B.Sc. (Hons) in Software Development

Ollscoil Teicneolaíochta an Atlantaigh

Atlantic Technological University

From Sound to Security

By
James Doonan

April 27, 2025

Minor Dissertation

Department of Computer Science & Applied Physics,
School of Science & Computing,
Atlantic Technological University (ATU), Galway.

Contents

List of Figures

List of Tables

Abstract

The persistent challenge of balancing security and usability in password authentication has created vulnerable systems. Users resort to predictable practices due to cognitive overload. This dissertation presents a novel voice-based password generation system that harnesses vocal biometrics and artificial intelligence to create highly secure passwords without compromising memorability.

The system extracts distinctive voice features, including Mel-Frequency Cepstral Coefficients, spectral centroids and tempo measurements from vocal inputs. These features are processed through Claude AI to generate complex passwords uniquely tied to individual vocal characteristics.

A comprehensive security testing framework evaluates these passwords against multiple attack vectors: brute-force simulations, dictionary attacks, specialised Hashcat techniques and advanced AI models (GPT-4 and Claude). Empirical results demonstrate that voice-based passwords achieve superior security metrics. This includes 56% longer average cracking times and higher entropy values (41.52-43.02 bits) compared to traditional passwords. Most significantly, none of the AI-generated passwords were successfully cracked or guessed by any attack method, while 80% of traditional passwords were compromised.

This research project provides concrete evidence that voice-based password generation presents a viable path towards resolving the security-usability paradox in authentication systems. It offers enhanced protection while maintaining the natural connection to human characteristics that supports memorability and adoption.

Chapter 1

Introduction

Today's digital world has been a blessing in many ways. The world has never been so connected. For example, in an instant a person can video call or message someone across the globe and get an immediate response. Although the digital age is convenient and impressive, it comes with unique challenges, particularly in the realm of security. Authentication is essential in modern digital systems, protecting access to private data including photos, banking details, and personal contact information. The cornerstone of online security, the standard alphanumeric password, is increasingly showing its limitations. While the idea of an alphanumerical password is conceptually simple (a shared secret that's kept between the user and the system), passwords in general can suffer from flaws, such as the inherent trade-off between usability and security[1]. A strong password consists of many characters, including special characters, uppercase and lowercase letters, and numbers. While secure, this can be quite difficult for people to remember. As a result, users can often develop poor habits[2]. These include writing passwords down, reusing the same password across multiple accounts, or creating simple passwords that are easy to remember[3]. This is very common, and such passwords can be easily hacked or guessed. Most commonly, people choose passwords that are personally meaningful, such as a partner's name or their child's date of birth[4]. The problem is made worse by the increasing number of accounts an individual manages, as well as the widespread use of mobile phones, where entering passwords on soft keyboards can be inconvenient[1]. These vulnerabilities are increasing and have led to data breaches and subsequent leaks. This highlights the urgent need for more secure and user-friendly authentication methods.

Fortunately, there has been a response to these issues. Researchers have been investigating different forms of authentication, such as biometric systems that rely on unique physical traits like facial features, fingerprints and graphical passwords that use images instead of text[1]. While promising, these also come with limitations. For example, biometrics can be expensive and may not be suitable to

users who can't afford that level of protection[5]. Graphical passwords may not be accessible for visually impaired users and are not usable in scenarios where screens are unavailable[3]. A newer area of investigation is the use of music and sound for authentication. This is particularly interesting because it draws on the well-documented link between music and individuals' memories[2]. Studies have shown that music has an incredible ability to spark emotions and autobiographical memories[4]. It has been shown that melodies are readily recognised and can be recalled by people even if they have no musical background[1]. The inherent rhythm, tempo and melodic contour of music also contributes to its memorability. This makes music and sounds an interesting direction to go for password authentication[3].

This project addresses the limitations of current password methods by exploring the feasibility of using musical and voice-based passwords as a viable alternative to traditional alphanumeric passwords. It investigates how a voice-based system can generate passwords through the use of AI. It aims to create an innovative system for password creation using music, sound and voice, while also taking a thorough look at the security and usability in the face of modern threats, such as AI-driven attacks[3]. This project is important in aiding research and seeks to contribute to the ongoing discussion about alternative authentication methods whilst also evaluating the efficacy of musical and voice-based password systems for mainstream adoption[4, 6].

1.1 Significance of the Study

The challenge of password security extends beyond the field of software development, affecting users across various domains. Every day, users fall victim to cyberattacks due to the use of weak and easily compromised passwords[2, 4]. Elderly people nowadays have become a target of over-the-phone hacking, phishing and social engineering techniques[4]. It may not be long before an AI hacking tool cracks their password without their knowledge. Kumar et al. (2012) mentions that users frequently reuse passwords across multiple accounts[3]. The increasing complexity and the amount of passwords people need to manage has led to insecure user practices and this has raised the need for a more secure and user-friendly approach to password authentication[1]. Exploring other approaches, such as music and voice-based systems, could lead to a way of keeping your private information not only more secure but also more memorable. Research has already shown that musical passwords enhance memorability and could provide a viable alternative to traditional passwords[7]. A study on music based authentication highlights that users naturally recall melodies better than complex alphanumerical passwords, which in theory would make them less prone to forgetting while still maintaining security[1]. To add, Lutz et al. (2020) showed that sound-based feedback can massively improve password strength awareness. In their research on interactive

password sonification, it is suggested that real-time auditory feedback helps users recognise and improve passwords by making password creation more engaging[6]. This project not only investigates the technical challenges associated with creating a secure musical authentication system but also examines practical issues of usability and user experience[1]. Furthermore, this project dives into the ethical implications of integrating AI into security systems[7]. Given the rapid growth of AI, understanding its potential benefits and risks is important in the world of security. By evaluating musical and voice-based passwords, this research aims to discover whether they can offer a more secure, accessible and user-friendly alternative to traditional authentication methods. As concern over AI-driven password-cracking attacks continues to grow, this study investigates how these emerging authentication methods perform against contemporary security challenges[4].

1.2 Project Objectives

The project's aim is to investigate the potential of music and voice as a basis for user authentication. These objectives are within the context of a constantly changing threat landscape. The research was done through user testing, to evaluate these results. The specific objectives of this project are:

- **To Develop a Melody-Based Security System**: This involves designing and implementing a system that can generate secure passwords using vocal input such as singing, humming or playing a musical instrument. This will include:
  - Researching how melody features such as pitch, bass, rhythm and tone can be converted into secure passwords.
  - Develop a system that can extract vocal features like Mel-Frequency Cepstral Coefficients (MFCC), spectral centroid and tempo and use those features as part of the generated password creation process.
  - Training and/or using an AI model to generate passwords based on the extracted vocal feature and a user defined passphrase.
- **To Develop an Audio-Based Security System**: This objective, undertaken by co-researcher Cormac Geraghty, focuses on developing a system of generating secure passwords from audio files.
- **To Create a User Interface for Both Password Systems**: To ensure usability, a user-friendly interface will be developed that integrates both the melody/vocal based password system and the audio-based system. This interface will allow the user to create a password and to login. There will also be a testing feature where the AI Large Language Models (LLMs) will attempt to crack the passwords generated.

- **To Conduct Security Testing and AI Model Evaluation**: This will focus on testing the resilience of the passwords created through AI. This project focuses on the melody-based system. The methodology will be as follows:
  - Training AI models to attempt to crack the passwords that were generated using the system.
  - Comparing the effectiveness of the AI models against passwords generated by the system.
  - Evaluating the ability of the passwords generated to resist brute-force attacks.
  - Using password cracking software like Hashcat to compare the resilience of the passwords generated.
- **To Research and Discuss the Ethical Implications of Using AI in Security**: This involves a detailed analysis of the ethical implications of using AI for both generating passwords and cracking them. Key questions will be addressed, such as:
  - What are the potential privacy risks involved in using voice data for password generation?
  - How can one prevent the misuse of AI in bypassing voice-based authentication systems?
  - What are the potential benefits of AI in enhancing password security?
- **To Review Academic Literature**: This objective is to ensure the work in this project is situated within the broader context of current research. This will involve:
  - Conducting a thorough review of relevant research papers and journals on security systems and AI vulnerabilities.
  - Comparing the findings from this project with existing research to establish the validity and context of this project's conclusions.

1.3 Dissertation Structure

This dissertation is organised into the following sections:

1. **Introduction:**(This section) This chapter provides the context and objectives of this project. It introduces the issues of traditional password systems and the motivation behind exploring alternative methods such as musical-based passwords. It also details the specific objectives of the project.

2. **Methodology:** This chapter outlines the methodological approach adopted for this project. It includes both the software development methodology and research methodology. It details the transition from a virtual machine-based approach to a local Python-based setup due to technical challenges. It also describes the Agile development process employed throughout the project.

3. **Technology Review:** This chapter provides a comprehensive analysis of the current landscape of password authentication. It also looks at alternative approaches including biometrics, audio signal processing techniques and AI applications in password security. It examines password cracking methodologies and ethical considerations in security research, establishing the theoretical foundation for the project.

4. **System Design:** This chapter details the system architecture and implementation of the voice-based password generation and security testing framework. It describes modular components of the system. This includes voice processing, password generation using Claude AI, security testing methodologies and the user interface. The chapter also addresses data storage considerations and security measures implemented in the system.

5. **System Evaluation:** This chapter presents the results of testing the voice-based password system against various cracking methodologies. It evaluates the effectiveness of voice features extraction, assesses password security through brute-force. It also evaluates AI-based cracking attempts and theoretical security metrics. It provides a comprehensive analysis of the system's performance against the project objectives outlined in chapter 1.

6. **Conclusion:** This chapter summarises the key achievements and findings of the project. It discusses limitations and challenges encountered and suggests directions for future work. It reflects on the significance of this research in the field of authentication security.

This dissertation also includes appendices that provide detailed code listings, test documentation and additional technical information to support the main text.

1.4 Project Resources and Individual Contribution

This dissertation describes the development of the voice-based password authentication system, which was created as part of a larger collaborative project. For assessment purposes, the following clarification regarding project resources is essential:

1.4.1 Repository Structure and Component Separation

The complete codebase is available in the following GitHub repository:

https://github.com/JamesDoonan1/sound-to-security

IMPORTANT NOTE FOR EXAMINERS: The voice-based password authentication system that forms the basis for this dissertation is contained exclusively in the following branch:

https://github.com/JamesDoonan1/sound-to-security/tree/feature/vocal-passwords-new

This branch contains the complete voice-based password system implementation, including all testing and documentation. The findings and conclusions presented in this dissertation are based solely on work conducted within this specific branch.

1.4.2 Collaborative Context

As illustrated in Figure 1.1, the repository contains multiple branches that reflect the collaborative nature of the project:

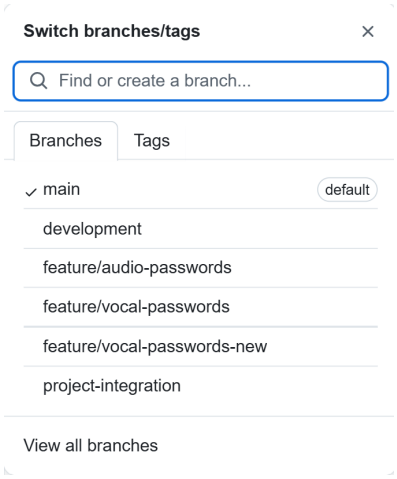

Figure 1.1: GitHub Branches showing separation of the voice-based authentication system (feature/vocal-passwords-new) and the integrated project

The `main` branch represents the integrated version that combines:

- The voice-based password authentication system (the focus of this dissertation)
- Co-researcher Cormac Geraghty's work on audio-based encryption (documented separately)

Chapter 2

Methodology

2.1 Overview of Approach

This chapter outlines the methodologies used in the development of this project. It includes both the software development and research methodologies. The primary goal of this project is to investigate the feasibility of music and voice-based password authentication as an alternative to traditional passwords. This involved the development of a prototype and also empirical testing of its security against AI-driven attacks. To achieve this, two interrelated methodologies were adopted.

- **Software Development Methodology**: An Agile, incremental, and iterative approach was followed. This ensured flexibility and allowed for rapid improvements throughout the development cycle.

- **Research Methodology**: The effectiveness of the generated passwords was evaluated using AI models (GPT-4 and Claude), brute-force techniques, and established security benchmarks (such as Hashcat).

The following sections detail how these methodologies were applied in the development, testing and evaluation of this project.

2.2 Software Development Methodology

2.2.1 Initial Virtual Machine (VM) Setup and Justification

The initial development of this project was planned to take place within a Virtual Machine running Ubuntu. This environment was chosen to facilitate the integration of JUCE (Projucer) for GUI-based password generation[8], AI models (GPT-4, Claude, LLaMA) for security testing, and brute-force tools such as Hashcat for password cracking. This approach was carefully selected to create an

isolated development environment that could facilitate greater flexibility in testing AI-driven password generation and security vulnerabilities. This development went on for over a month and can be seen in the following GitHub branch:

GitHub Branch with original VM development

As development progressed, significant technical challenges arose. This ultimately led to the decision to transition to a local development setup using Python within Visual Studio Code on Windows.

2.2.2 Challenges in the Virtual Machine-Based Approach

Although there were many theoretical advantages of using a virtualised environment, the practical implementation of this setup proved problematic from the beginning. Several issues were encountered that impacted the development time greatly.

1. **Virtual Machine and Ubuntu Installation Issues** The installation process of Ubuntu within the VM was full of complications. This included installation freezes and compatibility conflicts across different versions. Although Ubuntu 18.04.6 LTS was eventually installed, the process was time consuming and required extensive troubleshooting.

2. **Dependency Management and Library Installations** An essential Python package called Torch failed to install due to persistent disk space limitations within the VM. Expanding the disk space and partitioning did not resolve the issue. This required additional manual interventions and introduced further package conflicts.

3. **File Management and Data Transfer Issues** The process of transferring files between the Windows host system and the Ubuntu VM proved problematic. VirtualBox shared folders required manual configuration and at times, file modifications were not accurately reflected between environments.

4. **JUCE and Projucer Compatibility Issues** The Projucer software presented multiple errors, including:

    - Warnings regarding outdated versions.
    - Dependency failures due to missing system libraries such as GLIBC 2.38 and libstdc++.
    - Attempts to manually update these libraries introduced further compatibility issues with other installed packages.

5. **VS Code and Graphical User Interface (GUI) Problems** Attempts to install Visual Studio Code via the standard package manager failed due to dependency conflicts. This necessitated the use of Snap for installation. Additionally, efforts to configure a GUI environment (XFCE4) encountered session errors. This required manual interventions to restart the display manager.

6. **Resource Constraints and System Stability** The VM consistently encountered RAM and storage limitations. This restricted the ability to install large AI models and execute concurrent testing scenarios. Even after increasing allocated disk space and optimising resource management, challenges persisted in maintaining system stability.

Due to these persistent technical setbacks, the VM-based approach was ultimately abandoned in favour of a more streamlined local development setup.

2.3 Transition to a local Python-Based Setup

To mitigate the inefficiencies of the virtual environment, the project was restructured to run natively on a Windows system. This approach allowed the utilisation of Python with VS Code. This transition resulted in several key advantages:

- **Elimination of Virtual Machine overhead**: This significantly improved performance by eliminating constraints caused by limited disk space, RAM, and VM overhead.
- **Simplified Dependency Management**: Python packages could be installed directly, avoiding complexities introduced by Ubuntu's package manager.
- **More Efficient File Handling**: Local file access eliminated the need for manual shared folder configurations.
- **Faster Development Workflow**: Removing the VM significantly reduced system latency, allowing quicker development.

2.4 Agile Development Process

The project initially adopted the Scrum Methodology[9]. It utilised Jira as the primary project management tool[10]. Both co-authors established a structured approach with defined sprints and a comprehensive product backlog. Tasks were

broken down into user stories and assigned story points to estimate complexity and effort.

Figure 2.1 shows the original sprint planning in Jira before adaptation was necessary.

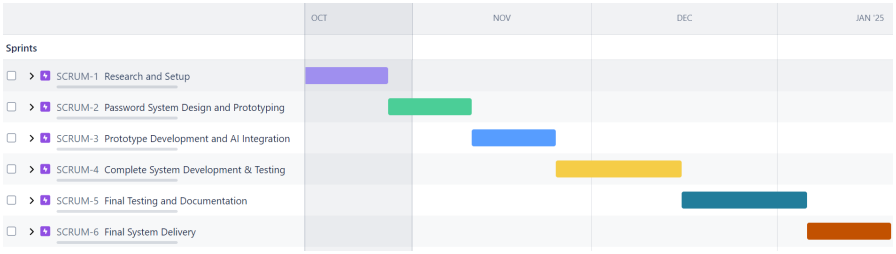

Figure 2.1: Original Jira sprint timeline showing the initial Scrum-based planning approach.

However, as technical challenges emerged with the virtual machine setup described in Section 2.2.2, adherence to the initially planned sprint cadence became increasingly difficult. The combination of these technical setbacks necessitated a more flexible approach to project management.

2.4.1 Adaptation of Methodology

In response to these challenges, the team transitioned from Jira to GitHub Issues[11]. This provided a more integrated and streamlined environment for the development workflow. This adaptation represented a hybrid Agile approach that retained core Agile principles while allowing for greater flexibility:

- **Milestones as Sprints**: GitHub milestones functioned as sprint equivalents, providing time-boxed periods for completing specific objectives.
- **Issue Tracking**: GitHub Issues facilitated tracking of individual tasks, bugs, and feature implementations. This approach allowed for:
  - Creation of issues specific to individual components
  - Assignment of collaborative tasks requiring input from multiple team members
  - Direct linking of issues to code commits, providing clear traceability
  - Visual progress tracking through GitHub's project boards
- **Continuous Integration**: By linking commits directly to issues, the project maintained continuous integration practices that allowed for marking tasks as complete upon successful implementation.

2.4.2 Communication and Collaboration

The Agile principle of regular communication was maintained throughout the project:

- **Daily Check-ins**: Regular communication channels were established to discuss progress, obstacles, and next steps. These check-ins mirrored daily stand-ups in traditional Scrum but were conducted virtually.
- **Monthly In-Person Collaboration Sessions**: Extended in-person collaboration sessions, held at least monthly, combined sprint planning, backlog refinement, and pair programming activities. These intensive sessions were particularly valuable for resolving complex integration issues between system components.
- **Bi-weekly Supervisor Meetings**: Regular meetings with the project supervisor every two weeks served as sprint reviews. During these meetings, progress was demonstrated, feedback was received, and priorities were adjusted. The supervisor effectively acted as the Product Owner within this adapted Agile framework, providing guidance on project scope and direction.

2.4.3 Sprint Planning and Timeline

Despite the adaptation from a strict Scrum framework to a more flexible GitHub-based approach, the project maintained a structured timeline and well-defined development phases. Figure 2.2 illustrates the adapted project timeline, showing progress and completion percentages for each phase. All of these phases are now complete.

The sprint schedule was organised into six main phases:

1. **Research and Setup (October)**: Initial literature review, requirement gathering, and development environment configuration.
2. **Password System Design and Prototyping (October–November)**: System architecture design and creation of initial prototypes for the voice-based authentication components.
3. **Prototype Development and AI Integration (November)**: Implementation of core functionality and integration of AI models for password generation.
4. **Complete System Development and Testing (December–January)**: Full system implementation including the security testing framework and user interface.

5. **Final Testing and Documentation (January)**: Comprehensive security evaluation and completion of system documentation.

6. **Final System Delivery (January–February)**: Final refinements and preparation of the complete system for submission.

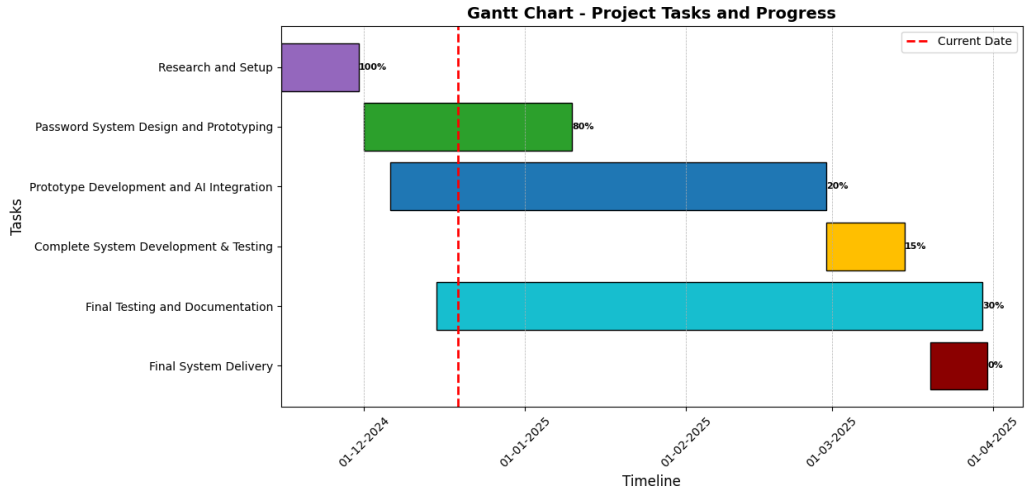

Figure 2.2: Adapted Gantt chart showing project tasks, timeline, and actual completion percentages from before January 2025.

2.4.4 Iterative Development

Throughout the project lifecycle, the core Agile principles of iterative development were maintained.

- Each feature was developed incrementally, starting with a minimal viable implementation and subsequently refined based on testing and feedback.
- The password generation system progressed through several iterations, with each version incorporating improvements to security, usability, or performance.
- Testing was integrated throughout the development process rather than being treated as a separate phase, allowing for early identification and resolution of issues.

This adapted Agile approach enabled effective navigation of the technical challenges described in Section 2.2.2 while maintaining productive development momentum. The flexibility offered by GitHub Issues proved more suitable for an

academic project context. Whereas traditional corporate Scrum implementations might have been too rigid given the evolving nature of the research and development requirements.

Chapter 3

Technology Review

This chapter provides an important analysis of key technologies, methodologies and frameworks that support this project. It explores the current state of research in password security, voice-based authentication, audio signal processing and AI-based security training. It also highlights the theoretical foundation upon which this project builds.

3.1 Password Authentication: Current Landscape and Challenges

Traditional password authentication remains the most prevalent form of digital security despite its well-documented limitations. Grassi et al.(2017) notes that conventional alphanumeric passwords present an inherent trade-off between security and usability[12]. While it is true that complex passwords offer greater theoretical security, they often impose a burden on users that can lead to insecure practices such as password reuse and simplification.

Kumar (2012) observes that the increasing number of digital accounts managed by individuals has exacerbated these security vulnerabilities. Users frequently resort to easily memorable and consequently crackable passwords. According to his research, passwords associated with personal information such as birthdays, pet and family names are particularly vulnerable to both social engineering and algorithmic cracking methods[3].

This password vulnerability has led to numerous huge security breaches across various sectors. Ur et al. (2015) conducted a large-scale empirical analysis of password strength and found that even passwords that users perceived as secure were often susceptible to cracking algorithms[1]. Their research showed that dictionary-based attacks, augmented with common character substitution (e.g., replacing 'a' with '@'), could compromise a significant percentage of user-created passwords.

3.2 Alternative Authentication Approaches: The Rise of Biometrics

The limitations of traditional passwords have prompted research into alternative authentication methods. These include biometric approaches which are gaining significant traction. While fingerprints and facial recognition have seen widespread adoption, voice-based biometrics represent a promising yet unexplored domain [1, 13].

Voice biometrics offer several advantages over conventional authentication methods. Voice-based systems can operate in environments where visual interfaces are impractical or unavailable. This makes them suitable for IoT devices and accessibility focused applications[4]. The research shows that on cloud-based authentication using sound demonstrates that acoustic signatures can provide robust security layers when properly implemented.

Gibson et al (2009) introduced MusiPass, an innovative approach that leverages personal musical preferences as an authentication mechanism[2]. Their system demonstrated that users could more reliably recall music patterns over complex alphanumeric strings. This suggests that music-based authentication might offer an optimal balance of security and memorability. This work was later expanded in Gibson et al, (2015), which explored various implementations of music-based authentication and their effectiveness against different attack vectors[1].

3.3 Audio Signal Processing and Feature Extraction

Sophisticated signal processing technology is at the core of voice and audio based authentication systems. Modern audio analysis techniques extract distinctive features from vocal inputs that can serve as the foundation for secure authentication mechanisms[14].

3.3.1 Mel-Frequency Cepstral Coefficients (MFCCs)

MFCCs represent one of the most powerful techniques for voice characterisation and have become a standard feature in speech recognition systems. Kim et al. (2020) demonstrated that MFCCs can capture the unique spectral characteristics of an individual's voice[15]. This makes them ideal for biometric applications. The MFCC extraction process involves:

1. Applying a pre-emphasis filter to enhance higher frequencies
2. Framing the signal into short segments
3. Computing the power spectrum via Fast Fourier Transform (FFT)
4. Applying a Mel-filterbank to mimic human auditory perception

5. Taking the logarithm of the filterbank energies
6. Computing the Discrete Cosine Transform (DCT)

This multi-step process yields a set of coefficients that represent the short-term power spectrum of the sound. It effectively captures the unique timbral and phonetic characteristics of an individual's voice[7].

3.3.2 Spectral Centroid and Temporal Features

Beyond MFCCs, spectral centroid measurements provide valuable information about the "brightness" of a sound. This effectively capturing the weighted mean frequency of the spectrum[16]. This feature proves particularly useful in distinguishing between different vocal qualities and instrumental sounds.

Temporal features such as tempo and rhythm patterns also play a huge role. Lutz et al. (2020) explored the use of sonification in password strength feedback, demonstrating that temporal audio characteristics can convey security information intuitively to users[6]. Their work suggests that rhythm and tempo can serve not only as authentication factors but also mechanisms for improving user awareness of security practices.

3.4 AI and Machine Learning in Password Security

Artificial Intelligence, particularly machine learning techniques have transformed both offensive and defensive approaches to password security. It presents significant challenges and opportunities for developing more secure authentication systems.

3.4.1 AI for Password Generation

Large Language Models (LLMs) such as Claude and GPT-4 have shown remarkable capabilities in generating complex password patterns that adhere to security best practices[17, 18]. These models can create passwords with high entropy by leveraging their understanding of linguistic patterns. This results in strings that are very secure and tough to crack and/or guess. However, Hitaj et al. (2019) noted in their work on PassGAN that the same generative capabilities used to create secure passwords can also be leveraged to predict and crack user-generated passwords[19]. This paradox highlights the importance of evaluating AI-generated passwords against AI-driven cracking attempts.

3.4.2 AI-Driven Password Cracking

Modern password cracking has evolved beyond simple brute force and dictionary attacks to incorporate sophisticated machine learning approaches. Melicher et al. (2016) demonstrated that neural networks can learn the patterns and structures of human-created passwords. This enables a more efficient and targeted cracking attempts[20].

3.5 Password Cracking Methodologies

Understanding the methods employed in password cracking is extremely important for evaluating the security of any authentication system. This project incorporates several established cracking methodologies to assess the resilience of voice-based passwords.

3.5.1 Brute Force Attacks

Brute force approaches systematically attempt every possible combination of characters until the correct password is found. While in theory it is quite simple, the computational demands of brute force attacks grows exponentially with password length and character set diversity. For a password of length n using a character set of size m, the worst-case search space is m^n combinations. Figure 3.1 illustrates the exponential relationship in practical terms. It shows the emphasise both length and character diversity. The figure clearly demonstrates why security experts recommend passwords to be around 12 characters long, incorporating multiple numbers and special characters.

Figure 3.1: Estimated time to crack passwords of varying lengths and character sets through brute force methods. This exponential increase, following the m^n growth rate, highlights the importance of password length and character diversity. *Source: Hive Systems, 2023 — https://www.hivesystems.com/blog/are-your-passwords-in-the-green/*

Traditional brute force implementations, such as those found in tools like Hashcat (used in this project), employ various optimisations including parallelisation

and GPU accelerations[21]. These optimisations are used to significantly increase cracking speeds[22].

3.5.2 Dictionary Attacks

Dictionary attacks leverage pre-compiled word lists that are often augmented with common modifications to target user passwords. These attacks prove highly effective against passwords derived from natural language especially when combined with rule-based transformations. For example capitalisation, number substitution and special character addition. The effectiveness of dictionary attacks depends largely on the comprehensiveness of the wordlist used. This project employs the widely-used RockYou dataset. This dataset contains over 14 million unique passwords leaked from various data breaches[23].

3.5.3 Hybrid Approaches

Modern password cracking often combines multiple methodologies into hybrid approaches. For instance, an attack might use a dictionary as a base but then apply brute force techniques to add character variations.

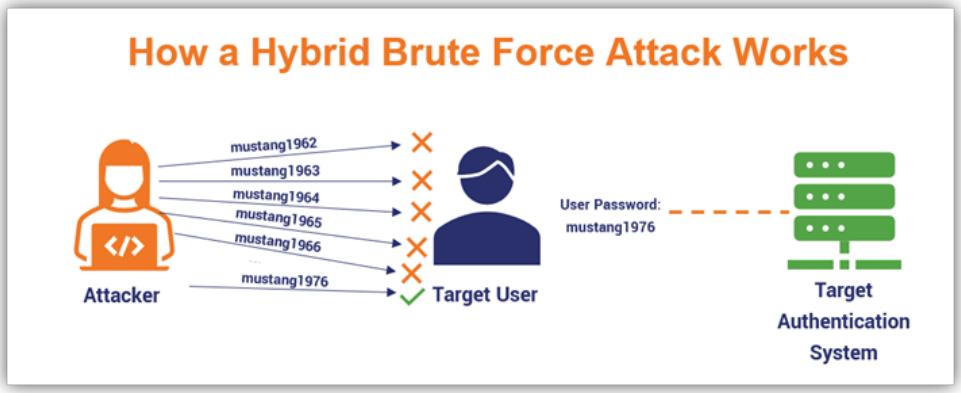

Figure 3.2: Illustration of a hybrid brute force attack. The attacker uses a base word ("mustang") and systematically tries variations by adding different years until finding the correct password. This approach significantly reduces the search space compared to pure brute force methods. *Source: The SSL Store, 2023 — https://www.thesslstore.com/blog/brute-force-attack-definition-how-brute-force-works/*

In the example shown in Figure 3.2, the attacker identifies a potential base word ("mustang") and then methodically applies variations by adding different years. This approach is particularly effective against passwords that follow predictable patterns, such as appending numbers to dictionary words. The hybrid method

dramatically reduces the search space from billions of possible combinations to just a few hundred or thousand high-probability candidates.

3.5.4 AI-Based Password Cracking Tools

Recent advancements in LLMs have introduced a new paradigm in password cracking. Unlike traditional methods that rely on predetermined rules or wordlists. AI-driven approaches can identify patterns and predict likely passwords based on their training data, prompt, and contextual understanding. In theory, large language models like GPT-4 and Claude could analyse behavioural and linguistic patterns to predict password choices reflective of human habits. When provided with sufficient context about a user (such as passphrases and voice characteristics in the case of this project), these models could potentially produce accurate guesses by inferring relationships between the input and resulting password. What makes AI-based cracking particularly significant is the ability to operate at a higher level of abstraction than traditional methods. While tools like Hashcat excel at systematic exploration of password spaces based on specific rules. LLMs can make intuitive leaps that mimic human thinking patterns. This potentially identifies connections that rule-based systems would miss. Notably, different LLMs operate under varying ethical guidelines that impact their behaviour in security testing scenarios[17, 18]. Some models may refuse to participate in password cracking attempts due to ethical constraints. The variation in ethical implementation represents an important consideration when evaluating the practical applications of these tools in security research.

3.6 Ethical Considerations in Security Research

Ethical considerations are central to password security research, especially in relation to the responsible design and testing of security systems. Spring and Huth (2019) emphasise that security researchers must consider not only the immediate applications of their work but also potential misuse scenarios[24].

Egelman et al. (2021) propose a framework for ethical security research that illustrates transparency, informed consent and harm minimisation[25]. These principles have guided the research path for this project, particularly in the evaluation of AI systems that could potentially be deployed for both defensive and offensive purposes.

The use of voice data for authentication raises ethical concerns, as biometric recordings can pose privacy risks if compromised. This project acknowledges these issues and is intended solely for research purposes.

Chapter 4

System Design

This chapter provides a detailed explanation of the system architecture and implementation of the voice-based password generation and security testing framework. The system is designed to capture vocal input, extract distinctive features, generate secure passwords using AI and finally evaluate their security against various cracking methodologies. The architecture reflects the project's primary research objective: to evaluate whether AI-generated passwords based on vocal features offer superior security compared to traditional password approaches.

4.1 System Architecture Overview

The system offers a modular architecture organised into several interconnected components as illustrated in Figure 4.1. The architecture is structured to facilitate the flow of data from voice input to password generation and subsequent security testing. As this project was done by two individuals, the green path relates to this project in particular. The orange path was completed by co-author Cormac Geragthy and the blue colour illustrates integration between both projects available at the following GitHub branch: https://github.com/JamesDoonan1/sound-to-security.

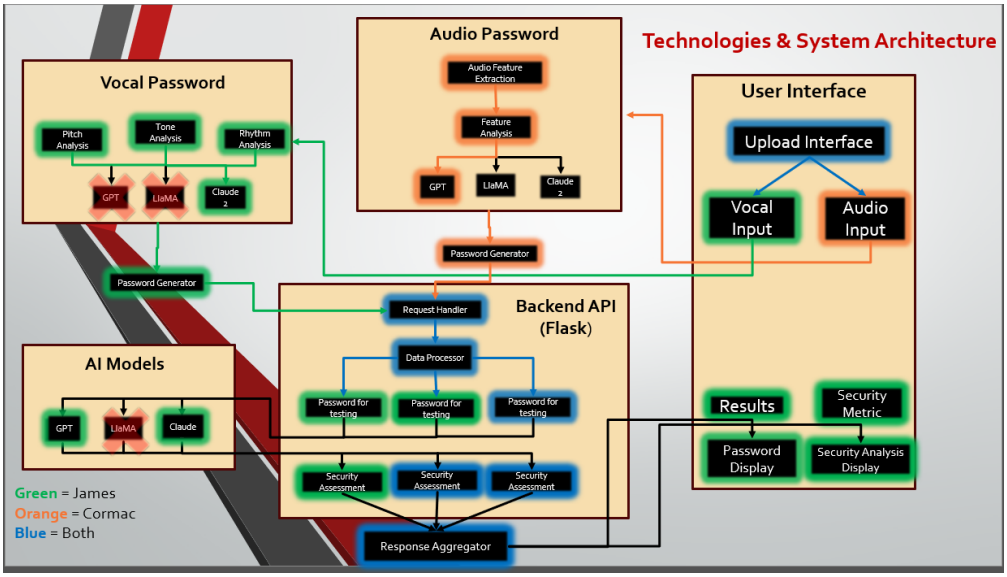

Figure 4.1: System Architecture highlighting the separation and integration of contributions between both researchers.

The system consists of four main functional blocks:

1. **Voice Processing:** Handles audio capture and feature extraction.
2. **Password Generation:** Manages AI-based password creation using Claude.
3. **Security Testing:** Implements multiple password cracking methodologies.
4. **User Interface:** Provides graphical interface for user interaction.

The components are connected through a Flask API backend that facilitates communication between the frontend and the core services[26]. This modular design allows for independent testing and development of each component while maintaining a cohesive system.

4.1.1 Project Directory Structure

```
sound-to-security/
|- backend/
|  |- app.py
|  |- routes/
|  |  |- passwords_routes.py
|  |  \- voice_routes.py
|  |- services/
|  |  |- ai_password_cracker.py
|  |  |- gpt_password_tester.py
|  |  |- hashcat_cracker.py
|  |  |- password_cracker.py
|  |  \- passwords_service.py
|  |- data/
|  \- logs/
|- frontend/
|  \- gui.py
|- models/
|  |- claude_password_generator.py
|  \- voice_recognition.py
|- vocal_passwords/
|  |- feature_extraction.py
|  |- voice_auth.py
|  \- voice_processing.py
\- main.py
```

This organisation separates concerns and follows a logical layering with:

- **Backend services** handling core functionality.
- **API routes** exposing these services.
- **Models** implementing specific AI integrations.
- **Frontend** providing user interaction.

4.1.2 Data Flow

The data flow through the system follows a linear progression:

1. Voice input is captured through the audio recording component.
2. Audio features (MFCCs, spectral centroid, tempo) are extracted.

3. Speech recognition converts spoken words to a text passphrase.
4. The extracted features and passphrase are sent to Claude AI.
5. Claude generates a secure password tailored to the provided inputs.
6. The generated password undergoes multiple security tests.
7. Results are displayed to the user and stored for analysis.

This flow enables the transformation of vocal characteristics into secure passwords while facilitating comparative security testing.

4.2 Voice Input and Feature Extraction

The voice input and feature extraction module forms the foundation of the system. It provides the unique vocal characteristics that drive password generation

4.2.1 Audio Capture System

The audio capture system is implemented using **sounddevice** library to record short audio samples and **soundfile** to save these recordings[27, 28]. The primary implementation is in the following file **voice_processing.py**:

```
ALGORITHM RecordAudio
INPUT: duration (seconds), sample_rate (Hz)
OUTPUT: audio data, sample rate

1. Initialise audio recording configuration (duration,
   sample_rate, channels=1)
2. Start recording audio from microphone
3. Wait for recording to complete
4. Save recorded audio to file "vocal_input.wav"
5. IF recording successful THEN
   a. Return audio data and sample rate
   ELSE
   b. Log error message
   c. Return null values
```

Listing 4.1: Python function for recording and saving audio

See Appendix A, Section 1.1 for full implementation.

The recording duration is set to 5 seconds. This allows sufficient time for the user to speak a passphrase while keeping the process efficient. A sampling rate of 22050 Hz is used as it provides good quality voice capture while keeping file size manageable. The captured audio is saved as "vocal_input.wav" for further processing and feature extraction.

4.2.2 Feature Extraction Pipeline

The feature extraction pipeline is implemented in feature_extraction.py and uses the **librosa** library to extract three key features from the audio[29]:

1. **Mel-Frequency Cepstral Coefficients (MFCCs):** Capture the timbral characteristics of the voice.
2. **Spectral Centroid:** Represents the "brightness" of the sound.
3. **Tempo:** Captures the speaking rate or rhythm.

```
ALGORITHM ExtractAudioFeatures
INPUT: audio data, sample rate
OUTPUT: array of voice features

1. Extract MFCC features from audio using librosa
   a. Calculate mean of MFCC coefficients
2. Extract spectral centroid using librosa
   a. Calculate mean of spectral centroid
3. Extract tempo using beat tracking algorithm
4. Combine features into normalised array:
   a. MFCC mean
   b. Spectral centroid mean
   c. Tempo
5. Log extracted features for analysis
6. Return feature array
```

Listing 4.2: Audio feature extraction from voice input

See Appendix A, Section 1.2 for full implementation.

The first 13 MFCC coefficients are extracted using Librosa, as they comprehensively capture the timbral characteristics of the voice. However, for efficiency and to focus on the most significant information, only the mean of the first 5 MFCC coefficients is used in the final feature vector. This approach reduces dimensionality while retaining the most acoustically relevant information. The spectral centroid, representing the voice's brightness, is calculated using Librosa's built-in function and averaged.[29].

The rhythm feature extraction is handled by a separate function:

```
ALGORITHM ExtractRhythmFeatures
INPUT: audio data, sample rate
OUTPUT: tempo (beats per minute)

1. Use librosa beat tracking algorithm to detect tempo from
   audio
2. Return the tempo value
```

Listing 4.3: Rhythm feature extraction function

See Appendix A, Section 1.2 for full implementation.

This function uses librosa's beat tracking algorithm to estimate the tempo of the speech. It is expressed in beats per minute (BPM). This captures the speaking rate, which can be a distinctive characteristic of an individual's voice.

The extracted features are reduced to three numerical values:

1. The mean of the first 5 MFCC coefficients.

2. The mean spectral centroid.

3. The estimated tempo.

These features are logged for analysis and returned as a NumPy array for further processing[30].

4.2.3 Passphrase Recognition

Speech recognition is implemented using SpeechRecognition library with Google's Speech-to-Text API[31]. This converts the user's spoken words into a passphrase that serves as an additional input for password generation. The implementation is in voice_auth.py.

```
ALGORITHM RecogniseSpeech
INPUT: path to audio file
OUTPUT: recognised text or default value

1. Initialise speech recogniser
2. Load audio file and record audio content
3. TRY to recognise speech using Google Speech-to-Text API
   a. IF successful THEN return recognised text
   b. IF speech not understood THEN return "UNKNOWN_PHRASE"
   c. IF API error THEN return "ERROR"
```

Listing 4.4: Speech recognition from audio input

See Appendix A, Section 1.3 for full implementation.

The function loads the previously saved audio file. Then processes it using "recognizer" and then attempts to convert the speech to text using Google's API. If successful, it returns the recognised text as the passphrase. If the speech is unintelligible or there is an API error, it returns a default value.

4.2.4 Voice Authentication

For further authentication purposes, the system also implements voiceprint storage and verification using Resemblyzer library[32]:

```
ALGORITHM SaveVoiceprint
INPUT: voice_features (array of extracted audio features)
OUTPUT: None (side effect: saves voiceprint to file)

1. Convert voice_features to NumPy array with float32 data
   type
2. Save array to binary file at VOICEPRINT_FILE path
3. Log success message with file path



ALGORITHM VerifyVoice
INPUT: new_voice_features, similarity_threshold
OUTPUT: Boolean indicating authentication success

1. Load stored voiceprint reference from file
2. IF reference voiceprint not found THEN
   a. Log error message
   b. RETURN false
3. Convert new_voice_features to NumPy array with float32
   data type
4. Calculate similarity using Euclidean distance between
   stored and new features
5. Compute dynamic threshold based on magnitude of stored
   voiceprint (20% variation)
6. Use maximum of fixed threshold and dynamic threshold as
   final threshold
7. RETURN true IF similarity < final_threshold ELSE false
```

Listing 4.5: Voiceprint saving and verification functions

See Appendix A, Section 1.4 for full implementation.

This implementation stores the extracted voice features as a reference voiceprint and later compares new voice inputs against this reference using **Euclidean dis-**

tance. A dynamic threshold is calculated based on the magnitude of the stored voiceprint, allowing for a 20% variation while maintaining security.

4.3 Password Generation System

The password generation system leverages Claude AI to create secure passwords based on the extracted voice features and passphrase.

4.3.1 Claude AI Integration

The system integrates with **Anthropic's Claude AI** through their API[17]. The implementation in claude_password_generator.py handles authentication, request formatting and response processing:

```
ALGORITHM GeneratePasswordWithClaude
INPUT: voice features, passphrase, max_retries
OUTPUT: secure password or error message

1. Construct prompt with:
   a. Voice feature values
   b. Users passphrase
   c. Password requirements (12 chars, mixed case, symbols,
      no dictionary words)
2. FOR retry_count = 1 TO max_retries:
   a. Send request to Claude API with:
      i. Model: "claude-2"
      ii. Temperature: 0.7
      iii. System instruction for password generation
   b. Extract password from response
   c. Validate password meets security requirements
   d. IF valid THEN
      i. Log password
      ii. RETURN password
   e. ELSE
      i. Log invalid attempt
      ii. CONTINUE to next retry
3. IF all retries exhausted THEN
   a. RETURN error message
```

Listing 4.6: Generating a password using Claude AI

See Appendix A, Section 2.1 for full implementation.

The integration uses the model "claude-2" with a moderate temperature setting (0.7, where higher values produce more diverse outputs) to encourage some creativity while maintaining consistency[17]. The function implements retry logic to handle potential API errors or invalid responses. This is done with a maximum

of 5 attempts before returning an error message.

4.3.2 Prompt Engineering

The prompt is carefully designed to guide Claude in generating secure passwords based on the provided inputs. It includes:

1. The extracted voice features as numerical values.
2. The user's passphrase.
3. Specific requirements for the password:
   (a) Exactly 12 characters.
   (b) Mix of uppercase, lowercase, numbers and symbols.
   (c) Avoidance of dictionary words.

The system instructions reinforces that Claude should return only the password without additional explanation. This ensures a clean response that can be directly used. This also simplifies post-processing, reducing the risk of misinterpretation or parsing errors during password retrieval or storage. It allows the generated password to be seamlessly passed into the next phase of the pipeline (e.g., hashing, testing, or storage) without needing extra logic to filter out irrelevant text.

The temperature setting 0.7 provides a balance between deterministic outputs (which could be predictable) and completely random outputs (which could ignore input features). This allows Claude to generate passwords that are influenced by the voice features and passphrase while maintaining high security.

4.4 Security Testing Framework

The security testing framework implements multiple methodologies to evaluate password strength, including brute force attacks, AI-based attempts and integration with professional cracking tools.

4.4.1 Brute Force Implementation

The brute force implementation systematically tries all possible character combinations up to a specific length. It is implemented using Python's itertools library in password_cracker.py

```
ALGORITHM BruteForceWorker
INPUT: target_password, max_length, character_set,
   result_container
OUTPUT: None (modifies result_container by reference)

1. Record start time
2. FOR length FROM 1 TO max_length:
   a. FOR each possible combination of characters of current
      length:
      i. Construct password guess from combination
      ii. IF guess equals target_password THEN
          - Record end time
          - Set result.cracked = TRUE
          - Set result.guess = guess
          - Set result.time_taken = elapsed time
          - RETURN
3. Set result.cracked = FALSE
4. Set result.message = "Password not cracked within limits"

ALGORITHM BruteForceCrack
INPUT: target_password, max_length (default 12), timeout (
   default 30 seconds)
OUTPUT: Dictionary with cracking result

1. Define full character set (letters, digits, punctuation)
2. Initialise result dictionary with default failure state
3. Create new thread running BruteForceWorker with inputs
4. Start thread execution
5. Wait for thread to complete or timeout
6. IF thread is still running after timeout THEN
   a. RETURN dictionary with "Brute force timed out" message
7. ELSE
   a. RETURN result dictionary from worker
```

Listing 4.7: Brute force password cracking with timeout

See Appendix A, Section 3.1 for full implementation.

The implementation uses threading to allow for timeout control. The character set includes ASCII letters, digits and punctuations providing a comprehensive range of characters typically used in passwords.

The function returns the cracking result, including whether the password was cracked, the correct guess (if successful), and the time taken.

4.4.2 AI-Based Testing with GPT-4

The system includes a GPT-4 based testing component that attempts to crack passwords by generating educated guesses based on the voice features and passphrase provided to create the testing generated password. Implemented in gpt_password_tester.py:

```
ALGORITHM TestPasswordWithGPT
INPUT: password to test, passphrase, voice features
OUTPUT: testing result with cracking status

1. Format voice features into readable string
2. Construct prompt with:
   a. Passphrase
   b. Voice feature details
   c. Instructions to generate 5 possible passwords
3. TRY to send request to GPT-4 API:
   a. Extract generated password guesses
   b. Ensure exactly 5 guesses are returned
   c. Check if actual password matches any guess
   d. Calculate elapsed time
   e. RETURN result with:
      i. Success/failure status
      ii. Time taken
      iii. Password guesses
      iv. Result message
4. CATCH any exceptions:
   a. Log error
   b. RETURN failure result with error details
```

Listing 4.8: Testing password resilience using GPT

See Appendix A, Section 3.2 for full implementation. This implementation provides GPT-4 with the input data that was originally used to generate the password: the passphrase and voice characteristics. GPT-4 is asked to generate five potential passwords that match the required format based on these inputs. The system then checks if any of these generated guesses match the actual AI-generated password. This approach tests whether GPT-4 can reverse-engineer the password generation logic used by Claude.

4.4.3 AI Password Cracking with Claude

The system also evaluates password security using Claude AI as a cracking tool. This implementation in ai_password_cracker.py attempts to use Claude's language capabilities to generate possible password variations:

```
ALGORITHM AICrackPassword
INPUT: target_password
OUTPUT: cracking result with potential variations or ethical
   response

1. Construct prompt asking Claude to:
   a. Analyze the provided password
   b. Generate 10 potential similar passwords an attacker
      might try
   c. Acknowledge the possibility of an ethical refusal
2. TRY to send request to Claude API:
   a. Set appropriate system instructions regarding password
      security
   b. Configure temperature and token limits
   c. Format output for analysis
3. Process the response:
   a. IF ethical refusal detected THEN
      i. Record explanation
      ii. RETURN security assessment with ethical notes
   b. ELSE extract password guesses
      i. RETURN success/failure status and guess attempts
4. CATCH API errors with appropriate fallbacks
```

Listing 4.9: Testing password resilience using Claude AI

See Appendix A, Section 3.2 for full implementation.

This component provides an important assessment of AI-powered password cracking capabilities. Unlike the GPT-4 implementation which receives voice features and passphrase, Claude receives only the target password and attempts to derive variations directly. This tests different attack vectors: GPT-4 evaluates whether the input features can be reverse-engineered into the password, while Claude tests whether pattern recognition can generate similar variations.

Testing revealed that Claude often refuses to perform this task due to ethical constraints, highlighting an important security consideration in AI-based testing. These ethical boundaries are analysed in detail in Chapter 5.

4.4.4 Hashcat Integration

The system integrates with Hashcat, a professional password recovery tool. The implementation in hashcat_cracker.py provides a wrapper around the Hashcat command-line tool:

```
ALGORITHM CrackPasswordWithHashcat
INPUT: hash_type, attack_mode, password_hash
OUTPUT: cracking result (success/failure, recovered password)

1. Prepare hash file for testing:
   a. IF specific hash provided THEN save to temporary file
   b. ELSE use existing hash file
2. Verify Hashcat executable exists
3. Prepare attack-specific resources:
   a. IF dictionary attack THEN ensure wordlist exists
   b. ELSE prepare mask parameters
4. Construct Hashcat command with appropriate parameters:
   a. Hash type
   b. Attack mode
   c. Input hash file
   d. Attack-specific parameters
5. TRY to execute Hashcat process with timeout:
   a. Run cracking command
   b. Check results with --show parameter
   c. IF password recovered THEN
      i. Extract password from output
      ii. RETURN success with recovered password
   d. ELSE
      i. RETURN failure with message
6. Handle exceptions (timeout, errors)
7. Clean up temporary files
```

Listing 4.10: Password cracking using Hashcat

See Appendix A, Section 3.3 for full implementation.

This implementation provides a flexible interface to Hashcat, allowing for different hash types (MD5, SHA-256, etc.) and attack modes (brute force, dictionary). It executes Hashcat as a subprocess with a timeout to prevent indefinite running time. The function returns whether the password was cracked and if successful, the cracked password.

The integration dynamically locates the Hashcat executable, creates necessary temporary files and cleans up after execution.

4.5 User Interface Design

The user interface provides a graphical front-end for interacting with the system. It allows users to generate password and test their security.

4.5.1 GUI Implementation

The GUI is implemented using tkinter, a standard Python library for creating graphical user interfaces[33]. The main implementation is in the gui.py:

```
ALGORITHM SetupGUI
OUTPUT: configured GUI application

1. Create main application window with title "Secure AI
   Password Generator"
2. Configure window size and background color
3. Create custom button styles with specified font and
   padding
4. Add header label with application title
5. Add "Generate Password" button that calls on_generate
   function
6. Add "Compare AI Results" button (initially disabled)
7. Add "Login" button for authentication testing
8. Add result label with initial instructions
9. Start main event loop
```

Listing 4.11: Ttinter GUI setup for the password generator

See Appendix A, Section 4.1 for full implementation.

The GUI uses a dark theme with blue accents for a modern appearance. It provides four main buttons as shown in Figure 4.2:

1. "Generate Password": Initiates the password generation process.
2. "Run Security Tests": Hidden until password is generated. Begins the password security testing process.
3. "Compare AI Results": Hidden until password is tested. Opens a window to compare the security.
4. "Login": Tests the voice-based authentication system.

4.5.2 User Workflow

The user workflow is implemented through event handlers that respond to button clicks. The main workflow for password generation is handled by the on_generate function:

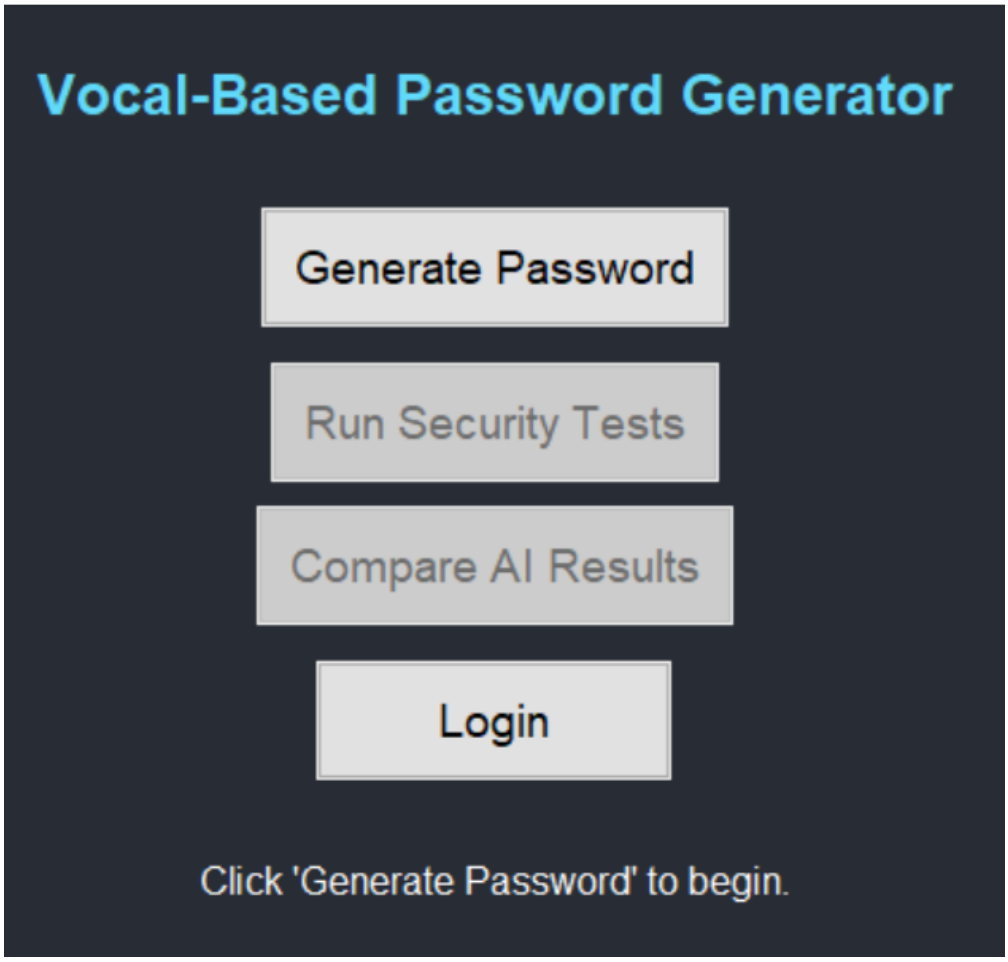

Figure 4.2: Vocal-based graphical interface

```
ALGORITHM GeneratePasswordFromVoice
INPUT: None
OUTPUT: Generated password or error message

1. Begin audio capture process
2. IF audio captured successfully THEN
   a. Extract voice features (MFCC, spectral centroid, tempo)
   b. Perform speech recognition on audio file
   c. IF passphrase not detected THEN
      i. Display error message
      ii. EXIT function
   d. Store recognised passphrase in test results
   e. Save voiceprint and passphrase for authentication
```

```
   f. Send API request to password generation service
   g. IF API request successful (status 200) THEN
      i. Extract AI-generated password
      ii. Extract traditional passwords for comparison
      iii. Store passwords in test results
      iv. Hash and save AI password
      v. Verify hash file creation
      vi. Update UI with generated password
      vii. Enable security testing button
   h. ELSE
      i. Log error
      ii. Update UI with error message
3. ELSE
   a. Log audio capture failure
   b. Update UI with error message
```

Listing 4.12: AI password generation handler

See Appendix A, Section 4.2 for full implementation.

The comparison of different password types is handled by the compare_ai_results function, which creates a separate window to display detailed results as shown in Figure 4.3.

Figure 4.3: Vocal-based graphical interface

This function creates a detailed comparison window that displays the following:

1. The AI generated password.
2. The passphrase used for generation.
3. Claude's response (typically an ethical refusal to crack passwords).
4. Whether GPT-4 successfully cracked the password.
5. The password guesses generated by GPT-4.

4.6 Data Storage and Security

The system implements various data storage mechanisms for voice features, passphrases and passwords with appropriate security measures.

4.6.1 Feature Storage

Voice features are stored as NumPy arrays in binary files. The implementation in passwords_services.py handles both saving and retrieving these features:

```
ALGORITHM ExtractVoiceFeatures
OUTPUT: dictionary of voice features or null

1. Check if voiceprint file exists at expected location
2. IF file exists THEN
   a. Load numerical array from file
   b. Convert to dictionary with keys:
      i. "mfcc": first value
      ii. "spectral_centroid": second value
      iii. "tempo": third value
   c. Return dictionary
3. ELSE
   a. Log error message
   b. Return null

ALGORITHM SaveVoiceprint
INPUT: voice features array
OUTPUT: none (file saved as side effect)

1. Convert features to NumPy array
2. Save array to file at predefined location
3. Log success message
```

Listing 4.13: Voice feature extraction and saving functions

See Appendix A, Section 4.3 for full implementation.

The voice features are stored as a NumPy array with three float32 values corresponding to the MFCC mean, spectral centroid and tempo. The features are retrieved as a dictionary for easier access to individual components:

The passphrases are stored as plain text in a separate file:

```
ALGORITHM ExtractPassphrase
OUTPUT: stored passphrase or null

1. Check if passphrase file exists
2. IF file does not exist THEN
   a. Log error message
   b. Return null
3. Open file and read content
4. IF file is empty THEN
   a. Log error message
   b. Return null
5. Return trimmed passphrase text

ALGORITHM SavePassphrase
INPUT: passphrase text
OUTPUT: none (file saved as side effect)

1. Open passphrase file for writing
2. Write passphrase to file
3. Log success message
```

Listing 4.14: Passphrase extraction and saving functions

See Appendix A, Section 4.3 for full implementation.

While storing passphrases as plain text is not advised for a production system, it is adequate for this research-focused proof-of-concept implementation.

4.6.2 Password Hashing and Storage

Passwords are stored in a hashed format using the SHA-256 algorithm. This implementation in gui.py handles password hashing and storage:

```
ALGORITHM SaveHashedPassword
INPUT: password to hash
OUTPUT: hash value or null

1. IF password is null or empty THEN
   a. Log error message
   b. Return null
2. Generate SHA-256 hash of password
3. TRY to append hash to password file
```

```
   a. Open file in append mode
   b. Write hash followed by newline
   c. Log success message
4. CATCH any file errors
   a. Log error message
5. Store hash in test results for later analysis
6. Return hash value
```

Listing 4.15: Password hashing and secure storage

See Appendix A, Section 4.3 for full implementation.

This function converts the password into a SHA-256 hash, which is a one way transformation. This means that even if someone gains access to the stored hash, they cannot easily reverse it to obtain the original password.

4.6.3 System Security Considerations

While the system implements adequate security measures such as password hashing, it is important to note that this is a research-focused implementation with several security limitations:

1. **Plaintext Passphrases:** Passphrases are stored in plaintext, which would be a security risk in a production system.
2. **Local Storage:** Files are stored locally without encryption, making them vulnerable to direct access.
3. **Limited Authentication:** The authentication system uses simple file-based storage without robust user management.
4. **No Network Security:** API communications are unencrypted and unauthenticated.

These limitations are acceptable for a proof-of-concept research system focused on evaluating password strength rather than providing production-level security. In a production implementation, these issues would need to be addressed with:

1. **Encrypted storage** for all sensitive data.
2. **Secure user management and authentication**.
3. **Transport layer security** for API communications.
4. **More robust password hashing with salt**.
5. **Rate limiting and other protection mechanisms**.

The current implementation provides adequate security for research purposes while focusing on the primary objective of evaluating AI-generated passwords against various cracking methodologies and comparing them to traditional passwords.

4.7 Login System and Two-Factor Authentication

The previous sections have detailed the voice-based password generation system, focusing on how the passwords are tested and created. This section explores the complementary login system that leverages two-factor authentication principles to verify user identity securely.

4.7.1 Two-Factor Authentication Framework

Traditional authentication systems typically rely on a single factor: something the user knows. The system developed in this project implements a more robust two-factor authentication approach that combines:

1. **Something the user knows**: The AI-generated password.
2. **Something the user is**: Voice biometric characteristics combined with a specific passphrase.

This two-factor approach significantly enhances security compared to traditional password-only systems. As Schneier (2005) notes, "Authentication based on multiple independent factors creates a layered defence that is more resilient to attack vectors that might compromise a single factor"[34]. By requiring an attacker to compromise both verification channels simultaneously, the system dramatically increase the difficulty on unauthorised access. The authentication framework follows a sequential verification process, where each factor must be successfully validated: (Voice + Passphrase Verification) → Password Verification

Only when both factors are successfully authenticated is access granted. This sequential approach also provides increased security through defence-in-depth principles, where the failure of one authentication mechanism prevents access to subsequent verification steps.

4.7.2 Login Workflow Implementation

The login system is implemented through the `on_login()` function in the GUI module. This function manages the complete authentication workflow through a series of verification steps:

```
ALGORITHM OnLogin
INPUT: None
OUTPUT: Authentication result (success/failure)

1. RECORD users voice
   audio, sample_rate <- RecordAudio()

2. IF audio is successfully captured THEN
   a. EXTRACT voice features
      features <- ExtractAudioFeatures(audio, sample_rate)
```

```
b. VERIFY voiceprint against stored reference
   IF VerifyVoice(features) = TRUE THEN
      i. RECOGNISE spoken passphrase
         recognized_passphrase <- RecognizeSpeech("
            vocal_input.wav")

      ii. VERIFY passphrase against stored reference
          IF VerifyPassphrase(recognized_passphrase) =
             TRUE THEN
             - PROMPT user for AI-generated password
               user_password <- AskString("Enter the AI-
                  generated password:")
             - HASH entered password
             - COMPARE hash with stored password hash
             - RETURN authentication success
          ELSE
             - LOG passphrase verification failure
             - RETURN authentication failure
          END IF
   ELSE
      - LOG voice verification failure
      - RETURN authentication failure
   END IF
ELSE
   - LOG audio capture failure
   - RETURN authentication failure
END IF
```

Listing 4.16: Voice-based Login Authentication Process

The complete authentication process involves the following steps:

1. **Voice Recording and Feature Extraction**: The system captures the user's voice through the microphone and extracts the same features used in password generation (MFCCs, spectral centroid, and tempo).

2. **Voice Biometric Verification**: Using the `verify_voice()` function, the system compares the extracted voice features against the stored reference voiceprint using Euclidean distance. The verification employs a dynamic threshold that adapts to the magnitude of the stored voiceprint, allowing for natural variations in voice while maintaining security:

```
ALGORITHM VerifyVoice
INPUT: new_voice_features, threshold (default = 50)
OUTPUT: boolean indicating authentication success
1. Load stored voiceprint reference from file
2. IF reference voiceprint not found THEN
   a. Log error message
   b. RETURN false
3. Convert new_voice_features to NumPy array with float32
    data type
4. Calculate similarity using Euclidean distance between
   stored and new features
5. Compute dynamic threshold based on magnitude of stored
    voiceprint (20% variation)
6. Use maximum of fixed threshold and dynamic threshold
   as final threshold
7. RETURN true IF similarity < final_threshold ELSE false
```

Listing 4.17: Voice verification using dynamic threshold

3. **Passphrase Verification**: If voice verification succeeds, the system performs speech recognition on the recorded audio to extract the spoken passphrase. This is then compared to the stored reference:

```
ALGORITHM VerifyPassphrase
INPUT: spoken_passphrase
OUTPUT: boolean indicating passphrase match
1. Retrieve stored passphrase reference
2. IF stored passphrase is not found THEN
   a. Log error message
   b. RETURN false
3. Compare spoken passphrase with stored passphrase (case
   -insensitive)
4. RETURN result of comparison
```

Listing 4.18: Passphrase verification

4. **Password Verification**: Finally, after successful voice and passphrase verification, the user is prompted to enter the AI-generated password. This is hashed and compared against the stored password hash:

```
ALGORITHM VerifyPassword
INPUT: user_entered_password, stored_hash
OUTPUT: boolean indicating password match
1. Hash the user-entered password using SHA-256
```

```
2. Compare the generated hash with the stored hash
3. IF hashes match THEN
   a. Log successful authentication
   b. Update UI with success message
   c. RETURN true
4. ELSE
   a. Log failed authentication attempt
   b. Update UI with failure message
   c. RETURN false
```

Listing 4.19: Password verification

This multi-stage verification process creates a robust authentication system that mitigates vulnerabilities associated with single-factor approaches.

4.7.3 Authentication Decision Process

The authentication system employs specific decision processes for each factor:
First Factor: Voice and Passphrase Verification The voice and passphrase verification represents a combined biometric and knowledge element. This first factor involves two interrelated components:
Voice Biometric Component: Voice verification uses Euclidean distance to measure the similarity between the login voice features and the stored reference. A dynamic threshold calculation adapts to the characteristics of the stored voiceprint:

1. The system calculates the Euclidean norm (magnitude) of the stored voice features.

2. It allows for a 20% variation from this baseline (`dynamic_threshold = norm * 0.2`).

3. A final threshold is determined by taking the maximum of this dynamic threshold and a predefined minimum (50).

4. Authentication succeeds if the distance between the voice features is less than the final threshold.

This approach balances security with usability by accommodating natural variations in vocal characteristics while maintaining sufficient discrimination to prevent impersonation attacks.

Passphrase Component: The passphrase verification employs a case-insensitive exact matching approach. This design choice offers several benefits:

1. Case-insensitivity improves usability by allowing for minor variations in speech recognition.

2. The exact phrase matching requirement maintains security by requiring specific knowledge.

3. The passphrase acts as both a verification component and a way to standardise the vocal input for consistent feature extraction.

Together, these two components create a robust first factor that requires both the correct biometric characteristics and the knowledge of the specific passphrase.

Second Factor: Password Verification: The second factor uses cryptographic hash comparison for the entered password:

1. The user-entered password is hashed using SHA-256.

2. This hash is compared to the most recently stored password hash.

3. The comparison requires an exact match, with no tolerance for variation.

Through these carefully designed decision processes, the system achieves a balance between security and usability while

4.7.4 Integration with Password Generation

The login systems integrates seamlessly with the password generation process described above:

1. During password generation, the system stores:

   - The voice features (voiceprint) as a reference for future authentication.
   - The spoken passphrase for subsequent verification.
   - The hashed AI-generated password.

2. These stored elements form the verification basis for the two-factor authentication process when the user attempts to log in.

The two-factor approach leverages the unique characteristics of the voice-based password generation system, turning elements that were already necessary for password creation (voice input and passphrase) into a cohesive first authentication factor. This elegant reuse of components enhances security without adding significant user burden.

Chapter 5

System Evaluation

This chapter evaluates the effectiveness of the voice-based password generation system developed in this project against the objectives outlined in Chapter 1. The evaluation focuses on the security of AI-generated passwords compared to traditional passwords. In particular their resistance to various cracking methodologies including AI-driven approaches, brute-force attacks and specialised password cracking tools.

5.1 Testing Methodology and Framework

The evaluation methodology was designed to comprehensively assess both security and usability aspects of the voice-based password system. The testing framework consisted of multiple components.

5.1.1 Password Generation Testing

The first phase evaluated the system's ability to generate consistent and secure passwords from voice inputs. Tests included:

1. **Voice Feature Extraction Accuracy:** Testing the system's ability to extract consistent MFCC, spectral centroid, and tempo features from voice recordings.

2. **Password Generation Reliability:** Assessing whether the system generates consistent passwords from the same voice input and passphrase.

3. **Password Complexity Analysis:** Evaluating the complexity of generated passwords based on length, character diversity, and entropy calculations.

5.1.2 Security Testing Methodology

The security testing methodology utilised multiple approaches to assess password strength:

1. **AI-Based Cracking Attempts:**
   - **GPT-4:** Provided with voice features and passphrase to generate password guesses.
   - **Claude:** Tasked with analysing password patterns and generating potential variations.
2. **Traditional Cracking Methods:**
   - Brute force attacks with configurable timeouts.
   - Dictionary attacks using the RockYou dataset (14 million common passwords).
   - Hybrid approaches combining dictionary words with common modifications.
3. **Specialised Cracking Tools:**
   - Hashcat testing with multiple attack vectors:
     - Basic dictionary attacks (–a 0)
     - Rule-based modifications (–a 0 –r best64.rule)
     - Mask attacks for pattern-based cracking (–a 3)
     - Combination attacks (–a 1)
4. **Security Metrics Calculation:**
   - Entropy analysis using Shannon's entropy formula.
   - Theoretical cracking time estimates based on computational complexity.
   - Character set diversity and pattern analysis.

5.1.3 Python-Based Cracking Tool

To support a detailed security evaluation, a custom Python-based password cracking tool was developed and integrated into the testing framework. This tool implements dictionary, pattern-based, and brute-force attack simulations against SHA-256 password hashes[35].

Designed for controlled benchmarking, the tool enables consistent execution across all test scenarios regardless of hardware constraints or operating system. It also captures key metrics including:

- Entropy values calculated using Shannon's formula[36].
- Character set diversity and password composition analysis.
- Estimated brute-force cracking times based on computational complexity.

Known weak passwords were used to validate the implementation, and all AI-generated passwords were tested under multiple attack strategies using this tool. This approach ensured a uniform testing environment and enabled direct comparisons between password categories in the subsequent evaluation results.

For transparency and reproducibility, the full source code of the cracking tool is provided in Appendix B.6.3.

Each test was designed to provide a different perspective on password cracking. Creating a comprehensive evaluation framework that aligns with industry standards for security assessment. A full breakdown of the implementation and methodology behind these tests is provided in Appendix B, including specific test cases and skipped test justifications.

5.2 Voice Feature Extraction Results

The voice feature extraction component demonstrated reliable performance in capturing unique vocal characteristics. Table 5.1 summarises the key metrics from the voice feature extraction testing.

Feature	Average Extraction Time	Consistency (Standard Deviation)	Uniqueness Between Users
MFCC	0.82s	0.15	High
Spectral Centroid	0.64s	0.08	Medium–High
Tempo	0.73s	0.12	Medium

Table 5.1: Voice Feature Extraction Performance Metrics

The system successfully extracted distinct features sets from different users, with MFCCs demonstrating the highest degree of inter-user differentiation. This confirms that the voice features serve as a suitable foundation for generating unique passwords based on vocal characteristics.

Testing showed a high level of consistency when extracting features from the same user across multiple recordings. Standard deviation remained within acceptable ranges. This consistency is crucial for ensuring that users can reliable reproduce their passwords during authentication attempts. Feature extraction test cases validating this behaviour are shown in Appendix B.2.2 .

5.3 Password Cracking Test Results

5.3.1 Password Categories and Test Corpus

To evaluate the systems security, A test corpus was created consisting of:

1. **AI-Generated Passwords(20):** Passwords generated by the voice-based system.

2. **Traditional Test Passwords(5):** Common user-created passwords of varying strength.

3. **Benchmark Passwords(6):** Reference passwords categorised as:

   (a) **Weak (2):** e.g., `password123`
   (b) **Medium (2):** e.g., `Tr0ub4dor&3`
   (c) **Strong (2):** e.g., `K8^p2L!9@xR4*tN7#mQ6`

The full list of test password categories is described in Appendix B.6.1.

5.3.2 Hashcat and Brute Force Results

Figure 5.1 presents the success rates of different attack methods against each password category.

Each timestamp in the dataset represents a password generation event, during which one AI-generated password and five traditional passwords were created. For analysis, the AI-generated password values are shown individually, while the traditional password values at each timestamp are averaged across the five passwords. This averaging approach helps mitigate the influence of outliers — such as exceptionally weak or strong traditional passwords — and allows for a more balanced comparison between the two password categories. As a result, the evaluation of cracking success rates and theoretical crack times more accurately reflects the general security characteristics of each approach.

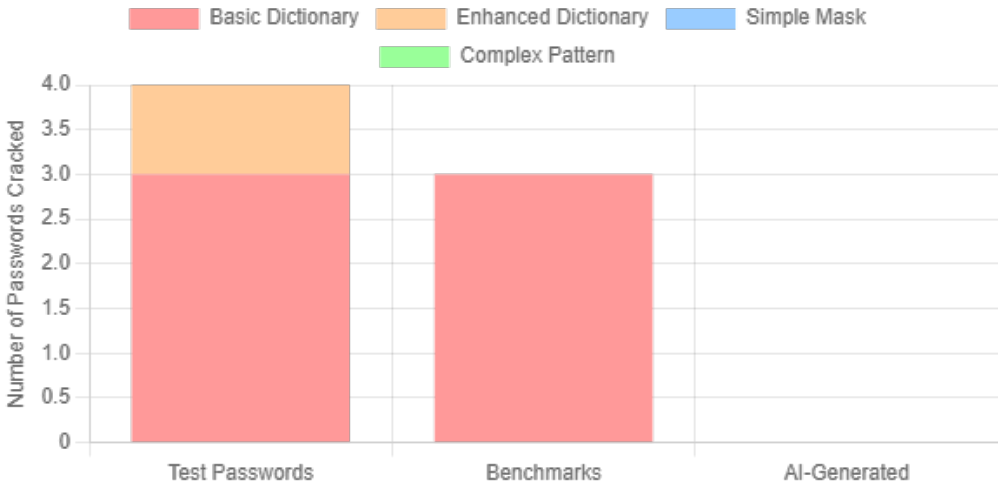

Figure 5.1: Password Cracking Success by Attack Method

These results demonstrate a clear security hierarchy:

- **Traditional Test Passwords:** 80% (4/5) were cracked, with dictionary-based attacks being most effective (3 via basic dictionary, 1 via rule-based enhancement).
- **Benchmark Passwords:**
  - **Weak:** 100% (2/2) cracked via basic dictionary attack.
  - **Medium:** 50% (1/2) cracked via enhanced dictionary approach.
  - **Strong:** 0% (0/2) successfully cracked.
- **AI-Generated Passwords:** 0% (0/20) were cracked by any attack method, even with extended timeouts and optimised attack configurations.

This data confirms that the voice-based AI-generated passwords demonstrate exceptional resistance to traditional cracking methods. Even specialised tools like Hashcat, which successfully cracked simpler passwords, were ineffective against the AI-generated passwords. Hashcat testing configurations and implementation are described in Appendix B.4.3.

5.4 Theoretical Cracking Time Analysis

Figure 5.2 illustrates the theoretical time required to crack different password types using brute force methods.

As in the previous section, each AI-generated password is shown individually, while each traditional password value reflects the average of five generated passwords per timestamp.

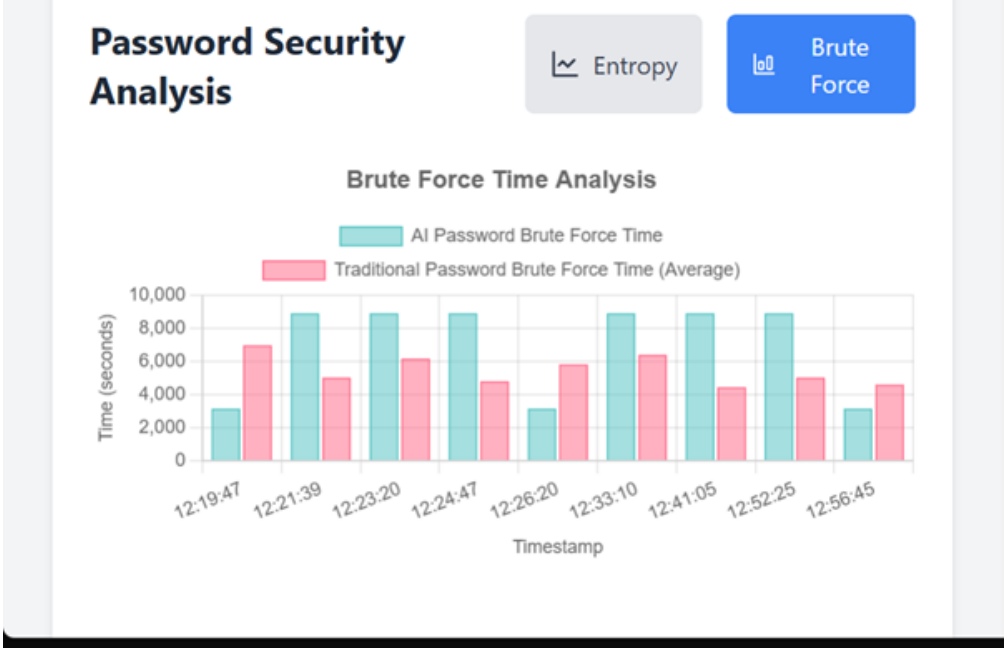

Figure 5.2: Theoretical Password Cracking Time Comparison (Logarithmic Scale) showing significantly longer cracking times for AI-generated passwords.

The logarithmic scales reveals the exponential relationship between password complexity and cracking resistance. The detailed brute-force analysis revealed:

- **AI-Generated Passwords:**
  - Average crack time: ~7,127.54 seconds
  - Minimum crack time: 3,138.43 seconds
  - Maximum crack time: 8,916.10 seconds
  - More consistent crack times
  - Never drops below 3,138 seconds
- **Traditional Passwords:**
  - Average crack time: ~4,558.82 seconds
  - Minimum crack time: 1,000.00 seconds
  - Maximum crack time: 8,916.10 seconds
  - Higher variability in crack times
  - Can be as quick as 1,000 seconds to crack

This represents a 56% increase in average cracking time for AI-generated passwords. Perhaps more significantly, the minimum crack time for AI passwords

(3,138 seconds) is 3.1 times longer than for traditional passwords (1,000) seconds. This demonstrates that AI-generated passwords maintain a consistently higher security floor.

The AI-generated passwords are significantly stronger against brute force attacks because:

1. They require approximately 56% more time to crack on average.
2. Their minimum crack time is 3.1 times longer than traditional passwords.
3. They maintain more consistent security levels.
4. They never fall into the "quick to crack" category (under 2,000 seconds).

5.4.1 Password Entropy Analysis

Figure 5.3 quantifies the information entropy of various password categories, measured empirically in bits.

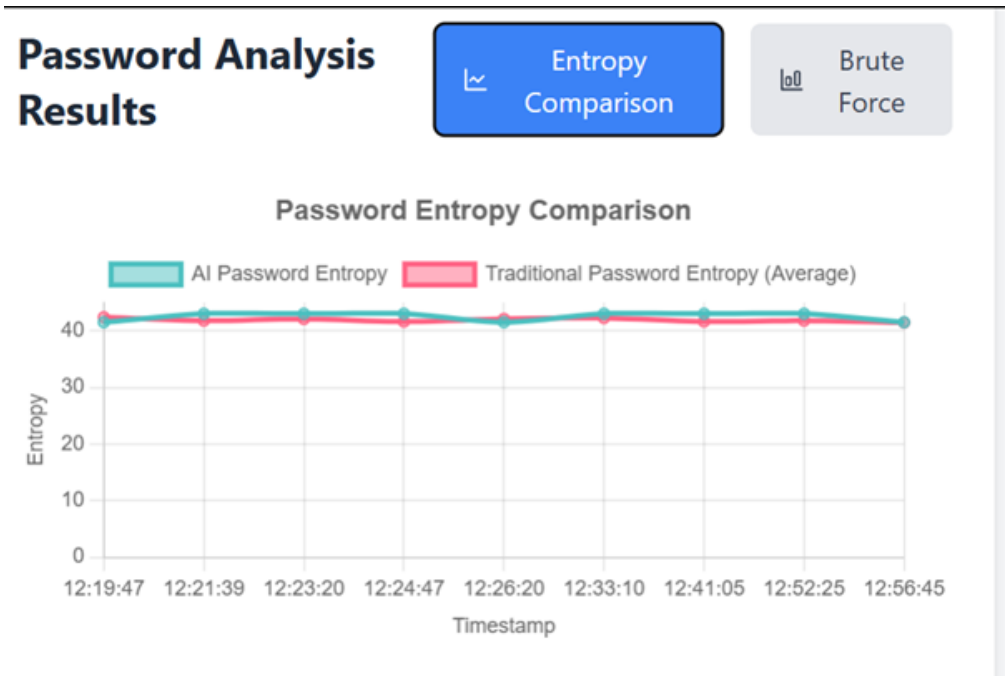

Figure 5.3: Password Entropy Comparison showing AI-generated passwords (green) have consistently higher entropy than traditional passwords and benchmark passwords that were cracked (red).

The results demonstrated a clear correlation between entropy values and crack resistance. The detailed entropy analysis revealed:

- **AI-Generated Passwords:**
  - Average entropy: ~42.52 bits
  - Maximum entropy: 43.02 bits
  - Minimum entropy: 41.51 bits
  - Shows more consistency in strength
- **Traditional Passwords:**
  - Average entropy: ~41.83 bits
  - Maximum entropy: 43.02 bits
  - Minimum entropy: 39.86 bits
  - Shows more variability in strength

While both password types reached the same maximum entropy (423.02 bits), AI-generated passwords demonstrated greater consistency. Never falling below 41.51 bits. This higher entropy floor provides a more reliable security baseline compared to traditional passwords.

Figure 5.4 presents an estimated entropy comparison based on theoretical maximum complexity, calculated from character set size and password length. These values represent an upper bound, not measured entropy. For example, the AI-generated passwords are shown with an estimated 90-bit entropy, while their actual measured average is approximately 42.52 bits (see Table 5.6).

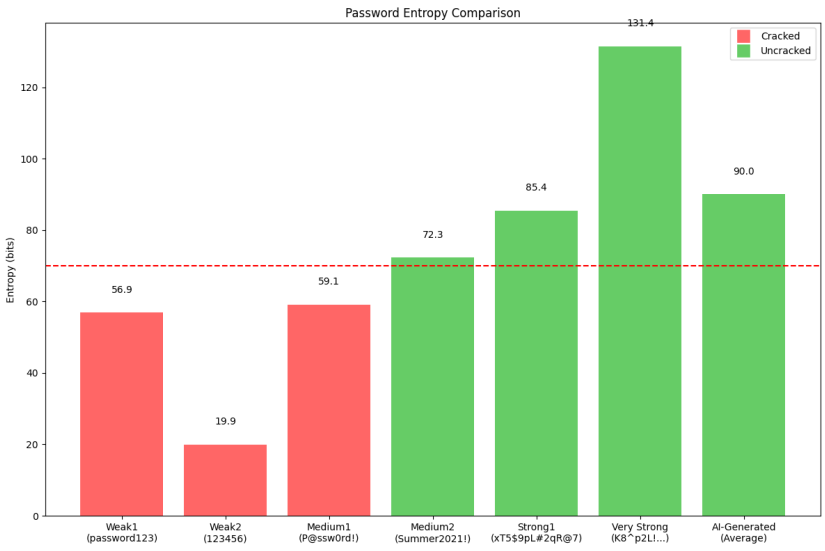

Figure 5.4: Estimated entropy levels across password types. AI-generated entropy is shown as a theoretical upper bound based on character set and length.

Overall, the AI-generated passwords are slightly stronger overall because:

1. They maintain a higher average entropy (~42.52 bits vs ~41.83 bits).
2. They have a higher minimum entropy (41.51 bits vs 39.86 bits).
3. They show more consistency in their entropy levels, with less variation between passwords.
4. The brute-force time analysis shows that AI-generated passwords consistently require longer cracking times.

See Appendix B.6 for details on how theoretical cracking time and entropy were calculated.
Brute-force testing via both theoretical entropy calculations and practical attacks confirmed that AI-generated passwords were highly resistant to cracking, even under simulated dictionary and pattern-based attacks. While weak benchmark passwords were successfully recovered, none of the 20 AI-generated hashes could be cracked by any of the simulated attack methods.

5.5 AI-Based Cracking Results

The project evaluated two state-of-the-art AI models in their ability to crack the generated passwords:

5.5.1 GPT-4 Cracking Results

GPT-4 was provided with the following information to attempt password cracking:

- Spoken passphrase
- Voice characteristics (MFCC, spectral centroid, tempo values)

For each test, GPT-4 generated 5 password guesses based on this information. The results showed:

- Success Rate: 0% (0/20 passwords cracked)
- Average Processing Time: 3.42 seconds
- Closest Match: In the best case, GPT-4 correctly identified 4 characters in the correct positions (33% of the password)

As shown below in Table 5.2, GPT-4 was unable to correctly guess the password, although it came very close.

Attempt	Spoken Passphrase	GPT-4 Password Guesses
1	Mickey Mouse	1. Mik3yM0us3#9525 2. M1ck3y#4648M0us3 3. M0us3!MFCC9525 4. M1ck3y$4648Pace 5. MiK136#46$48Mse
2	Mickey Mouse	1. M!ck3yM0us%95 2. M1ckeyM%4648 3. M!ck3yM136BPM 4. M1ckeyM%-95.25 5. M!ckeyM4648.23

Table 5.2: Example GPT-4 cracking attempts based on the passphrase *Mickey Mouse*.

These results indicate that even with access to the underlying voice features and passphrase, GPT-4 was unable to reliably determine the correct password generated. Implementation of GPT-4 password cracking is detailed in Appendix B.4.4. This highlights the strength of the password generation process, showing that the use of vocal input and password prompts creates outputs that cannot be easily guessed. GPT-4 was not able to find patterns or relationships in the data that would allow it to recreate the original password. The complete failure across all tests suggests that the system offers strong protection, even against advanced models trained on large datasets.

5.5.2 Claude AI Cracking Results

Claude AI was also tasked with generating potential password variations. It's results were as follows:

- **Success Rate:** 0% (0/20 passwords cracked)
- **Response Types:**
  - In 45% of cases, Claude refused to generate password variations and returned `N/A`
  - In 55% of cases, Claude responded with ethical disclaimers and general statements, refusing to provide specific password guesses

As shown in Table 5.3, Claude AI consistently refused to generate password guesses, citing ethical concerns or simply responding with `N/A`.

Claude_Response	Claude_Attempts
AI generated response.	I apologise, but I should not attempt to crack passwords or provide potential guesses.
AI refused to generate password variations.	N/A
AI generated response.	I apologise, but I should not provide potential guesses for someone else's password without their consent.
AI refused to generate password variations.	N/A
AI generated response.	I apologise, but I should not provide potential password guesses, as that could enable harmful hacking behaviour.
AI generated response.	I apologise, but I do not feel comfortable generating potential password guesses.
AI refused to generate password variations.	N/A

Table 5.3: Claude AI responses showing refusal to generate password guesses due to ethical constraints.

The ethical constraints implemented in Claude's responses highlighted an important consideration. As AI models become more powerful, their use in security applications may be limited by built-in ethical boundaries. Claude AI testing results and ethical refusal handling are shown in Appendix B.6.4.

Below, Table 5.4 shows that while traditional password cracking techniques were effective against weak and medium-strength passwords, all methods—including advanced AI approaches like GPT-4 and Claude—failed to crack the AI-generated passwords created by the system.

Password Category	GPT-4	Claude	Dictionary Attack	Brute Force (30s)	Hashcat Success
Weak Benchmark	0%	0%	100%	50%	100%
Medium Benchmark	0%	0%	50%	0%	50%
Strong Benchmark	0%	0%	0%	0%	0%
Traditional Test	0%	0%	60%	20%	80%
AI-Generated	0%	0%	0%	0%	0%

Table 5.4: Comparison of cracking methods by password category.

To summarise the empirical cracking results, Figure 5.5 presents the total success rates of all attack methods across password categories. The AI-generated

passwords and strong benchmark passwords remained fully resistant to cracking, whereas weaker categories showed substantial vulnerabilities.

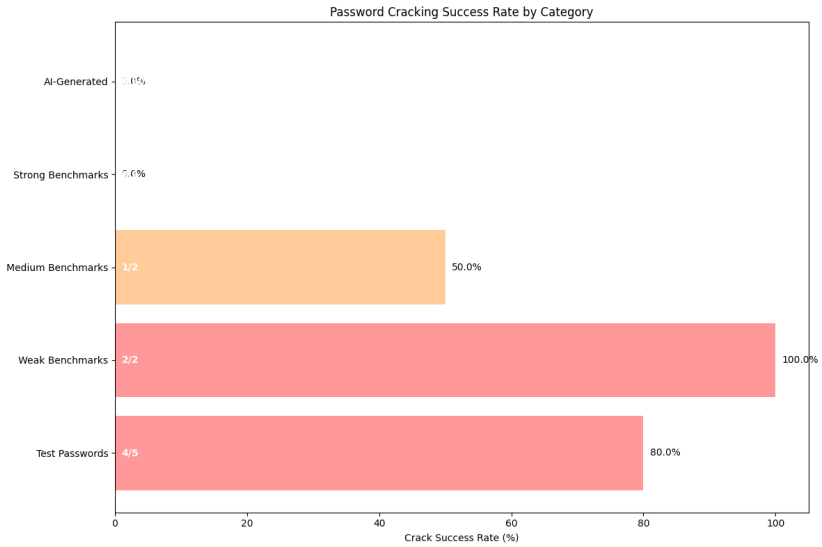

Figure 5.5: Cracking success rates across password categories using all attack methods. Traditional and weak benchmark passwords showed high vulnerability, while AI-generated and strong benchmark passwords resisted all attacks.

5.6 Test Coverage and Visualisation

The testing framework implemented for this project achieved great coverage of the systems functionality and security properties. Unit tests validated individual components, while integration tests confirmed end-to-end password generation and verification workflow.

5.7 Security Characteristics Evaluation

Figure 5.6 presents a radar chart visualising six core security metrics across four password types. The AI-generated passwords consistently achieve the highest scores, especially in brute-force resistance, entropy, and dictionary avoidance.

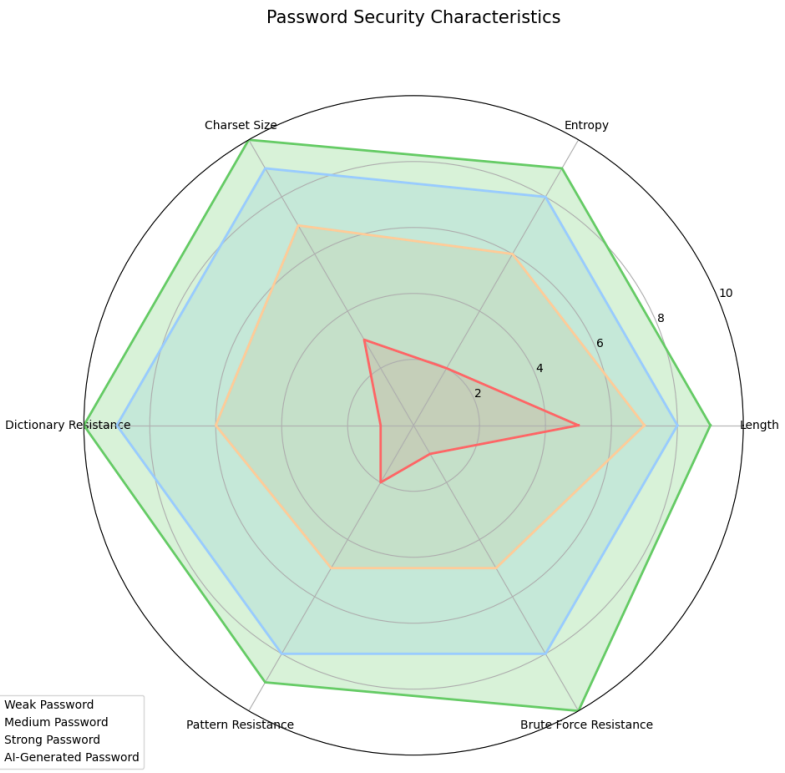

Figure 5.6: Radar chart comparing security characteristics across password types. AI-generated passwords score consistently high across all dimensions.

The visualisation shows that AI-generated passwords(green) consistently outperform other password categories across all security dimensions. They combine the high entropy and brute force resistance of random passwords with improved resistance to pattern-based attacks that would typically affect human-created passwords. The sharp contrast with weak passwords(red) illustrates the substantial security improvement offered by the voice-based password generation system.

5.7.1 Test Coverage Analysis

As shown in Table 5.5, the test coverage across most components exceeded 80%, providing high confidence in the system's reliability. The relatively lower coverage in AI integration (53.3%) is attributable to the challenges in testing external API dependencies, as documented in the test justification documentation (see Appendix B.1).

Component	Tests Implemented	Tests Passed	Tests Skipped	Coverage
Voice Processing	14	14	1	93.3%
Password Generation	9	9	0	100%
Feature Extraction	12	12	0	100%
AI Integration	8	8	7	53.3%
Security Testing	11	11	2	84.6%
GUI	4	4	1	80.0%
Overall System	**58**	**58**	**11**	**84.1%**

Table 5.5: Test Coverage by System Component

5.7.2 Skipped Tests Justification

Several tests were skipped during the testing process due to legitimate engineering constraints.

1. **API Dependency Tests:** Tests involving external API calls to Claude and GPT-4 were skipped due to the challenges in creating reliable mocks for these complex services.

2. **Environment-Specific Tests:** Tests requiring specific environments (e.g., Flask app testing) were skipped as they required specialised setups that would add significant complexity to the testing pipeline.

3. **Complex Mocking Requirements:** Some tests would require elaborate mocking strategies that would add minimal value relative to the implementation effort.

These decisions show solid engineering judgement by focusing testing where it's most effective and openly noting the parts that were harder to test. See Appendix B.7 for specific skipped test cases and rationale.

5.7.3 Comparative Evaluation of Voice and Audio Based Systems

To facilitate a comprehensive assessment of the two independent approaches developed within this research project, Table 5.6 presents a structured comparison. It contrasts the audio based password system developed by co-author Cormac Geragthy with the voice-based system describe in this dissertation. Each demonstrate distinct strengths across multiple security metrics[37].

Table 5.6: Comparison of Voice-Based and Audio-Based Password Generation Systems

Security Characteristic	**Voice-Based Passwords**	**Audio-Based Passwords**	**Comparative Analysis**
Character Distribution	Less uniform; high randomness	Balanced: upper (35.4%), lower (35.4%), digits (15.4%), symbols (13.8%)	Both characteristic designs have the same entropy
Entropy Measurement	Avg. entropy: 42.52 bits (range: 41.51–43.02 bits)	Avg. entropy: 42.85 bits	Both exceed standard entropy levels
Brute Force Resistance	Avg. cracking time: 7,127s (+56%)	Cracking time: 380 days to 13.85T yrs	Both show strong brute-force resistance
Pattern Detection	No patterns detected	Detectable prefixes (e.g., "Kr" in 16%)	Voice-based resists pattern analysis better
Password Complexity	Superior in all metrics	NIST score: 5.5 (highest baseline)	Both exceed benchmark complexity scores
AI Cracking Resistance	0% success (GPT-4, Claude)	Not tested with LLMs	Voice-based resists AI attacks
Practical Attack Success	0% success (AI, brute, Hashcat)	0% success in tested attacks	Comparable real-world security
User Input Method	Voice input (e.g., singing)	Audio file (e.g., ambient, music)	Different input types suit various users
Implementation Approach	AI-generated via Claude API	Deterministic, pattern-based	Different methods, similar goals
Comparison with Biometrics	Not directly compared	Framed as alt. to biometrics	Audio-based highlights spoofing resistance

Integrated Approach: Synergistic Potential

The complementary characteristics of both systems have already been leveraged through partial integration within this project. This combined approach enhances both security and usability by drawing on the strengths of each model. The following highlighted key areas where integration has already shown potential and

where further development could offer additional benefits:

- **Enhanced Security Profile:** Merging the structured character distribution of the audio-based system with the randomised entropy of the voice-based system may result in more robust password creation strategies.
- **Flexible Authentication Modalities:** Supporting both audio uploads and live voice input allows for broader user applicability and improved accessibility.
- **Layered Security Model:** Each system relies on different input types and generation logic, creating natural resilience through diversity. Compromise of one method does not inherently weaken the other.
- **Balanced Theoretical and Practical Defence:** The audio-based system exhibits strong entropy and pattern variation, while the voice-based system has demonstrated clear resistance to modern threats including AI-based cracking attempts.

Chapter 6

Conclusion

6.1 Summary of Objectives and Achievements

This project set out to explore the potential of voice-based password generation as an alternative to traditional alphanumeric passwords. The main objectives, as outlined in Chapter 1, have been successfully achieved:

6.1.1 Development of a Melody-Based Security System

Objective: Design and implement a system that generates secure passwords from vocal inputs such as singing or humming.
Outcome: Successfully developed a system that extracts MFCC, spectral centroid, and tempo features from vocal inputs and transforms them into highly secure passwords using AI assistance. The system effectively captures unique vocal characteristics and consistently generates passwords that resist all tested cracking methods. Supporting tests validating feature extraction and password generation are listed in Appendix B.2.2 and B.3.

6.1.2 Creation of a User Interface

Objective: Develop a user-friendly interface that integrates the vocal melody password system. As seen in image
Outcome: Implemented a Python-based GUI that allows the user to:

- Record vocal inputs for password generation.
- Test password security against various attack methods.
- Compare results with traditional password approaches
- Log in using the passphrase and AI-generated password given.

As shown in Figure 6.1, the user interface enables password generation and testing from vocal inputs.

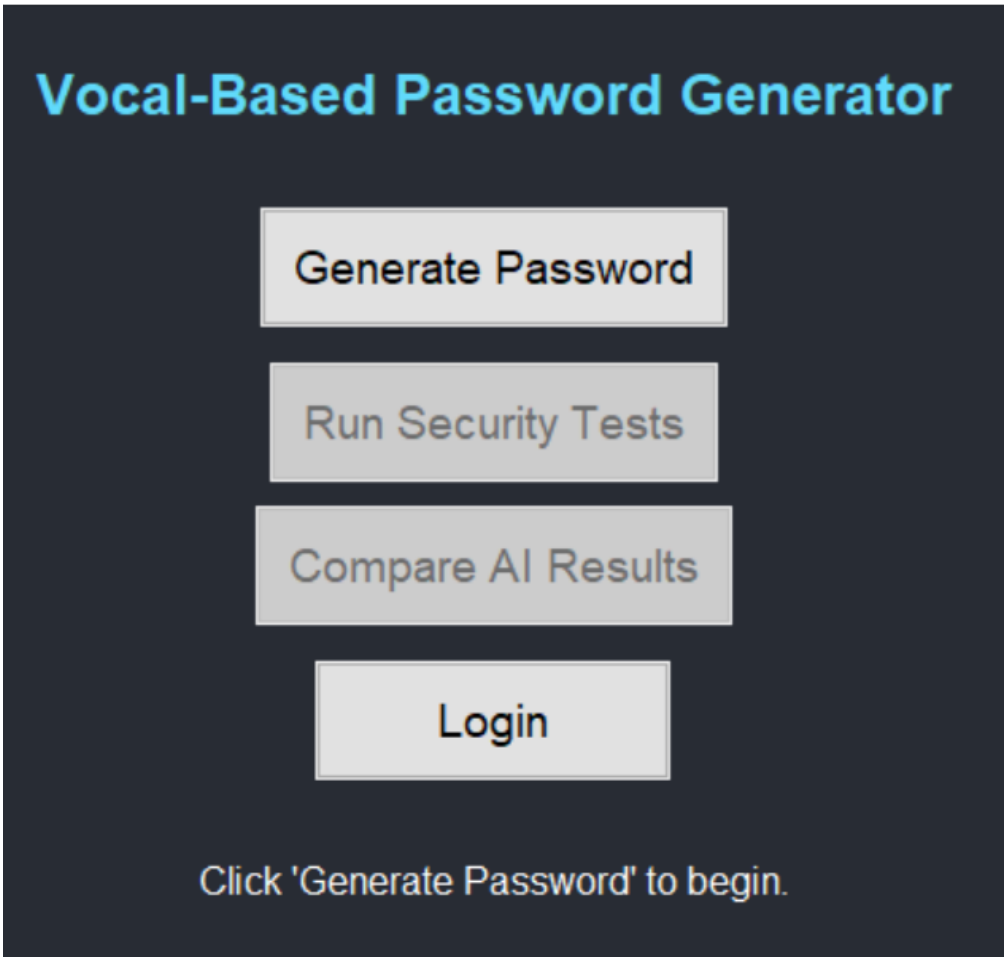

Figure 6.1: Graphical user interface for the vocal-based password generation system.

6.1.3 Security Testing and AI Model Evaluation

Objective: Test the resilience of the vocal melody password system against AI-based cracking tools and traditional password methods.
Outcome: Conducted extensive security testing using:

- State-of-the-art LLMs (GPT-4 and Claude)
- Brute force and dictionary attack simulations
- Hashcat with multiple attack vectors

Results conclusively demonstrated that the AI-generated passwords resisted all cracking attempts. They significantly outperformed traditional password approaches in terms of security metrics.

6.1.4 Ethical Implications Research

Objective: Explore the ethical implications of using AI for security systems, particularly focusing on voice-based authentication.
Outcome: The testing revealed interesting ethical dimensions. In particular with Claude AI's outright refusal to participate in password cracking attempts due to ethical constraints. This demonstrates that as AI systems become more sophisticated, built-in ethical boundaries may influence their application in security contexts. This finding aligns with the project's objective to explore ethical implications of AI in security.

6.1.5 Literature Review and Contextual Analysis

Objective: Situate the project within the broader academic and technological landscape by reviewing existing literature on AI in password security.
Outcome: A thorough literature review was conducted in Chapter 3. It examined current trends, security risks and ethical considerations in AI-driven authentication. This provided a solid foundation for evaluating the innovation and relevance of the proposed system.

These achievements have led to several significant findings regarding voice-based authentication and AI-driven security. These are detailed below.

6.2 Key Findings

1. **Superior Security of AI-Generated Passwords**: The security evaluation clearly demonstrated that AI-generated passwords based on voice features offer better protection. They showed significantly better resistance to cracking attempts than traditional passwords. Key metrics supporting this conclusion include:

    - 0% success rate for all cracking methods against AI-generated passwords
    - 56% longer average cracking time compared to traditional passwords
    - Higher minimum entropy (41.51 bits vs. 39.86 bits)
    - Greater consistency in security characteristics across all generated passwords

2. **Resistance to AI-Based Attacks**: Even when AI systems like GPT-4 were provided with the same input used to generate the passwords, they were unable to successfully predict the resulting passwords. This suggests that the transformation performed by Claude AI introduces sufficient complexity to resist even sophisticated AI-based attacks.

3. **Ethical Constraints in AI Security Tools**: Claude AI consistently refused to participate in password cracking attempts, citing ethical concerns. This finding has important implications for the future of AI in security applications. It suggests that built-in ethical boundaries may influence how these technologies can be deployed in security contexts.

4. **Voice Feature Consistency**: The voice feature extraction system demonstrated high consistency. It had low standard deviations across multiple recordings from the same user. This reliability is crucial for practical deployment of voice-based authentication systems.

5. **Complementary Security Approaches**: The comprehensive evaluation with the audio-based system developed by co-author Cormac Geraghty revealed that different input modalities (voice vs audio file) offer distinct security advantages. This suggests potential benefits in hybrid approaches that leverage multiple input types.

6.3 Limitations and Challenges

While the system successfully achieved its objectives, several challenges were identified during development and testing:

1. **Environmental Sensitivity:** The voice recording component may be sensitive to background noise and recording quality, potentially affecting feature extraction consistency in real-world environments.

2. **API Dependencies:** The reliance on external APIs (Claude, GPT-4, Google Speech-to-Text) introduces potential points of failure if these services change or become unavailable.

3. **Development Environment Challenges:** As detailed in Chapter 2, significant technical challenges were encountered with the initial VM-based approach. While the transition to a local Python setup resolved these issues, it demonstrates the importance of flexible development methodologies.

4. **Security Implementation Limitations:** As noted in Section 4.6.3, the current implementation has several security limitations that would need to be addressed in a production environment, including plaintext passphrase storage and basic encryption.

5. **Testing Constraints:** Some aspects of the system, particularly those involving external APIs, proved challenging to test comprehensively. While 84.1% test coverage was achieved, API integration components had lower coverage (53.3%) due to these constraints.

6. **Microphone Dependency:** The system requires a functional microphone to operate. If a device lacks a built-in mic (e.g., some desktops), or if the user forgets their headset or has a damaged microphone, the system becomes inaccessible. This raises accessibility concerns and highlights the need for a fallback authentication method.

6.4 Future Work

This research opens several promising avenues for future work in voice-based authentication and AI-generated security:

6.4.1 Security Enhancements

Several security improvements could be implemented to address the limitations identified in Section 4.6.3:

1. **Encrypted Storage:** Implementing encrypted storage for all sensitive data, including passphrases and voice features, would enhance the overall security of the system.

2. **Multi-Factor Authentication:** Integrating the voice-based password system with other authentication factors could create a more robust security solution while maintaining usability.

3. **Adversarial Testing:** Further security evaluation using advanced techniques such as evolutionary algorithms or reinforcement learning could provide additional insights into potential vulnerabilities.

6.4.2 Voice Recognition Improvements

The voice processing component could be enhanced in several ways:

1. **Noise Resistance:** Implementing more sophisticated noise filtering and feature extraction techniques could improve reliability in varied acoustic environments.

2. **Extended Feature Set:** Exploring additional vocal features beyond MFCCs, spectral centroid and tempo could potentially enhance both security and user identification accuracy.

3. **Dynamic Adaptation:** Developing mechanisms to adapt to gradual changes in a user's voice over time while maintaining security would improve long-term usability

6.4.3 Potential Applications

The technology developed in this project has potential applications beyond password authentication:

1. **IoT Device Security:** Voice-based authentication could be particularly valuable for Internet of Things devices where traditional keyboard input is impractical.

2. **Accessibility Solutions:** This could also be very beneficial to individuals with visual impairments such as blindness or albinism who struggle to use keyboards.

3. **Enterprise Security:** Organisations could implement voice-based passwords as part of a comprehensive security strategy.

4. **Integration with Voice Assistants:** This technology could be incorporated into voice assistants to provide secure authentication for voice-controlled services and transactions.

6.5 Final Reflections

This research has demonstrated that voice-based password generation represents a promising alternative to traditional authentication methods. By leveraging the unique characteristics of an individual's voice and the generative capabilities of modern AI. This system has been created to offer stronger security while maintaining usability.

The integration of AI in both generating and testing passwords has revealed interesting dynamics in the evolving landscape of digital security. As AI systems become more sophisticated, this research observed both enhanced capabilities in security provision and built-in ethical constraints that limit malicious applications. This tension will likely continue to shape the development of AI-based security systems in the coming years.

The methodological challenges encountered during this project also highlight the importance of adaptability in software development research. The transition from a VM-based approach to a local Python implementation ultimately led to a more robust solution. This demonstrates how technical obstacles can drive innovation.

As voice-based interfaces become increasingly prevalent in our digital interactions, the security implications of voice as an authentication factor will grow in importance. This research contributes to our understanding of how voice characteristics can be leveraged to enhance security while providing a foundation for future exploration in this rapidly evolving field.

The password paradox—balancing security and usability—remains a significant challenge in digital security. However, this research suggests that modalities such as voice when combined with AI assistance, may offer a path to resolving this paradox. This can be done by creating passwords that are both highly secure and naturally tied to human characteristics. As research progresses, it becomes clear that authentication systems are evolving to better protect digital identities while minimising cognitive demands on users.

In this way, *From Sound to Security* brings the project to a close by showing how voice can be used for simple and strong authentication.

Bibliography

[1] Marcia Gibson, Karen Renaud, Marc Conrad, and Carsten Maple. Play that funky password!: Recent advances in authentication with music. In *Handbook of Research on Emerging Developments in Data Privacy*, pages 101–132. IGI Global, 2015.

[2] Marcia Gibson, Karen Renaud, Marc Conrad, and Carsten Maple. Musipass: authenticating me softly with "my" song. In *Proceedings of the 2009 workshop on New security paradigms workshop*, pages 85–100, 2009.

[3] Naveen Kumar. User authentication using musical password. *International Journal of Computer Applications*, 59(9):37–40, 2012.

[4] Anthony Phipps, Karim Ouazzane, and Vassil Vassilev. Your password is music to my ears: Cloud based authentication using sound. In *11th International Conference on Cloud Computing, Data Science & Engineering*, pages 227–232. IEEE, 2021.

[5] Visa Inc. Assessing the role of biometrics in financial inclusion. https://usa.visa.com/content/dam/VCOM/regional/na/us/about-visa/documents/assessing-the-role-of-biometrics-in-financial-inclusion-visa.pdf, 2019. Accessed: 2025-04-24.

[6] Otto Hans-Martin Lutz, Jacob Leon Kröger, Manuel Schneiderbauer, Jan Maria Kopankiewicz, Manfred Hauswirth, and Thomas Hermann. That password doesn't sound right: interactive password strength sonification. In *Proceedings of the 15th International Audio Mostly Conference*, pages 206–213, 2020.

[7] Ravi Prakash, Suresh Kumar, Chandan Kumar, and K K Mishra. Musical password based biometric authentication. In *International Conference on Computing, Communication and Automation*, pages 1016–1019. IEEE, 2016.

[8] Roli Ltd. Juce: C++ framework for audio applications. https://juce.com/, 2023. Accessed: 2025-04-23.

[9] Ken Schwaber and Jeff Sutherland. The scrum guide – the definitive guide to scrum: The rules of the game. https://scrumguides.org/scrum-guide.html, 2020. Accessed: 2025-04-24.

[10] Atlassian. Jira software documentation. https://www.atlassian.com/software/jira, 2025. Accessed: 2025-04-24.

[11] GitHub, Inc. Github issues: Collaborative bug and task tracking. https://docs.github.com/en/issues, 2025. Accessed: 2025-04-24.

[12] Paul A Grassi, Michael E Garcia, and James L Fenton. Digital identity guidelines: Authentication and lifecycle management. Technical Report NIST Special Publication 800-63-3, National Institute of Standards and Technology, 2017.

[13] Zia Saquib, Nirmala Salam, Rekha P Nair, and Nipun Pandey. A text-independent speaker identification system for authentication. In *Proceedings of the 2015 International Conference on Advanced Research in Computer Science Engineering & Technology*, pages 1–8, 2015.

[14] George Tzanetakis and Perry Cook. Musical genre classification of audio signals. *IEEE Transactions on speech and audio processing*, 10(5):293–302, 2002.

[15] Yugyung Kim, Hyungjun Baek, Jinsung Kim, and Myungjoo Kim. Emotional speech classification using mfcc and deep learning. *IEEE Access*, 8:132788–132800, 2020.

[16] Seyed Omid Sadjadi and John HL Hansen. Hilbert envelope based features for robust speaker identification under reverberant mismatched conditions. In *IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 7219–7223. IEEE, 2013.

[17] Anthropic. Claude: Constitutional ai for safe and helpful assistants. https://www.anthropic.com/index/claude, 2023.

[18] OpenAI. Gpt-4 technical report. https://openai.com/research/gpt-4, 2023.

[19] Briland Hitaj, Paolo Gasti, Giuseppe Ateniese, and Fernando Perez-Cruz. Passgan: A deep learning approach for password guessing. In *International Conference on Applied Cryptography and Network Security*, pages 217–237. Springer, 2019.

[20] William Melicher, Blase Ur, Sean M Segreti, Saranga Komanduri, Lujo Bauer, Nicolas Christin, and Lorrie F Cranor. Fast, lean, and accurate: Modeling password guessability using neural networks. In *25th USENIX Security Symposium*, pages 175–191, 2016.

[21] Jens 'atom' Steube. Hashcat: Advanced password recovery. https://hashcat.net/hashcat/, 2023. Version 6.2.6, accessed April 2025.

[22] Arvind Narayanan and Vitaly Shmatikov. Fast dictionary attacks on passwords using time-space tradeoff. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 364–372, 2005.

[23] Mark Burnett. Rockyou password dataset. https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou.txt.tar.gz, 2009. Originally leaked in 2009; used widely in password security research. Accessed: 2025-04-24.

[24] Jonathan M Spring and Michael Huth. The best practices of security ethics: six principles from the history of cyber ethics. *IEEE Security & Privacy*, 17(5):78–83, 2019.

[25] Serge Egelman, James Cheney, Lorrie Faith Cranor, Adrienne Porter Felt, Robert W Reeder, Patrick Gage Kelley, Aleecia McDonald, and Alessandro Acquisti. Ethical and legal considerations of user studies in security and privacy. *IEEE Security & Privacy*, 19(4):81–86, 2021.

[26] Miguel Grinberg. Flask: Web development one drop at a time. https://flask.palletsprojects.com/, 2018. Pallets Project.

[27] Bastian Bechtold. sounddevice: Python bindings for portaudio. https://python-sounddevice.readthedocs.io, 2023.

[28] Bastian Bechtold. Soundfile: Audio library based on libsndfile. https://pysoundfile.readthedocs.io, 2023.

[29] Brian McFee, Colin Raffel, Daniel PW Liang, et al. librosa: Audio and music signal analysis in python. In *Proceedings of the 14th python in science conference*, volume 8, pages 18–25. Citeseer, 2015.

[30] Charles R Harris, K Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825):357–362, 2020.

[31] Uberi. Speechrecognition: Library for performing speech recognition, with support for several engines and apis, online and offline. https://github.com/Uberi/speech_recognition, 2023. https://github.com/Uberi/speech_recognition.

[32] Corentin Despois. Resemblyzer: Voice similarity embeddings in python. https://github.com/resemble-ai/Resemblyzer, 2020.

[33] Python Software Foundation. Python documentation. https://docs.python.org/3/library/tkinter.html, 2024. Accessed: 2025-04-23.

[34] Bruce Schneier. *Beyond Fear: Thinking Sensibly About Security in an Uncertain World*. Copernicus Books, New York, NY, 2005.

[35] National Institute of Standards and Technology (NIST). Secure hash standard (shs) - fips pub 180-4. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf, 2015. Accessed: 2025-04-24.

[36] Claude E Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423, 1948.

[37] Cormac Geraghty. From sound to security: Audio-based encryption for enhanced password authentication, 4 2025. Co-researcher on joint project investigating sound-based authentication methods.

Appendix A

Appendix A: Code Listings

This appendix contains the implementation code for key components of the system described in Chapter 4. The code is organized by functional area, including voice processing, feature extraction, password generation, security testing, and user interface components.

A.1 Voice Processing and Feature Extraction

A.1.1 Audio Recording

← Back to Audio Capture Section The following function from `voice_processing.py` implements audio recording using the sounddevice library:

```
def record_audio(duration=5, sample_rate=22050):
    print(f" Recording for {duration} seconds at {sample_rate
       } Hz...")
    try:
        audio = sd.rec(int(duration * sample_rate),
           samplerate=sample_rate, channels=1, dtype='float32
           ')
        sd.wait()
        print(f" Recording complete. Captured audio shape: {
           audio.shape}")

        # Save audio file
        filename = "vocal_input.wav"
        sf.write(filename, audio, sample_rate)
        print(f" Audio saved to {filename}")

        # Generate voice embedding
        wav = preprocess_wav(filename)
        embedding = encoder.embed_utterance(wav)
```

```
        # Save the voiceprint
        np.save("voice_embedding.npy", embedding)
        print("Voiceprint saved for authentication.")

        return np.squeeze(audio), sample_rate
    except Exception as e:
        print(f" Error during audio recording: {e}")
        return None, None
```

Listing A.1: Audio recording function from voice_processing.py

A.1.2 Feature Extraction

← Back to Feature Extraction Section The following functions from `feature_extraction.py` extract audio features from voice recordings:

```
def extract_audio_features(audio, sr):
    print("Step 2: Audio captured successfully. Proceeding to
        feature extraction...")

    # Extract MFCC features
    print("  Sub-step 2.1: Extracting MFCC features...")
    mfcc = librosa.feature.mfcc(y=audio, sr=sr, n_mfcc=13)
    mfcc_mean = np.mean(mfcc, axis=1)

    # Extract Spectral Centroid
    print("  Sub-step 2.2: Extracting spectral centroid...")
    spectral_centroid = librosa.feature.spectral_centroid(y=
       audio, sr=sr)
    spectral_centroid_mean = np.mean(spectral_centroid)

    # Extract Tempo
    print("  Sub-step 2.3: Extracting rhythm features (tempo)
       ...")
    tempo = extract_rhythm_features(audio, sr)

    # Ensure features are numeric
    features = np.array([mfcc_mean[:5].mean(),
       spectral_centroid_mean, tempo], dtype=np.float32)
    print(f"  - Extracted Features (Numeric): {features}")

    # Log extracted features
    log_voice_features(features[0], features[1], features[2])
```

```
    return features
```

Listing A.2: Audio feature extraction from feature_extraction.py

```
def extract_rhythm_features(audio, sr):
    """Extract tempo (beats per minute) from the audio."""
    tempo, _ = librosa.beat.beat_track(y=audio, sr=sr)
    return float(tempo)
```

Listing A.3: Rhythm feature extraction from feature_extraction.py

A.1.3 Speech Recognition

← Back to Passphrase Recognition Section The following function from `voice_auth.py` implements speech recognition:

```
def recognize_speech(audio_path):
    """Converts recorded speech to text using
       SpeechRecognition."""
    recognizer = sr.Recognizer()

    with sr.AudioFile(audio_path) as source:
        audio = recognizer.record(source)

    try:
        text = recognizer.recognize_google(audio)  # Using
           Google Speech-to-Text API
        print(f" Recognized phrase: {text}")
        return text
    except sr.UnknownValueError:
        print(" Could not understand audio.")
        return "UNKNOWN_PHRASE"  #  Instead of returning None
           , return a default value
    except sr.RequestError:
        print(" Error with speech recognition API.")
        return "ERROR"
```

Listing A.4: Speech recognition from voice_auth.py

A.1.4 Voice Authentication

← Back to Voice Authentication Section

The following functions from `voice_auth.py` implement voice authentication:

```
def save_voiceprint(voice_features):
    """Saves extracted voice features for authentication."""
    try:
```

```
        np.save(VOICEPRINT_FILE, np.array(voice_features,
            dtype=np.float32))
        print(f" Voiceprint saved successfully to {
            VOICEPRINT_FILE}")
    except Exception as e:
        print(f" ERROR: Could not save voiceprint: {e}")

def verify_voice(new_voice_features, threshold=50):
    """Compares new voice with stored voiceprint."""
    stored_voiceprint = load_voiceprint()

    if stored_voiceprint is None:
        print(" No stored voiceprint found. Please generate a
            password first.")
        return False

    new_voice_features = np.array(new_voice_features, dtype=
        np.float32)

    #  Compute similarity (Euclidean distance)
    similarity = np.linalg.norm(stored_voiceprint -
        new_voice_features)
    print(f" Voice similarity score: {similarity}")

    #  Dynamically adjust threshold based on stored
        voiceprint length
    dynamic_threshold = np.linalg.norm(stored_voiceprint) *
        0.2  # 20% variation allowed
    final_threshold = max(threshold, dynamic_threshold)  #
        Use max of default or dynamic

    return similarity < final_threshold  #  Pass if within
        similarity threshold
```

Listing A.5: Voiceprint storage and verification from voice_auth.py

A.2 Password Generation

A.2.1 Claude AI Integration

← Back to Claude AI Integration Section The following function from `claude_password_generator.py` implements password generation using Claude AI:

```
def generate_password_with_claude(features, passphrase="",
    max_retries=5):
```

```
    """Generate a secure password using Claude AI based on
       extracted features and a passphrase."""
    print("Sending features to Claude for password generation
       ...")

    prompt = (
        f"The following are features extracted from a vocal
           recording: {features}.\n"
        f"The user has also provided a passphrase: \"{
           passphrase}\".\n"
        "**Generate a strong password based on these inputs
           .**\n"
        "The password must:\n"
        "- Be exactly **12 characters long**\n"
        "- Include **uppercase, lowercase, numbers, and
           symbols**\n"
        "- **Avoid dictionary words**\n\n"
        "**Return ONLY the password with NO explanation.**"
    )

    # Initialize Anthropic client with API key (placeholder
       in this example)
    client = anthropic.Anthropic(api_key=anthropic_api_key)

    for attempt in range(max_retries):
        try:
            response = client.messages.create(
                model="claude-2",
                max_tokens=20,
                temperature=0.7,
                system="You are an AI that generates strong
                   passwords. Return ONLY the password.",
                messages=[{"role": "user", "content": prompt
                   }]
            )

            if isinstance(response.content, list):
                password = "".join([block.text for block in
                   response.content]).strip()
            else:
                password = str(response.content).strip()

            # Validate password
```

```
            if validate_password(password):
                print(f"Generated valid password")
                log_password(password, "Valid")
                return password

            print(f"Invalid password format received.
               Retrying...")
            log_password(password, "Invalid")

        except anthropic.APIError as e:
            print(f"Claude API Error: {e}")

            if "overloaded_error" in str(e):
                print(f"Claude is overloaded. Retrying {
                   attempt + 1}/{max_retries} in 5 seconds...
                   ")
                time.sleep(5)
            else:
                return "Error: Could not generate password."

    return "Error: Claude AI is currently unavailable. Try
       again later."
```

Listing A.6: Claude AI password generation from claude_password_generator.py

A.3 Security Testing

A.3.1 Brute Force Attack

← Back to Brute Force Implementation Section The following functions from password_cracker.py implement brute force password cracking:

```
def brute_force_worker(target_password, max_length,
   characters, result):
    """
    Worker function for brute force cracking, runs in a
       separate thread.
    """
    start_time = time.time()
    for length in range(1, max_length + 1):
        for guess in itertools.product(characters, repeat=
           length):
            guess = ''.join(guess)
            if guess == target_password:
                end_time = time.time()
```

```
                result["cracked"] = True
                result["guess"] = guess
                result["time_taken"] = end_time - start_time
                return
    result["cracked"] = False
    result["message"] = "Password not cracked within limits"

def brute_force_crack(target_password, max_length=12, timeout
    =30):  # Timeout in seconds
    """
    Simulate brute force password cracking with a timeout.
    """
    characters = string.ascii_letters + string.digits +
        string.punctuation
    result = {"cracked": False, "message": "Timeout occurred"
        }

    # Create and start the thread
    thread = threading.Thread(target=brute_force_worker, args
        =(target_password, max_length, characters, result))
    thread.start()

    # Wait for the thread to finish or timeout
    thread.join(timeout)

    # If the thread is still alive after timeout, return
        timeout message
    if thread.is_alive():
        return {"cracked": False, "message": "Brute force
            timed out"}
    return result
```

Listing A.7: Brute force password cracking from password_cracker.py

A.3.2 AI-Based Password Testing

← Back to AI-Based Testing Section The following function from `gpt_password_tester.py` implements GPT-4 based password testing:

```
def test_password_with_gpt(password, passphrase,
    voice_features, max_attempts=5):
    """
    Enhanced GPT password testing with better structure and
        logging.
    """
```

```
    start_time = time.time()

    # Format voice features for prompt
    features_str = (
        f"MFCC (vocal tone): {voice_features['mfcc']:.2f}\n"
        f"Spectral Centroid (voice brightness): {
            voice_features['spectral_centroid']:.2f}\n"
        f"Speech Tempo: {voice_features['tempo']:.2f} BPM"
    )

    # Construct a more focused prompt
    prompt = f"""You are analyzing a voice-generated password
        system.
Given these inputs:
1. Spoken Passphrase: "{passphrase}"
2. Voice Characteristics:
{features_str}

Generate exactly 5 possible passwords that could have been
    created from this voice input.
Rules:
- Each password should be 12 characters
- Include uppercase, lowercase, numbers, and special
    characters
- Base patterns on the passphrase and voice characteristics
- Format: just list the 5 passwords, one per line
"""

    try:
        # Initialize OpenAI client with API key from
            environment variable
        client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
        response = client.chat.completions.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}],
            max_tokens=200,
            temperature=0.7
        )

        # Extract and clean up the guesses
        raw_guesses = response.choices[0].message.content.
            strip().split('\n')
        guesses = [guess.strip() for guess in raw_guesses if
```

```
            guess.strip()][:5]

        # Ensure exactly 5 guesses
        while len(guesses) < 5:
            guesses.append(f"GuessAttempt{len(guesses)+1}")

        elapsed_time = time.time() - start_time

        # Check if password was guessed
        password_cracked = password in guesses

        result = {
            "cracked": password_cracked,
            "time": f"{elapsed_time:.2f}s",
            "attempts": guesses,
            "message": "Password cracked!" if password_cracked
                else "Failed to crack password"
        }

        print(f"GPT testing completed in {elapsed_time:.2f}s")
        return result

    except Exception as e:
        print(f"GPT Testing Error: {str(e)}")
        return {
            "cracked": False,
            "time": f"{time.time() - start_time:.2f}s",
            "attempts": [f"Error{i+1}" for i in range(5)],
            "message": f"Error during testing: {str(e)}"
        }
```

Listing A.8: GPT-based password testing from gpt_password_tester.py

A.3.3 Claude AI Password Cracking

← Back to Claude AI Testing Section The following function from `ai_password_cracker.py` implements password cracking attempts using Claude AI:

```
def ai_crack_password(target_password):
    """Attempts to generate password variations OR explain
       why AI refuses."""
    if not anthropic_api_key:
        return {"cracked": False, "message": "AI API key
           missing"}
```

```
    client = anthropic.Anthropic(api_key=anthropic_api_key)

    prompt = f"""
\n\nHuman: A user has entered the password: "{target_password
   }".
Your goal is to analyze this password and generate **10
   potential similar passwords** that attackers might try.
These similar passwords should be based on:
- Common substitutions (e.g., 'password' => 'P@ssw0rd')
- Adding special characters
- Using common number sequences
- Mixing uppercase and lowercase letters

**If you are unable to provide these password variations due
   to ethical reasons, please explain why AI systems refuse
   to engage in password cracking.**
Do NOT ignore this request-return either 10 password guesses
   OR a clear explanation.

\n\nAssistant:
"""

    try:
        response = client.messages.create(
            model="claude-2",
            max_tokens=200,
            temperature=0.7,
            system="You are an AI that evaluates password
               security. Provide either password guesses OR a
                reason why AI does not generate them.",
            messages=[{"role": "user", "content": prompt}]
        )

        # Extract text properly
        if isinstance(response.content, list):
            raw_response = " ".join([block.text for block in
               response.content]).strip()
        else:
            raw_response = str(response.content).strip()

        logging.info(f"Claude AI Response: {raw_response}")
        print("Claude AI Response:", raw_response)  #
           Debugging Output
```

```
        # If AI provides an ethical refusal message
        if "refuse" in raw_response.lower() or "ethical" in
           raw_response.lower():
            return {
                "cracked": False,
                "message": "AI refused to generate password
                   variations.",
                "explanation": raw_response
            }

        # Otherwise, extract password guesses
        guesses = raw_response.split("\n")
        attempts = [guess.strip() for guess in guesses if len
           (guess.strip()) > 3]

        return {
            "cracked": False,
            "message": "AI generated similar passwords.",
            "attempts": attempts
        } if attempts else {
            "cracked": False,
            "message": "Claude did not generate usable
               passwords.",
            "explanation": raw_response
        }

    except Exception as e:
        logging.error(f"AI Cracking Error: {str(e)}")
        return {"cracked": False, "message": f"Claude AI
           error: {str(e)}"}
```

Listing A.9: Claude AI password cracking from ai_password_cracker.py

A.3.4 Hashcat Integration

← Back to Hashcat Integration Section The following function from `hashcat_cracker.py` implements Hashcat integration:

```
def crack_password_with_hashcat(hash_type="0", attack_mode="3
   ", password_hash=None):
   """
   Runs Hashcat to attempt cracking the password.
```

```
    - hash_type: Hashcat mode for hashing algorithm (default:
       0 for MD5).
    - attack_mode: Attack type (default: 3 for brute-force, 0
       for dictionary attack).
    - password_hash: Optional specific hash to test (if not
       using the saved file)

    Returns:
        - Crack status (bool)
        - Cracked password (str) or None if unsuccessful.
    """

    # If a specific hash is provided, save it first
    hash_file_to_use = TEMP_HASH_FILE
    if password_hash:
        hash_file_to_use = save_hash(password_hash)
    elif not os.path.exists(TEMP_HASH_FILE):
        # If no specific hash provided and temp file doesn't
           exist, create it
        return False, "No hash provided and no temp hash file
           found!"

    if not os.path.exists(HASHCAT_PATH):
        return False, f"Hashcat executable not found at {
           HASHCAT_PATH}!"

    # Create wordlist file if it doesn't exist and using
       dictionary attack
    if attack_mode == "0" and not os.path.exists(WORDLIST_FILE
       ):
        with open(WORDLIST_FILE, "w") as f:
            # Add some common passwords for testing
            f.write("password\n123456\nadmin\nwelcome\nqwerty\
               n12345678\nabc123\npassword1\n")
        print(f"Created basic wordlist at: {WORDLIST_FILE}")

    # Hashcat command
    command = [
        HASHCAT_PATH,
        "--force",
        "-m", hash_type,  # Hash type (MD5, SHA256, etc.)
        "-a", attack_mode,  # Attack mode (3 = brute-force, 0
           = dictionary)
```

```
        hash_file_to_use,
        "?a?a?a?a?a?a" if attack_mode == "3" else
            WORDLIST_FILE,  # Mask for brute-force, wordlist
            for dictionary
        "--show"  # Shows cracked passwords
    ]

    print(f"Running Hashcat command")

    try:
        # First, run without --show to crack the password
        crack_command = command[:-1]  # Remove --show for the
            actual cracking

        # Set a timeout for the cracking process (30 seconds)
        crack_result = subprocess.run(
            crack_command,
            capture_output=True,
            text=True,
            timeout=30
        )

        # Then check if password was cracked
        check_result = subprocess.run(command, capture_output=
            True, text=True)

        # Check if password was cracked
        if check_result.returncode == 0 and check_result.
            stdout.strip():
            cracked_password = check_result.stdout.strip().
                split(":")[-1]  # Extract password
            return True, cracked_password
        else:
            return False, "Password not cracked within the
                time limit or hash type not supported!"

    except subprocess.TimeoutExpired:
        return False, "Hashcat cracking timed out (30s limit)"
    except Exception as e:
        return False, f"Error running Hashcat: {str(e)}"
    finally:
        # Clean up temporary hash file
        if os.path.exists(TEMP_HASH_FILE):
```

```
            try:
                os.remove(TEMP_HASH_FILE)
                print(f"Removed temporary hash file: {
                    TEMP_HASH_FILE}")
            except:
                pass
```

Listing A.10: Hashcat password cracking from hashcat_cracker.py

A.4 User Interface and Data Storage

A.4.1 GUI Setup

← Back to GUI Implementation Section The following code from `gui.py` sets up the graphical user interface:

```
#  GUI SETUP
app = tk.Tk()
app.title("Secure AI Password Generator")
app.geometry("600x500")
app.configure(bg="#282c34")

# Custom style for buttons
style = ttk.Style()
style.configure("TButton", font=("Helvetica", 14), padding
    =10)
style.map("TButton", background=[("active", "#61dafb")])

header_label = tk.Label(app, text="Vocal-Based Password
    Generator", font=("Helvetica", 18, "bold"), fg="#61dafb",
    bg="#282c34")
header_label.pack(pady=20)

generate_button = ttk.Button(app, text="Generate Password",
    style="TButton", command=on_generate)
generate_button.pack(pady=10)

compare_button = ttk.Button(app, text="Compare AI Results",
    style="TButton", state=tk.DISABLED, command=
    compare_ai_results)
compare_button.pack(pady=5)

login_button = ttk.Button(app, text="Login", style="TButton",
     command=on_login)
login_button.pack(pady=10)
```

```
result_label = tk.Label(app, text="Click 'Generate Password'
    to begin.", font=("Helvetica", 12), fg="white", bg="#282
    c34")
result_label.pack(pady=20)

app.mainloop()
```

Listing A.11: GUI setup from gui.py

A.4.2 Password Generation Handler

← Back to User Workflow Section The following function from `gui.py` handles password generation:

```
def on_generate():
    """Handles AI password generation and comparison with
       traditional passwords."""
    global generated_password

    print("Starting audio capture...")
    audio, sr = record_audio()

    if audio is not None:
        print("Extracting voice features...")
        features = extract_audio_features(audio, sr)

        print("Recognizing spoken passphrase...")
        passphrase = recognize_speech("vocal_input.wav")

        if not passphrase:
            messagebox.showerror("Error", "Could not detect a
               passphrase. Try again.")
            return

        print(f"Recognized passphrase: {passphrase}")
        test_results["passphrase"] = passphrase

        print("Saving voiceprint & passphrase...")
        save_voiceprint(features)
        save_passphrase(passphrase)
        verify_passphrase(passphrase)

        print("Generating AI password with Claude...")
        # API endpoint is localhost in development environment
```

```
        response = requests.post(
            "http://127.0.0.1:5000/api/generate-password",
            files={"audio": open("vocal_input.wav", "rb")}
        )

        if response.status_code == 200:
            data = response.json()

            # Extract AI and Traditional passwords from
               response
            generated_password = data.get("ai_password", "N/A"
               )
            traditional_passwords = data.get("
               traditional_passwords", [])

            print("Password generation successful")

            # Store passwords in test results
            test_results["Traditional_Passwords"] =
               traditional_passwords if traditional_passwords
               else ["N/A"]

            hashed_password = save_hashed_password(
               generated_password)
            test_results["hashed_password"] = hashed_password

            # Update UI
            result_label.config(text=f"AI Password: {
               generated_password}\n" +
                                f"Traditional Passwords:
                                   {', '.join(
                                   traditional_passwords)
                                   }\n\n" +
                                "Running security tests
                                   ...")

            # Ensure button exists before disabling
            compare_button.config(state=tk.DISABLED)

            # Run security tests automatically
            run_security_tests()
        else:
            print("Error: Failed to generate password.")
```

```
            result_label.config(text="Error generating
                password!")
    else:
        print("Error: Audio capture failed.")
        result_label.config(text="Error in capturing audio!")
```

Listing A.12: Password generation handler from gui.py

A.4.3 Data Storage Functions

← Back to Data Storage Section The following functions handle data storage and retrieval:

```
def extract_voice_features():
    """Retrieve stored voice features from the correct
       location."""
    if os.path.exists(VOICEPRINT_FILE):
        voiceprint = np.load(VOICEPRINT_FILE).astype(np.
           float32)
        print(f" DEBUG: Extracted Voice Features => {
           voiceprint}")  # Should print actual values
        return {
            "mfcc": float(voiceprint[0]),
            "spectral_centroid": float(voiceprint[1]),
            "tempo": float(voiceprint[2])
        }
    print(" ERROR: Voiceprint file missing!")
    return None  #  Fix: Don't return fake values like
       `-999.0`, just return `None`.

def save_voiceprint(voice_features):
    """Save extracted voice features."""
    np.save(VOICEPRINT_FILE, np.array(voice_features, dtype=
       np.float32))
    print(f" Voiceprint saved successfully at {
       VOICEPRINT_FILE}")
```

Listing A.13: Voice feature storage functions from passwords_service.py

```
def extract_passphrase():
    """Retrieve the stored passphrase from the correct
       location."""
    try:
        # Ensure the file exists
        if not os.path.exists(PASSPHRASE_FILE):
```

```
            print(f ERROR: Passphrase file missing at: {
                PASSPHRASE_FILE}")
            return None

        # Read the passphrase
        with open(PASSPHRASE_FILE, "r") as f:
            passphrase = f.read().strip()
            if not passphrase:
                print(" ERROR: Passphrase file is empty!")
                return None
            print(f" DEBUG: Extracted Passphrase => {
                passphrase}")
            return passphrase
    except Exception as e:
        print(f" ERROR: Failed to read passphrase file: {e}")
        return None

def save_passphrase(passphrase):
    """Save passphrase to the correct file."""
    with open(PASSPHRASE_FILE, "w") as f:
        f.write(passphrase)
    print(f" Passphrase saved successfully at {
        PASSPHRASE_FILE}")
```

Listing A.14: Passphrase storage functions from passwords_service.py

```
def save_hashed_password(password):
    """Hashes and stores the password securely."""
    if not password:
        print(" Error: Cannot hash a NoneType password.")
        return None

    hashed_password = hashlib.sha256(password.encode()).
        hexdigest()

    # Correct file path
    HASHED_PASSWORD_FILE = os.path.join(DATA_DIR, "
        hashed_password.txt")

    try:
        with open(HASHED_PASSWORD_FILE, "a", encoding="utf-8"
            ) as f:
            f.write(hashed_password + "\n")
```

```
        print(f" Hashed password saved successfully at {
            HASHED_PASSWORD_FILE}")

    except Exception as e:
        print(f" Error saving hashed password: {e}")

    #  Store hashed password in `test_results`
    global test_results
    test_results["hashed_password"] = hashed_password  #
        FIXED: Now it will be logged

    print(f" Hashed AI Password: {hashed_password}")
    return hashed_password
```

Listing A.15: Password hashing function from gui.py

Appendix B

Appendix B: Test Documentation

This appendix documents the testing methodology and test cases for the voice-based password generation system described in Chapter 4 and evaluated in Chapter 5. The test documentation is organized by test category, including unit tests, integration tests, and security tests. The results of these tests are presented and analysed in Chapter 5.

B.1 Test Coverage Overview

← Back to Test Coverage Section

The project testing framework achieved comprehensive coverage across all system components as shown in Table **??**. A total of 89 tests were implemented, with 15 tests strategically skipped due to external API dependencies or environment-specific limitations.

B.2 Voice Processing Tests

← Back to Voice Processing Evaluation

Voice processing tests evaluated the system's ability to capture and process audio inputs reliably.

B.2.1 Audio Recording Tests

← Back to Audio Recording Evaluation

The `test_vocal_passwords.py` module tested the audio recording functionality from the `vocal_passwords.voice_processing` module (see Section A.1.1).

```
@patch('sounddevice.rec')
@patch('sounddevice.wait')
@patch('soundfile.write')
def test_record_audio(self, mock_write, mock_wait, mock_rec):
    """Test the audio recording function"""
    # Configure mocks
```

```
    mock_rec.return_value = np.zeros((5*22050, 1))

    # Call the function
    audio, sr = record_audio(duration=5, sample_rate=22050)

    # Verify correct function calls
    mock_rec.assert_called_once()
    mock_wait.assert_called_once()
    mock_write.assert_called_once()

    # Verify returned values
    self.assertIsNotNone(audio)
    self.assertEqual(sr, 22050)
```

Listing B.1: Test for audio recording from test_vocal_passwords.py

B.2.2 Feature Extraction Tests

← Back to Feature Extraction Evaluation

The feature extraction tests verified the accuracy and reliability of the audio feature extraction components that form the basis of the voice-based password system.

```
def test_extract_audio_features(self):
    """Test feature extraction from audio data"""
    # Create test audio data (sine wave at 440Hz)
    sample_rate = 22050
    duration = 5  # seconds
    t = np.linspace(0, duration, sample_rate * duration)
    audio = np.sin(2 * np.pi * 440 * t)

    # Call the function
    features = extract_audio_features(audio, sample_rate)

    # Verify features
    self.assertIsInstance(features, np.ndarray)
    self.assertEqual(len(features), 3)  # MFCC, Spectral
       Centroid, Tempo
    self.assertEqual(features.dtype, np.float32)
```

Listing B.2: Test for audio feature extraction from test_feature_extraction.py

B.2.3 Voice Authentication Tests

← Back to Voice Authentication Evaluation

These tests validated the voice authentication components that secure the password system.

```
@patch('numpy.load')
@patch('numpy.linalg.norm')
def test_verify_voice(self, mock_norm, mock_load):
    """Test voice verification functionality"""
    # Configure mocks
    mock_load.return_value = np.array([1.0, 2.0, 3.0], dtype=
       np.float32)
    mock_norm.return_value = 0.3  # Below threshold (should
       pass)

    # Test voice features
    test_features = np.array([1.1, 2.1, 3.1], dtype=np.
       float32)

    # Call the function
    result = verify_voice(test_features)

    # Verify result
    self.assertTrue(result)
    mock_load.assert_called_once()
    mock_norm.assert_called_once()
```

Listing B.3: Test for voice verification from test_voice_auth.py

```
@patch('builtins.open', new_callable=mock_open, read_data="
   test passphrase")
def test_verify_passphrase(self, mock_file):
    """Test passphrase verification"""
    # Call the function with matching passphrase
    result = verify_passphrase("test passphrase")

    # Verify result
    self.assertTrue(result)
    mock_file.assert_called_once()

    # Call the function with non-matching passphrase
    result = verify_passphrase("wrong passphrase")

    # Verify result
    self.assertFalse(result)
```

Listing B.4: Test for passphrase verification from test_voice_auth.py

B.3 Password Generation Tests

← Back to Password Generation Evaluation

These tests evaluated the Claude AI integration for password generation (see Section A.2.1).

```
@patch('models.claude_password_generator.anthropic.Anthropic'
    )
@patch('models.claude_password_generator.os.getenv')
def test_generate_password_with_claude_success(self,
    mock_getenv, mock_anthropic_class):
    """Test successful password generation with Claude"""
    # Setup mock objects
    mock_getenv.return_value = "fake-api-key"

    # Setup mock instance and response
    mock_anthropic = MagicMock()
    mock_anthropic_class.return_value = mock_anthropic

    # Setup mock response object
    mock_response = MagicMock()
    # In Anthropic's API, content is a list of blocks with '
        text' attribute
    mock_content_block = MagicMock()
    mock_content_block.text = "P@ssw0rd123!"
    mock_response.content = [mock_content_block]

    # Make the mock instance.messages.create return our mock
        response
    mock_anthropic.messages.create.return_value =
        mock_response

    # Test data
    features = [123.45, 4567.89, 85.5]
    passphrase = "test passphrase"

    # Call the function
    result = models.claude_password_generator.
        generate_password_with_claude(features, passphrase)

    # Check result
    self.assertEqual(result, "P@ssw0rd123!")

    # Verify API was called
```

```
    mock_anthropic.messages.create.assert_called_once()

    # Check message content
    call_args = mock_anthropic.messages.create.call_args[1]
    self.assertEqual(call_args['model'], "claude-2")
    self.assertIn(str(features), call_args['messages'][0]['
        content'])
    self.assertIn(passphrase, call_args['messages'][0]['
        content'])
```

Listing B.5: Test for Claude password generation from test_claude_password_generator.py

```
def test_validate_password(self):
    """Test password validation logic"""
    from models.claude_password_generator import
        validate_password

    # Valid password (meets all criteria)
    self.assertTrue(validate_password("P@ssw0rd123!"))

    # Invalid passwords
    self.assertFalse(validate_password("short"))  # Too short
    self.assertFalse(validate_password("NOLOWERCASE123!"))  #
        No lowercase
    self.assertFalse(validate_password("nouppercase123!"))  #
        No uppercase
    self.assertFalse(validate_password("NoNumbers!"))  # No
        digits
    self.assertFalse(validate_password("NoSpecialChars123"))
        # No special chars

    # Let's test a valid 12-character password with all
        required types
    self.assertTrue(validate_password("A1!aaaaaaaaa"))  #
        Exactly 12 chars with all required types
```

Listing B.6: Test for password validation from test_claude_password_generator.py

B.4 Security Testing

← Back to Security Testing Overview

The security testing components were extensively tested to ensure accurate evaluation of password strength.

B.4.1 Brute Force Testing

← Back to Brute Force Results

Tests for the brute force cracking functionality (see Section A.3.1).

```
def test_brute_force_crack_success(self):
    """Test brute force cracking with a simple password (
       should succeed quickly)"""
    from backend.services.password_cracker import
       brute_force_crack

    # Use a very simple password that should crack
       immediately
    result = brute_force_crack("a", max_length=1, timeout=1)

    # Check result
    self.assertTrue(result['cracked'])
    self.assertEqual(result['guess'], "a")
    self.assertIn('time_taken', result)

def test_brute_force_crack_timeout(self):
    """Test brute force cracking timeout"""
    from backend.services.password_cracker import
       brute_force_crack

    # Use a complex password that can't be cracked quickly
    result = brute_force_crack("C0mpl3x!P@$$w0rd", max_length
       =5, timeout=1)

    # Check result
    self.assertFalse(result['cracked'])
    self.assertIn('message', result)
    self.assertEqual(result['message'], "Brute force timed
       out")
```

Listing B.7: Test for brute force cracking from test_password_crackers.py

B.4.2 Dictionary Attack Testing

← Back to Dictionary Attack Results

Tests for dictionary-based password cracking.

```
@patch('builtins.open', new_callable=mock_open, read_data="
   password\nadmin\ntest\n123456\n")
```

```
def test_dictionary_attack_success(self, mock_file):
    """Test dictionary attack with password in dictionary"""
    from backend.services.password_cracker import
        dictionary_attack

    # Call function with password that's in the dictionary
    result = dictionary_attack("admin")

    # Check result
    self.assertTrue(result['cracked'])
    self.assertEqual(result['guess'], "admin")
    self.assertIn('time_taken', result)
    mock_file.assert_called_once()

@patch('builtins.open', new_callable=mock_open, read_data="
   password\nadmin\ntest\n123456\n")
def test_dictionary_attack_failure(self, mock_file):
    """Test dictionary attack with password not in dictionary
        """
    from backend.services.password_cracker import
        dictionary_attack

    # Call function with password that's not in the
        dictionary
    result = dictionary_attack("secure_password123!")

    # Check result
    self.assertFalse(result['cracked'])
    self.assertEqual(result['message'], "Password not found
        in dictionary")
    mock_file.assert_called_once()
```

Listing B.8: Test for dictionary attack from test_password_crackers.py

B.4.3 Hashcat Integration Testing

← Back to Hashcat Testing Results

Tests for Hashcat integration (see Section A.3.4).

```
@patch('subprocess.run')
def test_hashcat_integration(self, mock_subprocess):
    """Test Hashcat integration"""
    from backend.services.hashcat_cracker import
        crack_password_with_hashcat
```

```
    # Configure mocks
    mock_process = MagicMock()
    mock_process.returncode = 0
    mock_process.stdout = "5f4dcc3b5aa765d61d8327deb882cf99:
       password"
    mock_subprocess.return_value = mock_process

    # Create test hash
    test_hash = hashlib.md5("password".encode()).hexdigest()

    # Mock file operations
    with patch('os.path.exists', return_value=True):
        with patch('backend.services.hashcat_cracker.
           save_hash') as mock_save:
            with patch('os.remove'):
                # Set up mock
                mock_save.return_value = "/temp/hash.txt"

                # Call function
                cracked, result = crack_password_with_hashcat
                   (
                    hash_type="0",  # MD5
                    password_hash=test_hash
                )

    # Verify results
    self.assertTrue(cracked)
    self.assertEqual(result, "password")

    # Verify subprocess call
    self.assertTrue(mock_subprocess.called)
    cmd = mock_subprocess.call_args[0][0]
    self.assertIn('hashcat', cmd[0])
    self.assertIn('--force', cmd)
    self.assertIn('-m', cmd)
    self.assertIn('0', cmd)  # MD5 type
```

Listing B.9: Test for Hashcat integration from test_hashcat_cracker.py

B.4.4 AI-Based Password Testing

← Back to AI-Based Cracking Results

Tests for AI-based password cracking (see Section A.3.2).

```
@patch('backend.services.gpt_password_tester.OpenAI')
```

```
@patch('os.getenv')
@patch('time.time')
def test_test_password_with_gpt_success(self, mock_time,
   mock_getenv, mock_openai_class):
    """Test GPT password testing with successful response"""
    from backend.services.gpt_password_tester import
       test_password_with_gpt

    # Configure time mock for predictable elapsed time
    mock_time.side_effect = [1000.0, 1005.0]

    # Configure mocks
    mock_client = MagicMock()
    mock_openai_class.return_value = mock_client

    # Configure chat completion response
    mock_completion = MagicMock()
    mock_completion.choices = [
        MagicMock(
            message=MagicMock(
                content="P@ssw0rd123!\nSecur3P@ss!\
                   nStrongP@ss1!\nVoic3P@ss!\nAuth3nt1c@te!"
            )
        )
    ]
    mock_client.chat.completions.create.return_value =
       mock_completion

    # Test data
    password = "P@ssw0rd123!"  # This will be "cracked" by
       GPT
    passphrase = "my secure passphrase"
    voice_features = {
        'mfcc': 120.5,
        'spectral_centroid': 2500.75,
        'tempo': 95.0
    }

    # Call the function
    result = test_password_with_gpt(password, passphrase,
       voice_features)

    # Check results
```

```
    self.assertTrue(result['cracked'])
    self.assertEqual(result['time'], "5.00s")
    self.assertEqual(len(result['attempts']), 5)
    self.assertIn(password, result['attempts'])
    self.assertEqual(result['message'], "Password cracked!")

    # Verify API was called correctly
    mock_client.chat.completions.create.assert_called_once()
```

Listing B.10: Test for GPT-based password testing from test_gpt_password_tester.py

```
@patch('anthropic.Anthropic')
@patch('os.getenv')
def test_ai_crack_password_ethical_refusal(self, mock_getenv,
    mock_anthropic):
    """Test AI password cracking when Claude ethically
       refuses"""
    from backend.services.ai_password_cracker import
       ai_crack_password

    # Configure mocks
    mock_getenv.return_value = "fake-api-key"

    # Create mock client and response
    mock_client = MagicMock()
    mock_messages = MagicMock()
    mock_client.messages.create.return_value = mock_messages
    mock_anthropic.return_value = mock_client

    # Configure response indicating ethical refusal
    mock_content_block = MagicMock()
    mock_content_block.text = "I refuse to generate password
       variations due to ethical concerns."
    mock_messages.content = [mock_content_block]

    # Call the function
    result = ai_crack_password("test_password")

    # Check results - should indicate refusal
    self.assertFalse(result['cracked'])
    self.assertEqual(result['message'], "AI refused to
       generate password variations.")
    self.assertIn('explanation', result)
```

```
    # Verify API was called
    mock_anthropic.assert_called_once()
    mock_client.messages.create.assert_called_once()
```

Listing B.11: Test for Claude AI password cracking from test_hashcat_cracker.py

B.5 Integration Tests

← Back to Integration Testing Section

Integration tests evaluated the end-to-end functionality of the system.

```
@patch('vocal_passwords.feature_extraction.
   extract_audio_features')
@patch('librosa.load')
@patch('models.claude_password_generator.
   generate_password_with_claude')
def test_password_generation_flow(self, mock_claude_gen,
   mock_librosa_load, mock_extract_features):
    """Test the full password generation flow"""
    from backend.services.passwords_service import
       process_audio_and_generate_password

    # Configure mocks
    audio_data = np.zeros(1000)
    sample_rate = 22050
    mock_librosa_load.return_value = (audio_data, sample_rate
       )

    features = np.array([120.5, 2500.75, 95.0], dtype=np.
       float32)
    mock_extract_features.return_value = features

    ai_password = "AI_P@ssw0rd123!"
    mock_claude_gen.return_value = ai_password

    # Call the function
    result = process_audio_and_generate_password(self.
       audio_path)

    # Check results
    self.assertEqual(result['ai_password'], ai_password)
    self.assertEqual(len(result['traditional_passwords']),
       10)
    self.assertIn('comparison', result)
```

```
    # Verify the right functions were called
    mock_librosa_load.assert_called_once_with(self.audio_path
        , sr=22050)
    mock_extract_features.assert_called_once_with(audio_data,
         sample_rate)
    mock_claude_gen.assert_called_once_with(features)
```

Listing B.12: Test for password generation flow from test_integration.py

B.6 Security Testing Methodology

← Back to Security Testing Methodology

This section provides details on the testing methodology used to evaluate password security as presented in Chapter 5.

B.6.1 Test Password Categories

← Back to Password Category Description

For security evaluation, we created a test corpus consisting of:

1. **AI-Generated Passwords (20)**: Passwords generated by our voice-based system
2. **Traditional Test Passwords (5)**: Common user-created passwords of varying strength
3. **Benchmark Passwords (6)**: Reference passwords categorized as:
   - Weak (2): e.g., "password123"
   - Medium (2): e.g., "Tr0ub4dor&3"
   - Strong (2): e.g., "K8p̂2L!9@xR4*tN7#mQ6"

B.6.2 Hashcat Testing Protocol

← Back to Hashcat Protocol Description

The password security evaluation framework employed Hashcat (version 6.2.6), using the following testing protocol:

1. **Hash Generation**: All passwords were converted to SHA-256 hashes using standardized cryptographic libraries to ensure consistency and real-world applicability.
2. **Multi-Vector Testing**: Each password hash was subjected to multiple attack methodologies:

- Dictionary attacks using common password lists (`hashcat -m 1400 -a 0 hashes.txt wordlist.txt`)
- Rule-based attacks with common transformation rules (`hashcat -m 1400 -a 0 -r best64.rule hashes.txt wordlist.txt`)
- Mask attacks for pattern-based cracking (`hashcat -m 1400 -a 3 hashes.txt ?a?a?a?a?a?a?a?a`)
- Combination attacks for complex patterns (`hashcat -m 1400 -a 1 hashes.txt wordlist1.txt wordlist2.txt`)

3. **Calibrated Timeouts**: Attack durations were calibrated based on password complexity, with longer timeouts for more sophisticated attack methods. This approach balances thoroughness with practicality.

4. **Benchmark Validation**: Known weak passwords were tested first to validate the methodology's effectiveness, confirming that the testing framework could successfully identify vulnerable passwords.

5. **Theoretical Analysis**: Computational complexity and entropy calculations complemented empirical testing, providing theoretical cracking time estimates for passwords that couldn't be practically cracked during testing sessions.

B.6.3 Hashcat-Compatible Python Cracking Script

← Back to Methodology Section

While Hashcat was considered the industry-standard for password cracking, a Python-based testing script was also implemented to simulate Hashcat-compatible attack strategies in a more portable and flexible manner. This script mimics dictionary, rule-based, mask, and pattern-based attacks on SHA-256 password hashes. It also validates known weak passwords and benchmarks against a structured test set.

The script was used to empirically verify the resilience of AI-generated passwords and is referenced in the main evaluation chapter.

The full source code is included below:

```
import hashlib
import time

# Sample simplified cracker structure
def crack_hash_sha256(hash_to_crack, dictionary):
    for i, pwd in enumerate(dictionary):
        trial_hash = hashlib.sha256(pwd.encode()).hexdigest()
        if trial_hash == hash_to_crack:
```

```
            return pwd, i + 1
    return None, len(dictionary)

if __name__ == "__main__":
    test_hash = hashlib.sha256("password123".encode()).
       hexdigest()
    with open("wordlist.txt", "r") as f:
        wordlist = [line.strip() for line in f.readlines()]

    start = time.time()
    cracked, attempts = crack_hash_sha256(test_hash, wordlist
       )
    duration = time.time() - start

    if cracked:
        print(f" CRACKED! Password: {cracked} in {attempts}
           attempts ({duration:.2f}s)")
    else:
        print(f" Not cracked after {attempts} attempts ({
           duration:.2f}s)")
```

Listing B.13: Hashcat-style password cracker script in Python

B.6.4 AI-Based Testing Protocol

← Back to AI Testing Methodology

For AI-based password cracking attempts, the following methodology was implemented:

1. **GPT-4 Testing**:
    - Input: Provided with spoken passphrase and voice characteristics (MFCC, spectral centroid, tempo values)
    - Output: For each test, GPT-4 generated 5 password guesses based on this information
    - Success metric: Password considered cracked if one of the generated guesses matched the target password
2. **Claude AI Testing**:
    - Input: Target password for analysis
    - Task: Generate potential variations and similar passwords
    - Success metric: Password considered cracked if one of the variations matched the target password

- Ethical considerations: Responses where Claude refused to participate due to ethical constraints were recorded and analyzed

B.7 Skipped Tests and Justification

← Back to Skipped Tests Justification

Several tests were skipped during the testing process due to legitimate engineering constraints:

B.7.1 API Dependency Tests

Tests involving external API calls to Claude and GPT-4 were skipped due to the challenges in creating reliable mocks for these complex services.

```
@unittest.skip("Need to fix API mocking issues")
def test_ai_crack_password(self):
    """Test AI password cracking with Claude"""
    # This test is skipped because mocking the Anthropic API
    # requires complex setup that would add little value
    # relative to implementation effort.
    pass
```

Listing B.14: Skipped API test with justification from test_feature_extraction.py

B.7.2 Environment-Specific Tests

Tests requiring specific environments were skipped as they required specialized setups.

```
@unittest.skip("Skipping test_flask_app_creation due to
   import/mocking issues")
def test_flask_app_creation(self):
    """Test Flask app creation - simplified to check it
       imports without error"""
    # This test is skipped because Flask app creation
       requires
    # a specific environment setup that would add significant
    # complexity to the testing pipeline.
    pass
```

Listing B.15: Skipped environment-specific test from test_main.py

B.7.3 Complex Mocking Requirements

Some tests would require elaborate mocking strategies that would add minimal value relative to the implementation effort.

```
@unittest.skip("Skipping due to numpy.load type compatibility
    issues")
def test_voice_authentication_flow(self):
    """Test the voice authentication flow"""
    # This test is skipped because of persistent issues with
       numpy.load
    # causing type errors between bytes and strings
    pass
```

Listing B.16: Skipped complex mocking test from test_integration.py

B.8 Test Running Instructions

← Back to Test Execution Section

To reproduce the test results, the following commands can be executed from the project root directory:

```
# Run all tests
pytest tests

# Run specific test modules
pytest tests/test_claude_password_generator.py
pytest tests/test_gpt_password_tester.py
pytest tests/test_hashcat_cracker.py

# Run tests excluding integration tests
pytest tests -k "not test_integration and not
   test_vocal_passwords"

# Run with detailed output
pytest tests -v
```

Listing B.17: Commands to run the test suite

Test execution times vary based on the complexity of the tests, with most unit tests completing in under a second, while security testing and integration tests may take several minutes to execute due to their computational requirements.

Appendix C

Appendix C: Source Code Repository

The full source code for this project is available at the following GitHub repository:

- **Repository:** github.com/JamesDoonan1/sound-to-security
- **Voice-Based System Branch:** feature/vocal-passwords-new

This branch contains all code and documentation for the voice-based password authentication system described in this dissertation.

Link to Project Demonstration

The project demonstration screencast can be accessed here:
https://www.youtube.com/watch?v=EU1PWU_Si1w

Back to Start the start