to those in Sections 4.4.2 and 4.4.3 but conducted after the optimisation was applied, although notably, like mentioned previously, these tests were instead ran ten times each to verify if any performance gains had been truly achieved.

As shown in Figures 38 and 39, Generally algorithm hash rates increased, and time taken on average decreased the board (which was the purpose of the optimisation). With the exception for a few specific cases namely those with ASCON involved. This will be further considered within the summary section but as a general note these algorithms rarely showed any measurable gains. Reinforcing the idea that more substantial changes may be needed to reduce ASCON's overhead, an issue explored further in the concluding chapter.

```
===== 15 passed in 925.98s (0:15:25) =====
PASSED [ 6%]MD5 : 966,625.92 H/s | Avg CPU: 20.59% | Avg Mem: 62.43535 MB | Avg Time: 0.10371s
PASSED [ 13%]SHA-1 : 987,876.73 H/s | Avg CPU: 16.13% | Avg Mem: 61.67500 MB | Avg Time: 0.10137s
PASSED [ 20%]SHA-256 : 934,438.42 H/s | Avg CPU: 28.70% | Avg Mem: 62.47656 MB | Avg Time: 0.10730s
PASSED [ 26%]SHA-512 : 764,762.71 H/s | Avg CPU: 31.02% | Avg Mem: 61.76914 MB | Avg Time: 0.13102s
PASSED [ 33%]Ascon-Hash256 : 3,124.70 H/s | Avg CPU: 99.66% | Avg Mem: 61.93680 MB | Avg Time: 32.02018s
PASSED [ 40%]Ascon-XOF128 : 4,305.45 H/s | Avg CPU: 99.53% | Avg Mem: 61.66377 MB | Avg Time: 23.30129s
PASSED [ 46%]Ascon-CXOF128 : 4,095.19 H/s | Avg CPU: 99.54% | Avg Mem: 61.65948 MB | Avg Time: 24.45235s
PASSED [ 53%]NTLM : 41,090.79 H/s | Avg CPU: 96.91% | Avg Mem: 61.58982 MB | Avg Time: 2.43394s
PASSED [ 60%]LM : 11,340.81 H/s | Avg CPU: 99.25% | Avg Mem: 61.64062 MB | Avg Time: 8.82059s
PASSED [ 66%]SHA-224 : 933,047.73 H/s | Avg CPU: 15.87% | Avg Mem: 61.69896 MB | Avg Time: 0.10760s
PASSED [ 73%]SHA-384 : 797,402.41 H/s | Avg CPU: 24.56% | Avg Mem: 61.69883 MB | Avg Time: 0.12575s
PASSED [ 80%]BLAKE2b : 930,396.90 H/s | Avg CPU: 36.76% | Avg Mem: 61.71055 MB | Avg Time: 0.10821s
PASSED [ 86%]BLAKE2s : 1,012,843.40 H/s | Avg CPU: 28.04% | Avg Mem: 61.71836 MB | Avg Time: 0.09971s
PASSED [ 93%]SHA3-256 : 619,049.23 H/s | Avg CPU: 46.32% | Avg Mem: 61.72246 MB | Avg Time: 0.16199s
PASSED [100%]SHA3-512 : 630,127.42 H/s | Avg CPU: 39.90% | Avg Mem: 61.72617 MB | Avg Time: 0.15896s
```

Figure 38 – Wordlist performance test results after optimisation was conducted

```
===== 15 passed in 2686.27s (0:44:46) =====
PASSED [ 6%]MD5 : 25,466.02 H/s | Avg CPU: 4.27% | Avg Mem: 61.67915 MB | Avg Time: 9.69045s
PASSED [ 13%]SHA-1 : 24,069.31 H/s | Avg CPU: 5.36% | Avg Mem: 61.65688 MB | Avg Time: 10.23630s
PASSED [ 20%]SHA-256 : 22,730.58 H/s | Avg CPU: 4.89% | Avg Mem: 61.66570 MB | Avg Time: 10.84551s
PASSED [ 26%]SHA-512 : 21,228.94 H/s | Avg CPU: 5.39% | Avg Mem: 61.72339 MB | Avg Time: 11.60448s
PASSED [ 33%]Ascon-Hash256 : 4,945.73 H/s | Avg CPU: 5.47% | Avg Mem: 61.78562 MB | Avg Time: 49.62330s
PASSED [ 40%]Ascon-XOF128 : 6,888.01 H/s | Avg CPU: 5.72% | Avg Mem: 61.79687 MB | Avg Time: 35.62665s
PASSED [ 46%]Ascon-CXOF128 : 6,915.20 H/s | Avg CPU: 5.83% | Avg Mem: 61.80078 MB | Avg Time: 35.50177s
PASSED [ 53%]NTLM : 17,806.98 H/s | Avg CPU: 6.11% | Avg Mem: 61.50049 MB | Avg Time: 13.83209s
PASSED [ 60%]LM : 12,845.96 H/s | Avg CPU: 6.37% | Avg Mem: 53.09643 MB | Avg Time: 19.15973s
PASSED [ 66%]SHA-224 : 20,495.35 H/s | Avg CPU: 5.53% | Avg Mem: 53.10428 MB | Avg Time: 12.02070s
PASSED [ 73%]SHA-384 : 20,298.02 H/s | Avg CPU: 5.50% | Avg Mem: 53.10547 MB | Avg Time: 12.14565s
PASSED [ 80%]BLAKE2b : 20,996.81 H/s | Avg CPU: 4.71% | Avg Mem: 52.78562 MB | Avg Time: 11.73601s
PASSED [ 86%]BLAKE2s : 20,721.69 H/s | Avg CPU: 5.26% | Avg Mem: 52.73238 MB | Avg Time: 11.89049s
PASSED [ 93%]SHA3-256 : 20,244.54 H/s | Avg CPU: 5.37% | Avg Mem: 52.74219 MB | Avg Time: 12.17624s
PASSED [100%]SHA3-512 : 20,384.32 H/s | Avg CPU: 4.90% | Avg Mem: 52.74609 MB | Avg Time: 12.08738s
```

Figure 39 – Brute force performance test results after optimisation was conducted

4.4.5 – Optimisation Summary

Just by looking at the optimised function results (figures 38 and 39) it's evident that the performance outcomes fall short when compared to the benchmarks results from John the Ripper and Hashcat (Chapter 2). Well before the Implementation I understood that it would be difficult to even match their performance hence why the main performance target set in non-functional requirement 3 was instead set to averaging a hash rate of 10,000 H/s for both functions (across all algorithms). Through statistical analysis of the collected data from the optimisation results, we can critically assess the extent to which the tool meets this goal and identify areas for further improvement.

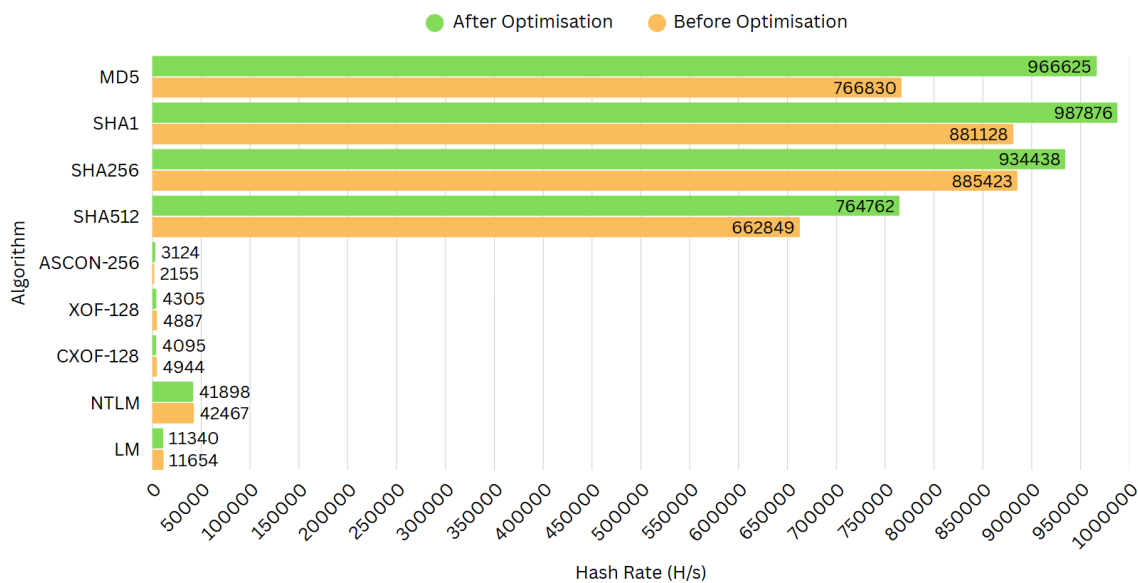


Figure 40 – A graph showing the hash rate improvements to **wordlist function** (for each algorithm) after the optimisation process (3.8)

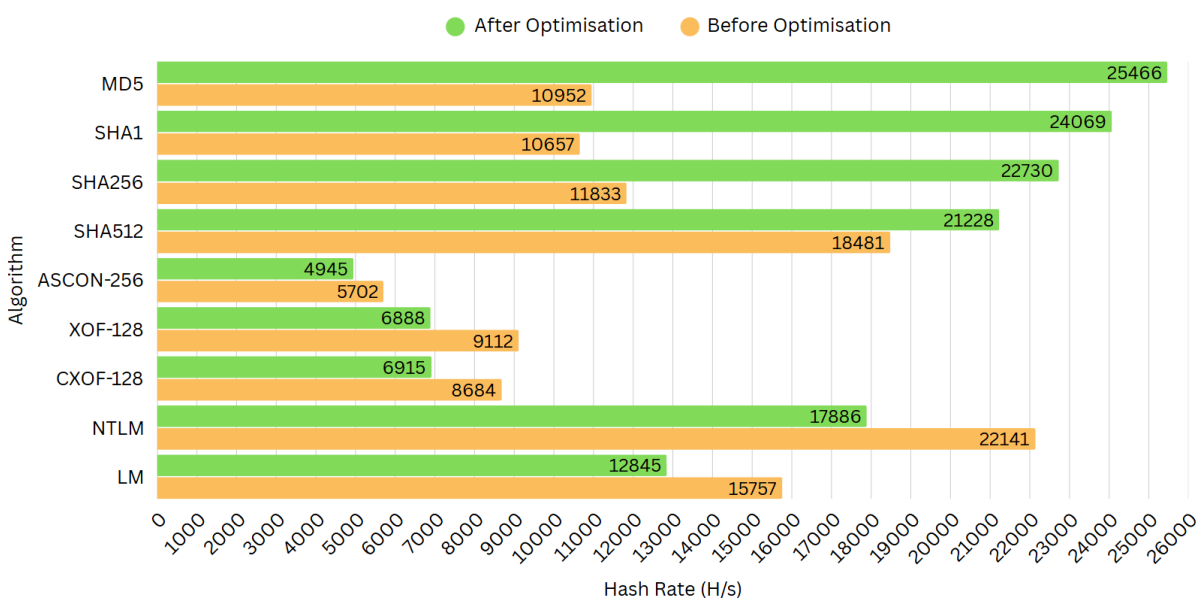


Figure 41 – A graph showing the hash rate improvement to the **brute force function** (for each algorithm) after the optimisation process (3.8)

After visualising the results from Figures 36, 37, 38, 39 I can successfully say that both optimised functions met this target (with the wordlist function averaging a hash rate of 408,495.89 H/s and the brute force averaging 16,441.33). Whilst another target set out in non-functional requirement 4 was to ensure that utilisation resources did not exceed past 75%. Through the following CPU utilisation visualisation, we can see that this was unfortunately not met (Figure 42).

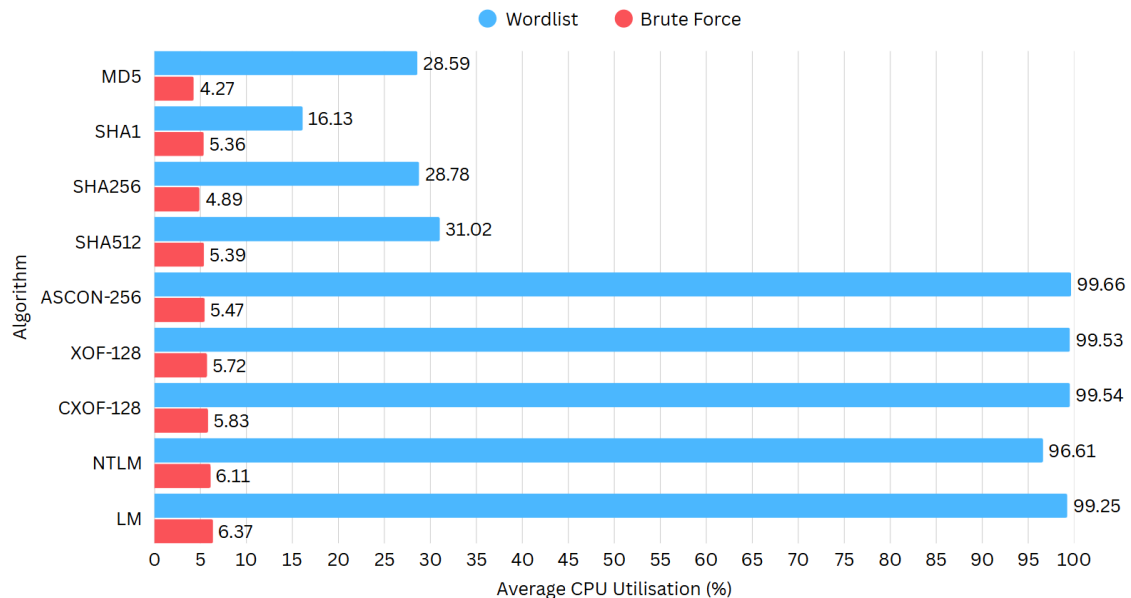


Figure 42 – A graph showing the average CPU utilisations found when conducting performance tests on the optimised wordlist and brute force functions.

As seen, this fails the requirement since multiple algorithms in the wordlist function make the CPU utilisation spike, specifically those dealing with ASCON functions as well as NTLM and LM. Interestingly there is also a lack of utilising from the Brute-force function generally. This is interesting as it may suggest there being untapped potential for improving CPU efficiency, which could enhance the performance of the brute-force approach.

The final graph (figure 43) shows the average memory complexity found after running the optimised performance test over ten rounds, it shows a result to be expected and one that isn't as volatile as that of the hash rates and CPU utilisation generally they stay around the 60MB mark for both modes. From the graph it also appears that generally there is good memory efficiency with minimal overhead between the algorithms. This also helps to suggest that the problems being encountered from ASCON are not memory related but may be more concerned with the CPU (hence the spikes in CPU utilisation as seen).

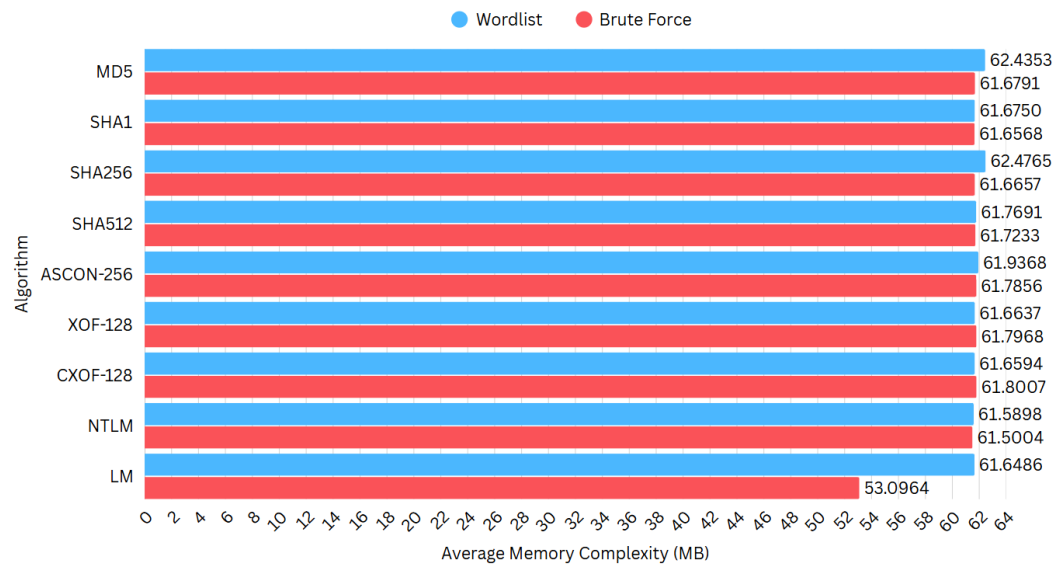


Figure 43 – A graph showing the average memory complexity found when conducting performance test on both the optimised wordlist and brute force functions.

Overall, the four figures in this section offer critical insights into the optimisation outcomes, revealing both successes and areas needing further attention. One of the most significant observations is the underperformance of the ASCON algorithms. Despite broader optimisation efforts, ASCON's throughput remains low, and its CPU utilisation disproportionately high compared to other algorithms. This supports the earlier hypothesis that the bottlenecks are not due to general program logic but are likely rooted in the inherent computational demands of the ASCON algorithm. Alternatively, the inefficiencies may stem from a suboptimal implementation of ASCON within the tool.

Another key takeaway is the contrasting outcomes related to the program's non-functional requirements. While non-functional requirement 3 (achieving an average hash rate of 10,000 H/s across all algorithms) was successfully met, requirement 4 (keeping CPU utilisation below 75%) was not. This disparity, particularly in the wordlist function, highlights inconsistencies in how the program handles system resources and suggests CPU-bound inefficiencies may be a key cause of slower performance.

In summary, the graphical analyses not only validate the program's ability to meet certain performance goals but also expose its limitations. These findings will be pivotal in the concluding chapter, guiding discussions on technical refinements, pinpointing inefficiencies, and outlining actionable directions for future development and optimisation.

4.5 – Requirement Fulfilment

A key part of evaluating the project is reviewing the requirements defined during the design phase. With, all high-priority objectives were successfully met, ensuring a solid foundation of success. Most medium-priority goals were also fully or partially achieved, further strengthening the outcome. The few unmet requirements were intentionally deprioritised (due to time constraints), as they did not affect the project's aims and objectives. Said requirement evaluation results are shown below.

4.5.1 – Functional Requirement Fulfilment

No.	Description	Priority	Met?
1	Accept user input and file uploads through a GUI interface. This includes uploading a target hash, wordlist and selecting the hashing algorithm to use.	High	Fully
2	Make use of two methods for auditing passwords specifically a brute-force methods and then a method using a password list.	High	Fully
3	Support hashing of all 3 hashing modes of ASCON (256, XOF128 and CXOF128).	High	Fully
4	support other commonly used and altered broken hashes such as MD5, SHA-family, NTLM, LM, Bcrypt and Scrypt.	Medium	Partially
5	Output results of an audit, by both generating visualisations and showing key metrics, and figures collected from the data.	Medium	Fully
6	Maintain a log or database of all audits that have been conducted using the application. These should be able to be sorted, filtered, exported and viewed by the user.	Medium	Not met
7	Provide a user friendly and intuitive GUI allowing for real time viewing of the progress of an audit and provide feedback to the actions users make.	Low	Partially
8	Ensure comprehensive error handling, including validation of user inputs, incorrect file formats, invalid hash types, and management of system resource consumption.	Medium	Fully
9	Make use of multi-processing such as threading and enable GPU processing.	Medium	Partially

4.5.2 – Non-Functional Requirement Fulfilment

No.	Description	Priority	Met?
1	The tool should be cross compatible upon any system that supports the use of python specifically on Linux, MacOS and Windows.	Medium	Fully
2	The system should have response times for operations under 5 seconds for processes that aren't running an audit. And a time for when running an audit to ensure the program doesn't continue running for a very long time.	High	Fully
3	General Hashing speeds for both audit methods and across all algorithms should be around 10,000 Hashes per second.	High	Fully
4	The program should make use of caps of system utilisation to ensure its not being overused, with a limit of 75% on both CPU and memory utilisation. This will hamper performance but increase reliability.	Medium	Not Met
5	Generated report logs should be saved locally to the machine and accessible for 30 days before being deleted by the application. Additionally identifiable data should be removed from them.	Medium	Not Met
6	Text file inputs should be scalable allowing for very large txt files to be uploaded without performance degradation.	Medium	Partially
7	The code should be written in a user friendly and easily understandable manor with lots of comments to aid with open-source collaboration.	Low	Fully
8	Live data measuring should be read into a txt file and then later read from a txt file instead of being directly interacted with by the application as processing such a vast amount would degrade performance.	Medium	Fully

4.6 – Limitations

The following section outlines a concise summary of the limitations identified throughout the course of this work. These include overarching constraints of the project, as well as specific limitations related to the program itself. The list encompasses limitations present from the project's inception, oversights encountered during development, and insights gained during the testing phase. These limitations will be directly addressed when considering the path of future work.

No.	Limitation
1	The overall performance of the program falls significantly short when compared to the alternative approaches discussed in Chapter 2, both of which demonstrate considerably faster execution. This performance gap can be attributed to several factors: the lack of GPU acceleration, the choice of Python as the implementation language, and the additional overhead introduced by GUI components.
2	The functionality for saving logs (functional requirement 6) and exporting audit results was never implemented. While not directly tied to any specific object within the project, including this feature would have enhanced the program's completeness and practicality, making it more suitable for real-world applications.
3	Ideally, the program should support a broader range of hashing algorithms, the greater the variety, the more versatile and applicable the program becomes. Currently, the absence of more robust algorithms such as Bcrypt and Scrypt significantly limits its usefulness when compared to alternative tools.
5	Although ASCON hashes have been functionally implemented, in practice they do not compete with the performance speeds and times of the other algorithms (seen in figure 36), making them harder to be applicable to real world situations.
6	The non-functional requirement 4 originally set a limit of 75% for both CPU and memory usage. In retrospect, this was an unrealistic target, as optimal program performance often requires intensive resource consumption. Moreover, system resource management within the program has been inconsistent: during ASCON audits, CPU usage can spike to over 99%, while during brute-force operations, it may drop below 5%, indicating periods of significant underutilisation.
7	Due to time constraints, the number of visualisations implemented on the report page was limited. As outlined in the original wireframe designs, additional components such as a pattern analysis plot and an attempt rate plot were planned but could not be completed within the project timeline.

Chapter 5 – Discussion & Conclusion

5.1 – Introduction

This final chapter serves as an overall conclusion, evaluating the complete success of the project in relation to its original aims and objectives. It also has the purpose of reflection on the personal lessons learned throughout the process and outlines potential directions for future work.

5.2 – Aim & Objective Fulfilment

The original overarching aim of the project was to develop a password auditing tool that integrates the ASCON hashing algorithm. This aim was broken down into a series of specific objectives, the successful completion of which would determine the overall success of the project. In this section, each objective will be reviewed in detail to assess whether it was achieved, thereby providing a measure of the project's success.

5.2.1 – Objective 1 – *Integrate ASCON's hashing functionality into an auditing tool to assess its performance compared to traditional hashing algorithms.*

This objective consists of two key components. The first involves implementing ASCON, which is demonstrated in Sections 3.5 and 3.6 where the implementation process is documented. The second component, evaluating ASCON's performance in comparison to traditional hashing algorithms, is addressed during the testing phase in Section 3.7 and further analysed in the evaluation found in Section 4.3. While the results indicate that ASCON's performance falls short it remains that functionally ASCON has been implemented and that the comparison was made between the ASCON and the other algorithms, thereby fulfilling this objective.

5.2.2 – Objective 2 – *Perform testing and debugging on the password auditing tool to ensure its accuracy, efficiency, and reliability*

Objective 2 was more challenging to address than Objective 1. While the testing and optimisation phases (Sections 3.7 and 3.8) did improve the tool's reliability and efficiency, I'm not fully satisfied with its overall performance. Given more time, I would have liked to address key limitations, such as the absence of GPU acceleration and the performance overheads associated with ASCON. These improvements could have made the tool far more suitable for real-world use. That said, the testing process was

thorough, including performance testing (CPU usage, memory consumption, and execution speed), as well as unit and functional testing of backend functions. Overall, this work was essential to partially meeting Objective 2, even if there's still room for future enhancement.

5.2.3 – Objective 3 – *Analyse and compare two examples of already existing password auditing tools against my tool*

Objective 3 was primarily addressed in Chapter 2 (Background Research), where I analysed two widely used password auditing tools: Hashcat and John the Ripper. I explored the features that contributed to their popularity and benchmarked their hash rates using common hashing algorithms. This analysis helped inform my implementation, particularly in adding features like multi-threading. It also allowed me to better understand the limitations of my own tool. Without this benchmarking, I wouldn't have identified the performance gap between my tool and the others during the evaluation phase. Therefore, I would also deem this objective to also be a success.

5.2.4 – Objective 4 - *produce a comprehensive documentation report that serves both as a user guide and a final analysis of the project*

Objective 4 is somewhat difficult to assess, as it relies heavily on the quality and impact of this report itself. However, I believe the report can be evaluated in two keyways. First, as an educational resource: it should help users or readers understand not only ASCON, but also password security more broadly. I believe this is achieved primarily through the background research and the explanation of the development process in Section 3.6. Second, the report should offer a clear, unbiased assessment of the tool, honestly addressing its limitations while highlighting the contributions I've made. I believe Chapters 4 and 5 (Evaluation and Conclusion) meet this goal. Therefore, I feel the project successfully meets Objective 4.

In summary, three out of the four objectives were fully met, with Objective 2 being only partially achieved due to my personal standards. However, it's worth noting that performance is an aspect of development that is constantly a factor that can be improved. Besides that, with the successful integration of ASCON, thorough comparative analysis, and the creation of a comprehensive report indicate that the project has largely met its original aim.

5.3 – Personal reflection

Carrying out a project of this scale has been a completely new experience for me, particularly due to its independent and self-reliant nature. As a result, the knowledge and experience I gained throughout the process have been personally invaluable.

Much of what I learned came from background research, particularly when studying tools such as John the Ripper and Hashcat. I developed a deeper understanding of the techniques these tools employ and the reasoning behind them. Another major learning area was ASCON, where I gained a detailed understanding of the cryptosystem, especially its hashing algorithm. Dissecting and implementing it hands-on significantly enhanced my learning experience.

The technical outputs of the project provided important insights. They highlighted just how powerful and optimised existing tools like John the Ripper and Hashcat are, especially compared to my own tool, which could not match their performance. The results also emphasised that algorithm complexity plays a major role in performance; for example, ASCON's performance was noticeably slower compared to simpler algorithms like MD5 and SHA.

Throughout the project, I developed several practical skills. Managing my time effectively became essential, especially when following the waterfall methodology, which required me to meet strict deadlines at each stage. The project also strengthened my problem-solving abilities during development and my critical thinking skills during testing and evaluation.

Overall, I consider the project a success, with the main objective achieved. However, part of me wishes the outcome had reached a higher level of performance, closer to that of the established tools. Because of this, I prefer to view the project as a "proof of concept", showing that ASCON can be functionally implemented into an auditing tool and that it is quite easily achievable. That said, there are design decisions I would approach differently if given the chance, many of which are discussed in the limitations section, but others would include the choice of design methodology (rather using Agile) and providing myself with more time for the testing phase. It's also important to note that the work here is far from complete, with many opportunities for future development and improvement, some of which will be mentioned within the following section.

5.4 – Future work

Like most projects, the work presented in this report remains far from complete, with many potential avenues for future development still available. This section will briefly explore these opportunities, highlighting both the weaknesses they aim to address, and the potential improvements that they could make. Each suggestion is based on lessons learned throughout the project and points toward meaningful areas for future improvement.

One potential way to build on the work conducted in this project would be to implement ASCON hashing into a more mainstream format. Now that this proof of concept has demonstrated that such an implementation is both functional and achievable, the logical next step is to integrate ASCON into widely used tools, helping to drive its broader adoption. This could involve application-level integration, such as adding ASCON hashing to tools like John the Ripper or Hashcat, or library-level integration by incorporating it into something like Python's Hashlib. Since this is currently the only known implementation of ASCON in auditing tools, expanding it into more mainstream applications would help address the issue of limited awareness. It would not only promote ASCON's adoption and highlight its brute-force vulnerabilities, but also further validate the work carried out in this project. Successfully achieving this would represent an ideal (outer scope) end goal and a significant milestone for the project.

Another possible direction for future work is to continue developing the tool created in this project. Unlike the previously mentioned approach, this would be less straightforward and would require significant time, effort, and dedication, as the tool remains incomplete in several key areas and would involve many additional steps.

The first way to improve the current tool is by investigating the performance overhead introduced by ASCON itself. This is the highest priority, as the ability to efficiently run ASCON audits is one of the main goals of the project. My initial step (which I have already begun exploring) is to examine the ASCON implementation. Currently, I call the hashing function directly, but by using the dedicated ASCON library may prove to be a better alternative. Given more time for the project, I would have already conducted a thorough comparison between the two. Alternatively, if this approach does not yield sufficient improvements, I would then explore other methods of increasing the hashes per second, which brings me to the second way I would enhance the tool.

The second way I'd improve the tool and one which would totally revise it would be with the inclusion of GPU acceleration, this increase performance across the board for most algorithms including ASCON. This is basically a way of utilising the performance from a graphical processing unit (GPU) as well as the CPU. Whilst the CPU would run the program itself the repetitive process of making guesses would be done via the GPU, which is much more adapted for heavier workloads, thereby significantly increasing the hash rate, reducing CPU and memory utilisation and generally improving performance of the program, which was one of the main limitations.

The third way I would improve the tool is by expanding its functionality. This involves addressing the incomplete requirements and three key aspects highlighted in the limitations section: adding logging functionality, supporting more algorithms, and introducing additional visualisations. First, the logging functionality should be prioritised, as it was part of the original requirements. Having a simple and effective method for saving and exporting results directly through the GUI would be a standout feature, helping to make the tool more independent and distinctive compared to similar systems. In addition, expanding the range of supported algorithms and enhancing number of visualisations would make the tool more practical and versatile, broadening its potential applications and appeal.

5.5 – Summary

In summary, the original goal of the project, to build an auditing tool from the ground up with ASCON support, has been successfully achieved, although not without some imperfections. Several minor issues remain, which was expected given the limited timeframe. With more time, I would have addressed the limitations outlined earlier, and in fact, I am already working on some of the future improvements mentioned. Beyond the technical achievements, completing this project has been incredibly valuable for my personal development, significantly boosting my confidence in both my coding skills and my broader computer science abilities and it is an experience that I will carry forward into my future work.

References

Bertoni, G., Daemen, J., Peeters, M. and Assche, G. (2007). *Sponge Functions*. [online] Available at: <https://keccak.team/files/SpongeFunctions.pdf>.

Computer Security Division, I.T.L. (2023). *Announcing Lightweight Cryptography Selection / CSRC*. [online] CSRC | NIST. Available at: <https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon>.

danielmiessler (2019). *danielmiessler/SecLists*. [online] GitHub. Available at: <https://github.com/danielmiessler/SecLists>.

Dobraunig, C., Eichlseder, M., Mendel, F. and Schl  ffer, M. (2019). *Submission to NIST*. [online] Available at: <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf>.

Dobraunig, C., Eichlseder, M., Mendel, F. and Schl  ffer, M. (2021). Ascon v1.2: Lightweight Authenticated Encryption and Hashing. *Journal of Cryptology*, 34(3). doi: <https://doi.org/10.1007/s00145-021-09398-9>.

Eichlseder, M. (2019). *Ascon – Lightweight Cryptography*. [online] Tugraz.at. Available at: <https://ascon.isec.tugraz.at>

Eichlseder, M. (2023). *Python implementation of Ascon*. [online] GitHub. Available at: <https://github.com/meichlseder/pyascon>.

Eichlseder, M. (2024). *Ascon – Specification*. [online] Tugraz.at. Available at: <https://ascon.isec.tugraz.at/specification.html>.

Hashcat (2018). *Hashcat - advanced password recovery*. [online] Hashcat.net. Available at: <https://hashcat.net/hashcat/>.

HavelBeenPwned (2022). *GitHub - HavelBeenPwned/PwnedPasswordsDownloader: A tool to download all Pwned Passwords hash ranges and save them offline so they can be used without a dependency on the k-anonymity API*. [online] GitHub. Available at: <https://github.com/HavelBeenPwned/PwnedPasswordsDownloader>.

Kodirov, K., Lee, H.-J. and Lee, Y.S. (2024). A Closer Look at Attacks on Lightweight Cryptosystems: Threats and Countermeasures. *Lecture Notes in Computer Science*, 14532, pp.171–176. doi: https://doi.org/10.1007/978-3-031-53830-8_17.

Ma, W., Campbell, J., Tran, D. and Kleeman, D. (2010). *Password Entropy and Password Quality*. [online] IEEE Xplore. doi: <https://doi.org/10.1109/NSS.2010.18>.

McKay, K.A., Bassham, L., Turan, M.S. and Mouha, N. (2017). Report on lightweight cryptography. [online] doi: <https://doi.org/10.6028/nist.ir.8114>.

NCSC (2012). [online] Ncsc.gov.uk. Available at: <https://www.ncsc.gov.uk/static-assets/documents/PwnedPasswordsTop100k.txt>.

NCSC (2023). *Next steps in preparing for post-quantum cryptography*. [online] www.ncsc.gov.uk. Available at: <https://www.ncsc.gov.uk/whitepaper/next-steps-preparing-for-post-quantum-cryptography>.

Openwall (2019). *John the Ripper password cracker*. [online] Openwall. Available at: <https://www.openwall.com/john/>.

Python (2024). *hashlib — Secure hashes and message digests — Python 3.8.4rc1 documentation*. [online] docs.python.org. Available at: <https://docs.python.org/3/library/hashlib.html>.

Python Software Foundation (n.d.). *What Is Python? Executive Summary*. [online] Python. Available at: <https://www.python.org/doc/essays/blurb/>.

The MITRE Corporation (2017). *Brute Force, Technique T1110 - Enterprise | MITRE ATT&CK®*. [online] attack.mitre.org. Available at: <https://attack.mitre.org/techniques/T1110/>.

Wheeler, D. (2012). *zxcvbn: realistic password strength estimation*. [online] Dropbox. Tech. Available at: <https://dropbox.tech/security/zxcvbn-realistic-password-strength-estimation>.

Yu, X., Liu, F., Wang, G., Sun, S. and Meier, W. (2023). *A Closer Look at the S-box: Deeper Analysis of Round-Reduced ASCON-HASH*. [online] Available at: <https://csrc.nist.gov/csrc/media/Events/2023/lightweight-cryptography-workshop-2023/documents/accepted-papers/02-a-closer-look-at-the-S-box.pdf>

Breault, J. (2017) The RockYou Wordlist. Available at:

<https://github.com/PentesterLab/rockyou>

Crackstation's (2021) Crackstation's Wordlist. Available at:

<https://crackstation.net/big-wordlist.php>

Zion3R (2024). *blackploit/hash-identifier*. [online] GitHub. Available at:

<https://github.com/blackploit/hash-identifier>.