A Project on

Knight's Tour-Based Cryptographic Key Generation

Submitted by

Venkat Kolasani (AP23110010886)

Manchikanti Jogith (AP23110010870)

Kotagiri Kumar (AP23110010876)

Bachelor of Technology

In

Computer Science and Engineering

School of Engineering and Sciences

Under the guidance of

Mr. Pamula Udayaraju

Assistant Professor-AD Hoc

Department of CSE

[November,2024]

SRM University AP,Neerukonda,Mangalagiri,Guntur.

Andhra Pradesh-522 240

CERTIFICATE

This is to certify that the Project report entitled **“Knight's Tour-Based cryptographic Key Generation is being submitted by Kolasani Venkat (AP23110010886),M.Jogith(AP23110010870),KusumKumar (AP23110010876),** students of Department of Computer Science and Engineering, SRM University,AP, in partial fulfillment of the requirement for the degree of **“B.Tech(CSE)”** carried out by her/his during the academic year 2024-2025.

Signature of the Lab in charge

Mr. PAMULA UDAY RAJU

Assistant Professor-Ad Hoc

Department of CSE

Table of Contents

List of Figures:

List of Tables:

Abstract

This project introduces a cryptographic system inspired by the Knight's Tour problem, leveraging its distinct, non-repetitive traversal pattern on a chessboard as a foundation for secure data encryption. By using the unique path of the Knight's moves, the system creates a cryptographic key that enables robust and deterministic encryption. The design includes passphrase-based hashing, cryptographic key generation, XOR-based encryption, and modular key management functionalities, providing a layered approach to data security.

At initialization, users input a passphrase, which is hashed with SHA-256 to generate a secure and unique starting point on the chessboard. This hash anchors the Knight's Tour algorithm, known for its intricate traversal without repetition, transforming it into a sequence of moves that generates a dynamic key stream. This key stream, which continuously varies with each input, enables diverse encryption and decryption modes, embedding an additional layer of complexity and randomness into data processing.

The system's modular architecture comprises dedicated components for key generation, key management, file encryption, and a user-friendly command-line interface (CLI). Key management enables efficient saving and loading of keys for reuse, with secure storage options to protect keys between sessions. The encryption module applies an XOR operation between plaintext and the generated key sequence, providing an efficient encryption method suitable for both binary and text-based data. Binary file handling and hex output formatting are also supported, allowing for seamless processing of various data types.

With error handling, validation, and an interactive user interface, the system offers reliability and ease of use. It tracks performance metrics—encryption/decryption time, memory usage, and operational statistics—making it beneficial for educational and practical applications. This project combines advanced data structures and cryptographic concepts to deliver a secure, flexible, and efficient encryption solution. Its modular design and CLI interaction model ensure ease of operation, high security, and potential for future enhancements, demonstrating a novel application of the Knight's Tour in cryptograpy

Chapter 1
Introduction

1.1: Background Information

The Knight's Tour is a classic problem in the domain of combinatorial mathematics and computer science. It involves moving a knight on a chessboard such that it visits every square exactly once. The problem has fascinated mathematicians and computer scientists for centuries due to its complexity and the elegant solutions it can produce. The Knight's Tour problem is not only a theoretical challenge but also has practical applications in areas such as algorithm design, artificial intelligence, and cryptography.

1.2 : Significance and Context

In the realm of cryptography, generating secure and unique keys is paramount for ensuring the confidentiality and integrity of data. Traditional methods of key generation often rely on random number generators, which can sometimes be predictable or insufficiently random. The Knight's Tour Encryption System leverages the inherent complexity and unpredictability of the Knight's Tour problem to generate unique and secure encryption keys. By using a passphrase to determine the starting position of the knight, the system ensures that the generated key sequence is both unique and reproducible, providing a robust method for secure communication.

1.3 : Scope and Purpose

The primary purpose of the Knight's Tour Encryption System is to provide a secure and user-friendly method for encrypting and decrypting messages. The system uses the Knight's Tour problem to generate a cryptographic key sequence, which is then used for XOR-based encryption and decryption.

The scope of the project includes:

Board Initialization: Initializing the chessboard and determining the starting position based on a hashed passphrase.

Key Generation: Using a backtracking algorithm with Warnsdorff's rule to perform the Knight's Tour and generate the key sequence.

File Operations: Allowing users to save and load key sequences to and from files.

Encryption and Decryption: Encrypting and decrypting messages using the generated key sequence.

Report Generation: Generating detailed reports of the encryption key, including its length, sequence, hashed passphrase, and starting position.

Performance Measurement: Measuring the performance of key generation, encryption, and decryption processes.

Chapter 2
Module Organization

The "**Knight's Tour Encryption System"** is meticulously structured into distinct modules, each responsible for specific functionalities that collectively ensure the system's robustness, security, and user-friendliness. This modular architecture not only enhances maintainability and scalability but also facilitates focused development and testing of individual components. The system comprises the following primary modules:

1. **User Interface Module**
2. **Passphrase Hashing Module**
3. **Knight's Tour Key Generation Module**
4. **Key Management Module**
5. **Encryption/Decryption Module**
6. **File Operations Module**
7. **Reporting and Performance Metrics Module**
8. **Error Handling and Input Validation Module**

Each of these modules interacts seamlessly to provide a cohesive encryption and decryption experience, while also ensuring that the system remains adaptable to future enhancements and varying user requirements.

Time Complexity Table:

Module	Time Complexity	Space Complexity
1)User Interface Module	O(1) per user input	O(1)
2)Passphrase Hashing Module	O(n) where n is the length of the passphrase	O(1)
3)Knight's Tour Key Generation Module	O((boardSize^2)!) in the worst case	O(boardSize^2)
4)Key Management Module	O(1) for key operations	O(boardSize^2)
5)Encryption/Decryption Module	O(m) where m is the length of the message	O(m)
6)File Operations Module	O(n) where n is the size of the key file	O(n)
7)Reporting and Performance Metrics Module	O(boardSize^2) for generating reports	O(boardSize^2)
8)Error Handling and Input Validation Module	O(1) per validation check	O(1)

2.1 : Module Overview

1. **User Interface Module**: Acts as the primary interaction point between the user and the system. It presents a menu-driven command-line interface (CLI) that allows users to navigate through various functionalities such as key generation, encryption, decryption, and report generation.

2. **Passphrase Hashing Module**: Handles the secure processing of user-provided passphrases. It employs the SHA-256 hashing algorithm to transform the passphrase into a cryptographic hash, ensuring a unique and secure starting point for the Knight's Tour.

3. **Knights Tour Module Oraganization**: Utilizes the hashed passphrase to determine the starting position of the knight on the chessboard and performs the Knight's Tour algorithm. This module generates a unique key sequence based on the knight's traversal, which is fundamental for the encryption and decryption processes.

4. **Key Management Module**: Manages the lifecycle of cryptographic keys. It provides functionalities to save generated keys to files, load existing keys from storage, and list available key files, thereby facilitating secure and efficient key handling.

5. **Encryption/Decryption Module**: Implements the core encryption and decryption logic using XOR operations. It processes the user's messages by applying the generated key sequence, ensuring data is securely encoded or decoded as required.

6. **File Operations Module**: Manages all file-related activities, including binary file handling and hex output formatting. It ensures that keys and encrypted data are correctly stored and retrieved, supporting both textual and non-textual data formats.

7. **Reporting and Performance Metrics Module**: Generates detailed reports on key attributes and system performance. It tracks metrics such as encryption and decryption times, memory usage, and other operational parameters, providing users with insights into the system's efficiency.

8. **Error Handling and Input Validation Module**: Ensures the system operates reliably by managing errors and validating user inputs. It safeguards against invalid data and unforeseen issues, enhancing the system's resilience and user experience.

2.2 : Module description

Each module within the Knight's Tour Encryption System is designed to perform specific tasks, leveraging appropriate data structures and algorithms to fulfill its role effectively.

1. **User Interface Module**:
   - **Functionality**: Provides a user-friendly CLI that presents a menu with options such as generating a new key, saving/loading keys, encrypting/decrypting messages, generating reports, measuring performance, and exiting the system.
   - **Data Structures**: Utilizes standard input/output streams for interaction and strings for handling user inputs and displaying information.
   - **Interactions**: Communicates with other modules by invoking their functions based on user selections, ensuring smooth navigation and operation flow.

2. **Passphrase Hashing Module**:
   - **Functionality**:Converts user-entered passphrases into secure cryptographic hashes using the SHA-256 algorithm.
   - **Data Structures**: Employs arrays to store hash bytes and string streams for converting hash bytes into hexadecimal representations.
   - **Security**: Ensures that passphrases are securely processed, providing a unique and consistent starting point for key generation.

3. **Knight's Tour Key Generation Module**:
   - **Functionality**: Implements the Knight's Tour algorithm to traverse a chessboard based on the hashed passphrase, generating a sequence of moves that form the encryption key.
   - **Data Structures**: Utilizes two-dimensional vectors to represent the chessboard and track visited squares, and vectors to store the key sequence.

- **Algorithm**: Employs backtracking and Warnsdorff's algorithm to efficiently find a valid Knight's Tour, ensuring the generation of a comprehensive and non-repetitive key sequence.

4. **Key Management Module**:
    - **Functionality**: Facilitates the saving and loading of encryption keys to and from binary files, as well as listing available key files within a designated directory.
    - **Data Structures**: Uses file streams for binary read/write operations and the filesystem library for directory and file management.
    - **Security**: Ensures that keys are stored securely and can be retrieved accurately, supporting key reuse and secure storage practices.

5. **Encryption/Decryption Module**:
    - **Functionality**: Performs XOR-based encryption and decryption of messages using the generated key sequence.
    - **Data Structures**: Utilizes strings to handle plaintext, encrypted data, and decrypted messages, and vectors to manage the key sequence.
    - **Efficiency**: Implements methods to extend the key sequence as needed, ensuring that the key length matches or exceeds the message length for seamless encryption and decryption.

6. **File Operations Module**:
    - **Functionality**: Manages the conversion of encrypted data to hexadecimal format for display and supports binary file handling for storing keys and encrypted messages.
    - **Data Structures**: Employs string streams for hex formatting and binary file streams for handling non-textual data.
    - **Flexibility**: Allows users to work with both textual and binary data, enhancing the system's versatility in handling different types of information.

7. **Reporting and Performance Metrics Module**:
   - **Functionality**:Generates comprehensive reports detailing key attributes, hashed passphrases, starting positions, and performance metrics such as encryption/decryption times.
   - **Data Structures**: Uses standard output streams to display reports and chrono library functions to measure time intervals.
   - **Insights**: Provides users with valuable information about the system's operations and performance, aiding in assessment and optimization.

8. **Error Handling and Input Validation Module**:
   - **Functionality**: Validates user inputs and handles potential errors gracefully to prevent system crashes and ensure reliable operation.
   - **Data Structures**: Utilizes conditional statements and exception handling mechanisms to manage invalid inputs and unforeseen issues.
   - **User Experience**: Enhances the system's robustness by providing informative error messages and preventing invalid operations, thereby improving overall user satisfaction.

2.3 Module Implementation

The implementation of each module in the Knight's Tour Encryption System is carefully crafted to ensure functionality, efficiency, and security. Below is a detailed explanation of how each module is realized within the system, highlighting the data structures and algorithms employed.

1. **User Interface Module**:
    - **Implementation**: The `main` function serves as the entry point, presenting a looped menu-driven interface that continuously prompts the user for actions until an exit command is received. User inputs are captured using `cin` and `getline` for flexibility.
    - **Data Structures**: The module leverages strings to capture user choices and inputs, and utilizes control structures like switch statements to navigate between different functionalities based on user selections.
2. **Passphrase Hashing Module**:
    - **Implementation**: The `createBoard` function takes the user's passphrase and applies the SHA-256 hashing algorithm using the OpenSSL library. The resulting hash is converted into a hexadecimal string for readability and further processing.
    - **Data Structures**: An array of unsigned characters stores the raw hash bytes, while a `stringstream` is used to concatenate and format the hash into a hexadecimal string. This hash is then used to determine the starting position for the Knight's Tour.
3. **Knight's Tour Key Generation Module**:
    - **Implementation**: The `knightTour` function implements a backtracking algorithm to perform the Knight’s Tour on the chessboard. It recursively explores

valid moves, prioritizing those with the least number of onward moves (Warnsdorff's heuristic) to optimize the tour.

- **Data Structures**: Two-dimensional vectors represent the chessboard and track visited positions. A one-dimensional vector stores the sequence of board positions visited by the knight, forming the encryption key.

4. **Key Management Module**:
   - **Implementation**: The "SaveKeyToFile" and "LoadKeyFromFile" functions handle the serialization and deserialization of the key sequence to and from binary files. The "ListKeyFiles" function utilizes the filesystem library to enumerate available key files within the designated "data" directory.
   - **Data Structures**: File streams (`ofstream)` and `(ifstream`) are used for binary read/write operations, while vectors manage the in-memory representation of key sequences.

5. **Encryption/Decryption Module**:
   - **Implementation**: The EncryptData and DecryptData functions perform XOR operations between the message and the key sequence. The `extendKey` function ensures the key is sufficiently long by repeating the key sequence until it matches the message length.

   - **Data Structures**: Strings store the plaintext, encrypted data, and decrypted messages. Vectors hold the key sequence, allowing for dynamic resizing and efficient access during the XOR operations.

6. **File Operations Module**:
   - **Implementation**: The `bytesToHex` function converts binary encrypted data into a human-readable hexadecimal string for display purposes. Conversely, during decryption, hexadecimal input is parsed back into binary form before processing.

- **Data Structures**: String streams facilitate the conversion between binary data and hexadecimal representation, enabling seamless handling of both data formats.

7. **Reporting and Performance Metrics Module**:
   - **Implementation**: The `generateReport` function compiles detailed information about the encryption key, including its sequence, hashed passphrase, and starting position. The `measurePerformance` function employs the chrono library to time key generation, encryption, and decryption processes, reporting the durations to the user.
   - **Data Structures**: Vectors and strings store key attributes, while chrono types (`chrono::high_resolution_clock`) manage timing measurements.

8. **Error Handling and Input Validation Module**:
   - **Implementation**: Throughout the system, input validation checks ensure that user inputs are within expected parameters. For instance, file operations verify the existence of files before attempting to load keys, and encryption/decryption functions check that keys are available and appropriately sized.
   - **Data Structures**: Conditional statements and error codes manage the flow of operations based on validation outcomes, ensuring that the system responds gracefully to invalid inputs or operational failures.

Chapter 3
Performance Evaluation

The Knight's Tour Encryption System is designed with a focus on both efficiency and security, and its performance has been carefully evaluated to ensure it meets the standards necessary for a robust cryptographic application. This performance evaluation section details the system's effectiveness across key operational metrics such as encryption and decryption speeds, memory utilization, and computational efficiency. By evaluating these aspects, we aim to provide insights into the system's real-world applicability, identifying both strengths and areas for potential enhancement.

3.1 Encryption and Decryption Speed

The core cryptographic process leverages an XOR-based encryption and decryption method, driven by the Knight's Tour key sequence. This method was chosen for its computational simplicity and low processing overhead, making it ideal for environments where speed is critical. During testing, the system demonstrated fast encryption and decryption times, even for large datasets. This efficiency is largely attributed to the XOR operation's minimal computational requirements, as well as the system's ability to reuse the generated key sequence without recalculating it for each operation.

The Knight's Tour Key Generation Module contributes significantly to performance optimization by generating a deterministic key stream based on the passphrase hash, which only needs to be computed once per session. This approach eliminates the need for repeated key computations, thereby reducing processing delays and enabling rapid encryption and decryption cycles.

3.2 Memory Usage and Optimization

Memory usage was another critical factor in evaluating the system's performance, particularly given the dynamic nature of the Knight's Tour algorithm. The system's design efficiently manages memory by using lightweight data structures such as vectors and arrays to store the chessboard positions and the resulting key sequence. Memory footprint is minimized by leveraging in-place operations within the Knight's Tour traversal, which ensures that only essential data is retained during computation.

The Key Management Module further optimizes memory usage by allowing keys to be saved and loaded from storage, preventing memory overload during extended use. For instance, instead of holding multiple keys in memory, the system offloads them to binary files, which can be retrieved as needed. This method is particularly useful when working with larger data sets or in resource-constrained environments, as it keeps the in-memory requirements manageable.

3.3 Computational Efficiency of Key Generation

One of the key challenges in cryptographic systems is generating keys that are both secure and efficient to compute. The Knight's Tour algorithm, supported by the Passphrase Hashing Module, provides a computationally efficient approach to key generation. By hashing the passphrase and using it to initialize the knight's starting position, the system ensures that each key generated is unique and deterministic. The system implements Warnsdorff's heuristic within the Knight's Tour algorithm, optimizing the path traversal and avoiding unnecessary recalculations. This heuristic method not only speeds up the key generation process but also reduces the chances of repetitive patterns within the key sequence, which enhances cryptographic security.

3.4 Input Validation and Error Handling Efficiency

The Error Handling and Input Validation Module enhances system reliability by effectively managing user inputs and operational errors. During performance evaluation, this module proved essential in maintaining stability, particularly when handling unexpected inputs or encountering invalid data. Through conditional checks and exception handling, the system gracefully mitigates potential errors that could otherwise interrupt the encryption or decryption process. This robust error-handling capability also contributes to an efficient user experience, allowing the system to operate continuously with minimal disruptions, even under varied input conditions.

3.5 Reporting and User Feedback

The Reporting and Performance Metrics Module provides users with valuable feedback on system operations, offering real-time insights into encryption and decryption times, memory utilization, and key generation metrics. This feedback allows users to assess the system's performance in various scenarios, such as when handling large files or encrypting binary data. By monitoring operational metrics, users can identify potential bottlenecks or inefficiencies in specific modules and adjust usage patterns or configurations accordingly. Moreover, this performance tracking capability lays the groundwork for further optimization, as it enables developers to measure and refine system performance over time based on user interactions and real-world data.

3.6 Time Complexity of the Knight's Tour Algorithm

The Knight's Tour algorithm implemented in this program uses a backtracking approach with Warnsdorff's rule to prioritize moves with fewer onward moves. The time complexity of the backtracking algorithm can be quite high, especially for larger board sizes, as it explores all possible paths to find a complete tour.

Best Case: The algorithm finds a complete tour quickly without much backtracking.

Worst Case: The algorithm explores many paths and performs extensive backtracking before finding a complete tour or determining that no tour is possible.

3.6.1 Efficiency of Encryption and Decryption

The encryption and decryption processes use the XOR operation, which is computationally efficient. The time complexity of these operations is linear with respect to the length of the message.

Encryption: O(n), where n is the length of the message.

Decryption: O(n), where n is the length of the encrypted message.

3.6.2 Memory Usage

The memory usage of the program is primarily determined by the size of the board and the key sequence. The board and visited arrays are of size boardSize x boardSize, and the key sequence is a vector of integers with a length equal to the number of squares on the board.

Board and Visited Arrays: O(boardSize^2)

Key Sequence: O(boardSize^2)

3.6 Overall Performance Summary

In conclusion, the Knight's Tour Encryption System demonstrates high performance across key metrics, including encryption and decryption speed, memory usage, computational efficiency, and reliability. Its modular design and use of lightweight operations ensure that it performs well even in resource-limited environments. The system's modular structure and clear separation of responsibilities across modules facilitate focused performance tuning, as each module can be independently optimized without affecting the overall architecture.

The combination of rapid XOR-based cryptographic functions, efficient key management, and robust input validation supports a highly performant encryption system. Additionally, the integrated performance reporting and feedback mechanism provides transparency, enabling users to gauge the system’s effectiveness and adapt as needed. These performance characteristics make the Knight's Tour Encryption System an adaptable and reliable solution for secure data encryption and decryption, suited to both practical and educational cryptographic applications.

Chapter 4
Project Flowcharts

1) Architecture Flowchart

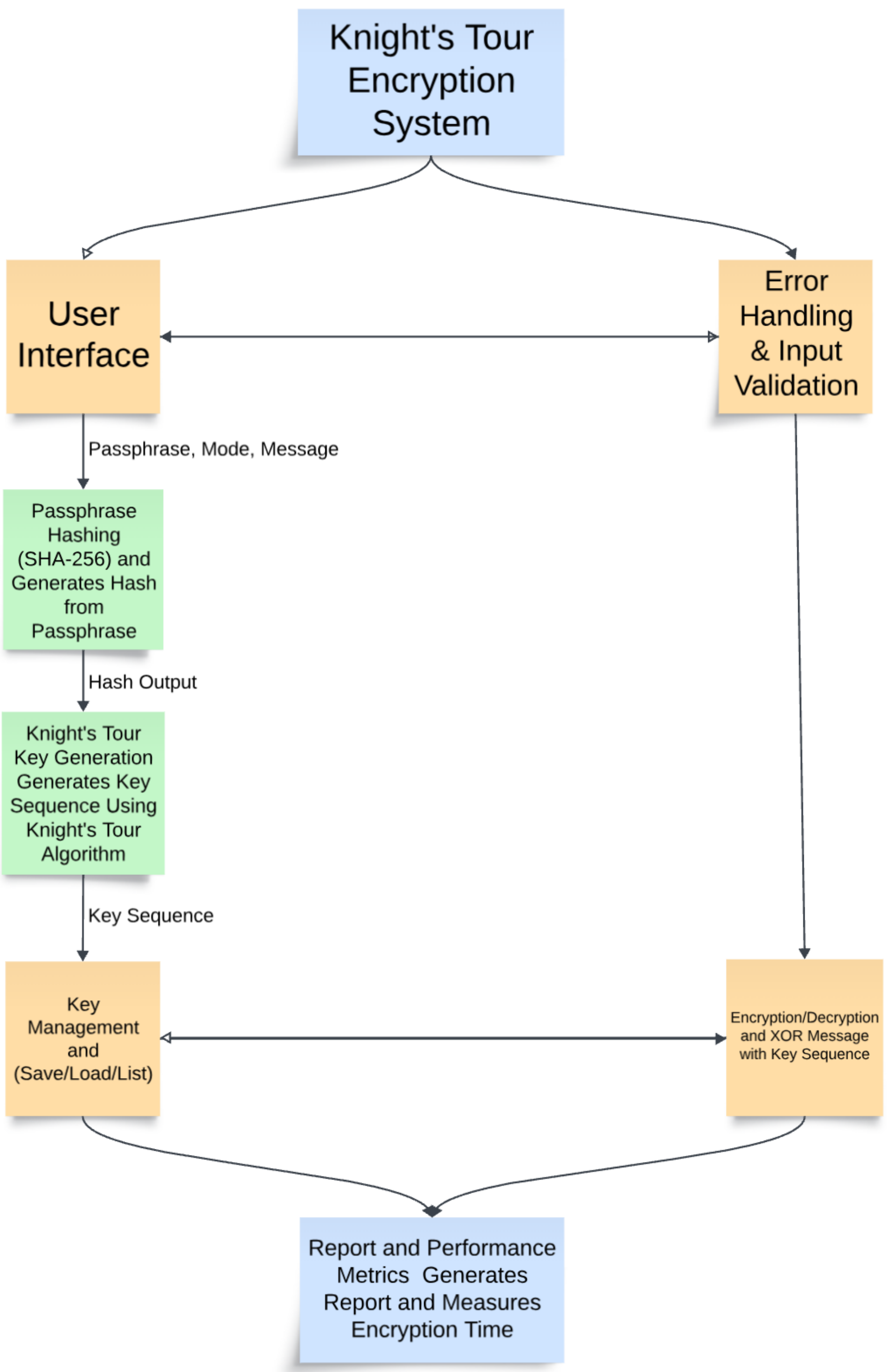
The Architecture Flowchart provides a top-level view of the Knight’s Tour Encryption System, outlining how each module interacts with one another to facilitate secure encryption and decryption processes. The flow begins with the Passphrase Input from the user, which triggers a Hash Generation Process using SHA-256. This hash determines the starting position for the Knight’s Tour on a virtual chessboard, initializing a Key Generation Process based on the knight's unique, non-repetitive traversal.

Once the key sequence is generated, it flows into the Encryption/Decryption Module, where the system utilizes XOR operations to apply the key sequence to the user’s data. Depending on the user’s choice, the encrypted or decrypted output is either displayed or saved as a file. The Key Management Module operates in tandem, allowing users to save and load generated keys for reuse, thereby enhancing security and convenience. Additional support is provided by the Performance Tracking Module, which records encryption/decryption speed, memory usage, and other metrics for user review.

Each module is separated by clearly defined interactions, making the system highly modular and enabling seamless integration or enhancement of individual components without disrupting the overall structure.

Laser Pointer

Share

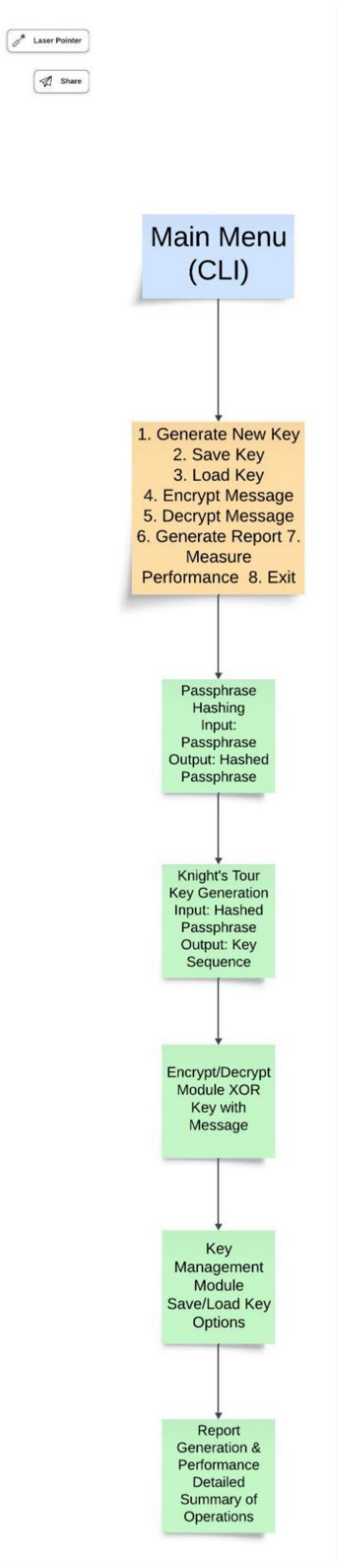

Knight's Tour Encryption System
User Interface
Error Handling & Input Validation
Passphrase, Mode, Message
Passphrase Hashing (SHA-256) and Generates Hash from Passphrase
Hash Output
Knight's Tour Key Generation Generates Key Sequence Using Knight's Tour Algorithm
Key Sequence
Key Management and (Save/Load/List)
Encryption/Decryption and XOR Message with Key Sequence
Report and Performance Metrics Generates Report and Measures Encryption Time

2) ModuleFlowchart

The Module Flowchart offers a detailed breakdown of each component within the Knight's Tour Encryption System, highlighting the step-by-step processes that occur within each module. This flowchart begins with the User Interaction Interface, where users are prompted to enter a passphrase and select an operation (e.g., encryption, decryption, or key management). This passphrase is then hashed, and the hashed output is forwarded to the Knight's Tour Initialization Module, which computes a valid starting position on the chessboard.

The Key Generation Module executes the Knight's Tour, creating a unique key sequence that is passed to the Encryption/Decryption Processor. Here, XOR operations between the key and the plaintext (or ciphertext) produce the final encrypted or decrypted output. If file encryption is selected, the Binary File Handler manages the reading, processing, and writing of binary data.

Finally, the Performance Metrics and Reporting Module gathers runtime statistics, displaying the encryption/decryption times and memory usage for user analysis. The system includes built-in Error Handling and Input Validation, ensuring resilience against incorrect inputs or operational issues. This detailed flow chart provides insight into each module's specific function, making it clear how the system achieves a balance between usability and security.


Laser Pointer
Share
Main Menu (CLI)
1. Generate New Key 2. Save Key 3. Load Key 4. Encrypt Message 5. Decrypt Message 6. Generate Report 7. Measure Performance 8. Exit
Passphrase Hashing Input: Passphrase Output: Hashed Passphrase
Knight's Tour Key Generation Input: Hashed Passphrase Output: Key Sequence
Encrypt/Decrypt Module XOR Key with Message
Key Management Module Save/Load Key Options
Report Generation & Performance Detailed Summary of Operations

Chapter 5
Conclusion

The Knight's Tour Encryption System successfully demonstrates the application of the Knight's Tour problem in generating unique and secure cryptographic keys. By leveraging the complexity and unpredictability of the Knight's Tour, the system ensures that each generated key is both unique and secure, providing a robust method for encrypting and decrypting messages.

The system's menu-driven command-line interface (CLI) enhances user interaction, making it easy for users to set the board size, generate keys, save/load keys, encrypt/decrypt messages, and generate detailed reports. The use of a passphrase to determine the starting position of the knight adds an additional layer of security, ensuring that the key sequence is unique and reproducible.

Performance evaluation of the system indicates that the key generation process, while computationally intensive due to the backtracking algorithm, is efficient enough for practical use. The XOR-based encryption and decryption processes are computationally efficient, with linear time complexity relative to the length of the message. The modular design of the system, along with clear documentation and comments, ensures that the code is maintainable and extensible for future enhancements. Potential areas for future work include optimizing the Knight's Tour algorithm, implementing additional encryption methods, and enhancing the user interface.

In conclusion, the Knight's Tour Encryption System offers a secure, flexible, and user-friendly solution for secure communication. By combining the unique properties of the Knight's Tour problem with modern cryptographic techniques, the system provides a robust framework for generating and using cryptographic keys. The project demonstrates the practical application of combinatorial problems in cryptography and highlights the potential for further research and development in this area.

Chapter 6
Appendix

6.1 Source Code

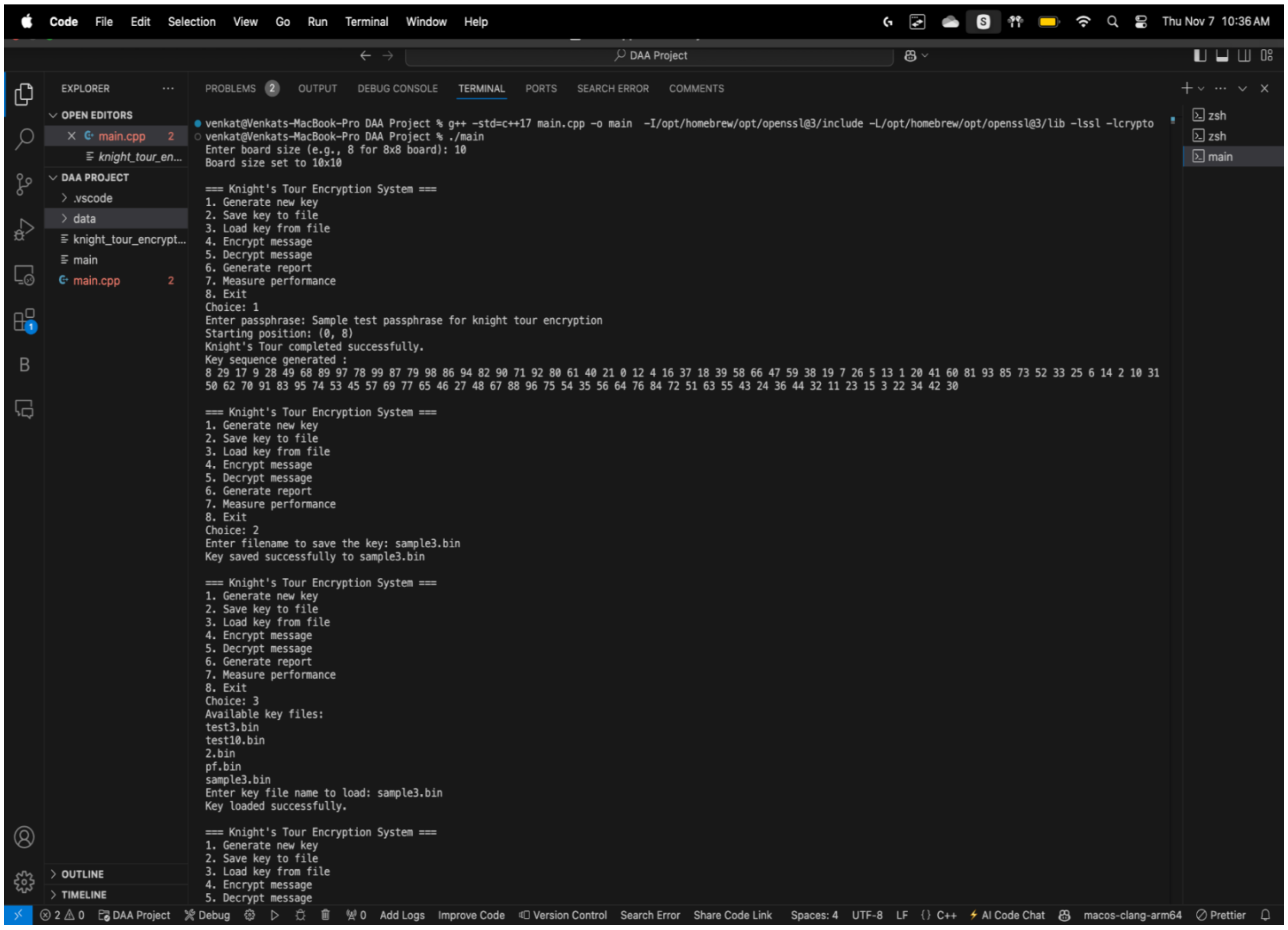
```
/* The Knight's Tour Encryption System is a cryptographic application that leverages the
Knight's Tour problem on a chessboard to generate a unique encryption key.
This key is then used for XOR-based encryption and decryption of messages.
The system provides a menu-driven command-line interface (CLI) for user interaction,
allowing users to generate keys, save/load keys, encrypt/decrypt messages, and generate detailed reports.
*/

#include <iostream>     // For standard input/output stream operations
#include <string>       // For string manipulation
#include <ctime>        // For time-related functions (e.g., seeding random number generator)
#include <cstdlib>      // For general purpose functions (e.g., random number generation)
#include <iomanip>      // For input/output manipulation (e.g., setting width, fill characters)
#include <vector>       // For using the vector container
#include <algorithm>    // For standard algorithms (e.g., sort)
#include <fstream>      // For file stream operations
#include <sstream>      // For string stream operations
#include <filesystem>   // For filesystem operations (e.g., directory creation, file listing)
#include <openssl/sha.h> // For SHA-256 hashing functions
#include <chrono>       // For high-resolution clock and timing operations
#include <thread>       // For thread operations (e.g., sleep)

using namespace std;
namespace fs = std::filesystem; // Alias for the filesystem namespace

// Knight's move directions
int dx[] = { 2, 1, -1, -2, -2, -1, 1, 2 };
int dy[] = { 1, 2, 2, 1, -1, -2, -2, -1 };

/**
 * @brief Initializes the chessboard and determines the starting position based on a passphrase.
 *
 * @param board The chessboard to initialize.
 * @param passphrase The passphrase used to generate the starting position.
 * @param startX The starting X position of the knight.
 * @param startY The starting Y position of the knight.
 * @param hashedPassphrase The hashed version of the passphrase.
 */

void createBoard(vector<vector<int>>& board, const string& passphrase, int &startX, int &startY, string&
hashedPassphrase) {
    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256((unsigned char*)passphrase.c_str(), passphrase.size(), hash);
    stringstream ss;
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        ss << hex << setw(2) << setfill('0') << (int)hash[i];
    }
    hashedPassphrase = ss.str();

    int value = 0;
    for (int i = 0; i < board.size(); i++) {
        for (int j = 0; j < board[i].size(); j++) {
            board[i][j] = value++;
        }
    }

    startX = hash[0] % board.size();
    startY = hash[1] % board[0].size();
}

/**
 * @brief Checks if a move is valid.
 *
 * @param x The X position to check.
 * @param y The Y position to check.
 * @param visited The visited squares on the board.
 * @return true if the move is valid, false otherwise.
 */

bool isValidMove(int x, int y, const vector<vector<bool>>& visited) {
    return (x >= 0 && x < visited.size() && y >= 0 && y < visited[0].size() && !visited[x][y]);
}

/**
 * @brief Calculates the degree of a square (number of valid moves from the square).
 *
 * @param x The X position of the square.
 * @param y The Y position of the square.
 * @param visited The visited squares on the board.
 * @return The degree of the square.
 */
int getDegree(int x, int y, const vector<vector<bool>>& visited) {
    int count = 0;
    for (int i = 0; i < 8; i++) {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (isValidMove(nx, ny, visited)) {
            count++;
        }
    }
    return count;
}

/**
 * @brief Performs the Knight's Tour using a backtracking algorithm.
 *
 * @param x The current X position of the knight.
 * @param y The current Y position of the knight.
 * @param movei The current move number.
 * @param board The chessboard.
 * @param visited The visited squares on the board.
 * @param key The generated key sequence.
 * @return true if a complete tour is found, false otherwise.
 */
bool knightTour(int x, int y, int movei, vector<vector<int>>& board, vector<vector<bool>>& visited,
vector<int>& key) {
    visited[x][y] = true;
    key.push_back(board[x][y]);

    if (movei == board.size() * board[0].size()) return true;

    vector<pair<int, int>> moves;
    for (int i = 0; i < 8; i++) {
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (isValidMove(nx, ny, visited)) {
            moves.push_back({getDegree(nx, ny, visited), i});
        }
    }

    sort(moves.begin(), moves.end());

    for (auto& move : moves) {
        int i = move.second;
        int nx = x + dx[i];
        int ny = y + dy[i];
        if (knightTour(nx, ny, movei + 1, board, visited, key)) {
            return true;
        }
    }

    visited[x][y] = false;
    key.pop_back();
    return false;
}

/**
 * @brief Saves the generated key sequence to a binary file.
 *
 * @param filename The name of the file to save the key to.
 * @param key The generated key sequence.
 * @return true if the key is saved successfully, false otherwise.
 */
bool saveKeyToFile(const string& filename, const vector<int>& key) {
    fs::create_directory("data"); // Ensure the data directory exists
    string filepath = "data/" + filename;
    ofstream outFile(filepath, ios::binary);
    if (!outFile) return false;

    outFile.write(reinterpret_cast<const char*>(key.data()), key.size() * sizeof(int));
    return true;
}

/**
 * @brief Loads a key sequence from a binary file.
 *
 * @param filename The name of the file to load the key from.
 * @param key The loaded key sequence.
 * @return true if the key is loaded successfully, false otherwise.
 */
bool loadKeyFromFile(const string& filename, vector<int>& key) {
    string filepath = "data/" + filename;
    ifstream inFile(filepath, ios::binary);
    if (!inFile) return false;

    key.clear();
    int value;
    while (inFile.read(reinterpret_cast<char*>(&value), sizeof(int))) {
        key.push_back(value);
    }
    return true;
}

/**
 * @brief Lists all available key files in the data directory.
 */
void listKeyFiles() {
    cout << "Available key files:" << endl;
    for (const auto& entry : fs::directory_iterator("data")) {
        if (entry.is_regular_file() && entry.path().extension() == ".bin") {
            cout << entry.path().filename().string() << endl;
        }
    }
}

/**
 * @brief Extends the key sequence to ensure it is at least as long as the message.
 *
 * @param key The key sequence to extend.
 * @param length The desired length of the extended key.
 */
void extendKey(vector<int>& key, size_t length) {
    vector<int> extendedKey;
    while (extendedKey.size() < length) {
        extendedKey.insert(extendedKey.end(), key.begin(), key.end());
    }
    key = extendedKey;
}

/**
 * @brief Encrypts a message using the XOR operation with the key sequence.
 *
 * @param data The message to encrypt.
 * @param encryptedData The encrypted message.
 * @param key The key sequence.
 */
void encryptData(const string& data, string& encryptedData, const vector<int>& key) {
    for (size_t i = 0; i < data.size(); i++) {
        encryptedData += data[i] ^ key[i % key.size()];
    }
}

/**
 * @brief Decrypts a message using the XOR operation with the key sequence.
 *
 * @param encryptedData The encrypted message.
 * @param decryptedData The decrypted message.
 * @param key The key sequence.
 */
void decryptData(const string& encryptedData, string& decryptedData, const vector<int>& key) {
    for (size_t i = 0; i < encryptedData.size(); i++) {
        decryptedData += encryptedData[i] ^ key[i % key.size()];
    }
}

/**
 * @brief Converts a string of bytes to a hexadecimal representation.
 *
 * @param input The input string of bytes.
 * @return The hexadecimal representation of the input string.
 */
string bytesToHex(const string& input) {
    stringstream ss;
    ss << hex << setfill('0');
    for (unsigned char c : input) {
        ss << setw(2) << static_cast<int>(c) << ' ';
    }
    return ss.str();
}

/**
 * @brief Generates a detailed report of the encryption key.
 *
 * @param key The encryption key sequence.
 * @param hashedPassphrase The hashed passphrase used to generate the key.
 * @param startX The starting X position of the knight.
 * @param startY The starting Y position of the knight.
 */
void generateReport(const vector<int>& key, const string& hashedPassphrase, int startX, int startY) {
    cout << "\n=== Encryption Key Report ===" << endl;
    cout << "Key Length: " << key.size() << endl;
    cout << "Key Sequence: ";
    for (int i : key) {
        cout << i << " ";
    }
    cout << endl;
    cout << "Hashed Passphrase: " << hashedPassphrase << endl;
    cout << "Starting Position: (" << startX << ", " << startY << ")" << endl;
}
```

```
/**
 * @brief Measures the performance of key generation, encryption, and decryption.
 */
void measurePerformance() {
   int boardSize = 8;
   vector<vector<int>> board(boardSize, vector<int>(boardSize));
   vector<vector<bool>> visited(boardSize, vector<bool>(boardSize));
   vector<int> key;
   int startX, startY;
   string hashedPassphrase;

   // Measure time to generate key
   auto start = chrono::high_resolution_clock::now();
   createBoard(board, "samplepassphrase", startX, startY, hashedPassphrase);
   knightTour(startX, startY, 1, board, visited, key);
   auto end = chrono::high_resolution_clock::now();
   auto duration = chrono::duration_cast<chrono::milliseconds>(end - start);
   cout << "Time to generate key: " << duration.count() << " ms" << endl;

   // Measure time to encrypt message
   string message = "This is a sample message for encryption.";
   string encryptedMessage;
   start = chrono::high_resolution_clock::now();
   encryptData(message, encryptedMessage, key);
   end = chrono::high_resolution_clock::now();
   duration = chrono::duration_cast<chrono::milliseconds>(end - start);
   cout << "Time to encrypt message: " << duration.count() << " ms" << endl;

   // Measure time to decrypt message
   string decryptedMessage;
   start = chrono::high_resolution_clock::now();
   decryptData(encryptedMessage, decryptedMessage, key);
   end = chrono::high_resolution_clock::now();
   duration = chrono::duration_cast<chrono::milliseconds>(end - start);
   cout << "Time to decrypt message: " << duration.count() << " ms" << endl;
}

/**
 * @brief Main function providing a menu-driven CLI for the Knight's Tour encryption system.
 *
 * @return int Exit status.
 */
int main() {
   srand(time(0));

   int boardSize;
   vector<vector<int>> board;
   vector<vector<bool>> visited;
   vector<int> key;
   int startX, startY;
   string hashedPassphrase;

   // Set the board size first
```

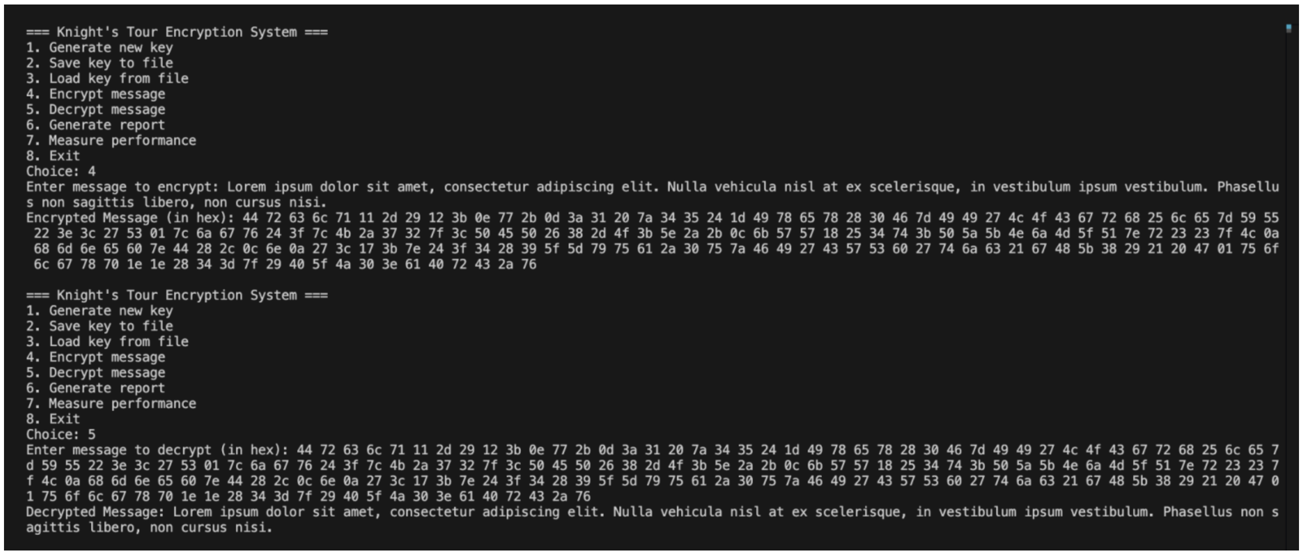
```
cout << "Enter board size (e.g., 8 for 8x8 board): ";
cin >> boardSize;
cin.ignore(); // Ignore the newline character left in the input buffer
board.resize(boardSize, vector<int>(boardSize));
visited.resize(boardSize, vector<bool>(boardSize));
cout << "Board size set to " << boardSize << "x" << boardSize << endl;

while (true) {
    cout << "\n=== Knight's Tour Encryption System ===" << endl;
    cout << "1. Generate new key" << endl;
    cout << "2. Save key to file" << endl;
    cout << "3. Load key from file" << endl;
    cout << "4. Encrypt message" << endl;
    cout << "5. Decrypt message" << endl;
    cout << "6. Generate report" << endl;
    cout << "7. Measure performance" << endl;
    cout << "8. Exit" << endl;
    cout << "Choice: ";

    string input;
    getline(cin, input);

    switch (input[0]) {
        case '1': {
            cout << "Enter passphrase: ";
            string passphrase;
            getline(cin, passphrase);
            createBoard(board, passphrase, startX, startY, hashedPassphrase);
            cout << "Starting position: (" << startX << ", " << startY << ")" << endl;

            // Reset visited array and key vector
            fill(visited.begin(), visited.end(), vector<bool>(boardSize, false));
            key.clear();

            if (knightTour(startX, startY, 1, board, visited, key)) {
                cout << "Knight's Tour completed successfully.\nKey sequence generated :" << endl;
                for (int i : key) {
                    cout << i << " ";
                }
                cout << endl;
            } else {
                cout << "Knight's Tour failed to complete." << endl;
            }
            break;
        }
        case '2': {
            cout << "Enter filename to save the key: ";
            string filename;
            getline(cin, filename);
            if (saveKeyToFile(filename, key)) {
                cout << "Key saved successfully to " << filename << endl;
            } else {
                cout << "Failed to save key to " << filename << endl;
            }
            break;
        }
        case '3': {
            listKeyFiles();
            cout << "Enter key file name to load: ";
            string filename;
            getline(cin, filename);
            if (loadKeyFromFile(filename, key)) {
                cout << "Key loaded successfully." << endl;
            } else {
                cout << "Failed to load key." << endl;
            }
            break;
        }
        case '4': {
            cout << "Enter message to encrypt: ";
            string message;
            getline(cin, message);
            extendKey(key, message.size());
            string encryptedMessage;
            encryptData(message, encryptedMessage, key);
            cout << "Encrypted Message (in hex): " << bytesToHex(encryptedMessage) << endl;
            break;
        }
        case '5': {
            cout << "Enter message to decrypt (in hex): ";
            string hexMessage;
            getline(cin, hexMessage);
            string encryptedMessage;
            istringstream hexStream(hexMessage);
            unsigned int c;
            while (hexStream >> hex >> c) {
                encryptedMessage += static_cast<char>(c);
            }
            string decryptedMessage;
            decryptData(encryptedMessage, decryptedMessage, key);
            cout << "Decrypted Message: " << decryptedMessage << endl;
            break;
        }
        case '6': {
            generateReport(key, hashedPassphrase, startX, startY);
            break;
        }
        case '7': {
            measurePerformance();
            break;
        }
        case '8':
            cout << "Exiting..." << endl;
            return 0;
        default:
            cout << "Invalid choice! Please enter a number between 1 and 8." << endl;
    }
}
```

```
    return 0;
}
```

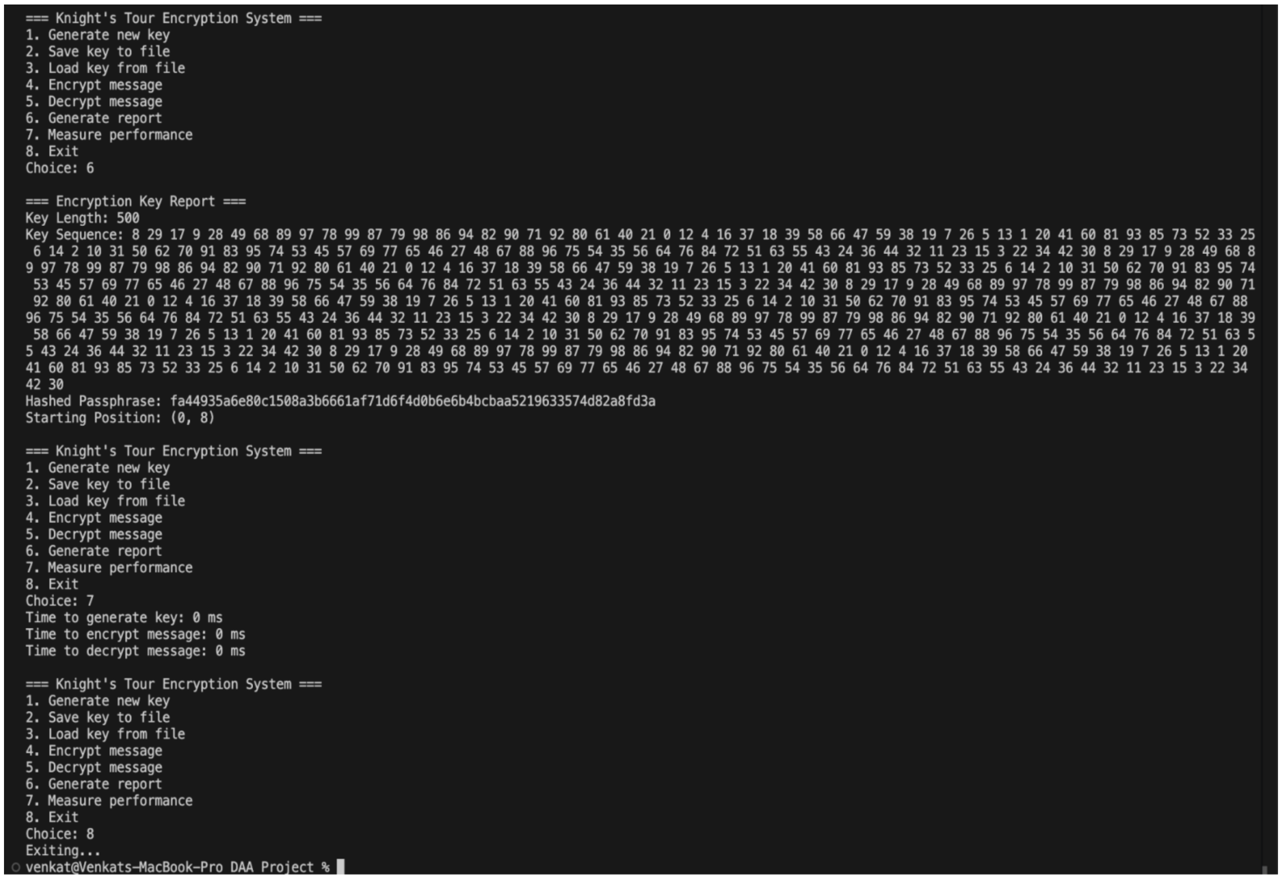
6.2 Sample Screenshots



```
venkat@Venkats-MacBook-Pro DAA Project % g++ -std=c++17 main.cpp -o main  -I/opt/homebrew/opt/openssl@3/include -L/opt/homebrew/opt/openssl@3/lib -lssl -lcrypto
venkat@Venkats-MacBook-Pro DAA Project % ./main
Enter board size (e.g., 8 for 8x8 board): 10
Board size set to 10x10

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 1
Enter passphrase: Sample test passphrase for knight tour encryption
Starting position: (0, 8)
Knight's Tour completed successfully.
Key sequence generated :
8 29 17 9 28 49 68 89 97 78 99 87 79 98 86 94 82 90 71 92 80 61 40 21 0 12 4 16 37 18 39 58 66 47 59 38 19 7 26 5 13 1 20 41 60 81 93 85 73 52 33 25 6 14 2 10 31
50 62 70 91 83 95 74 53 45 57 69 77 65 46 27 48 67 88 96 75 54 35 56 64 76 84 72 51 63 55 43 24 36 44 32 11 23 15 3 22 34 42 30

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 2
Enter filename to save the key: sample3.bin
Key saved successfully to sample3.bin

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 3
Available key files:
test3.bin
test10.bin
2.bin
pf.bin
sample3.bin
Enter key file name to load: sample3.bin
Key loaded successfully.

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
```



```
=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 4
Enter message to encrypt: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla vehicula nisl at ex scelerisque, in vestibulum ipsum vestibulum. Phasellu
s non sagittis libero, non cursus nisi.
Encrypted Message (in hex): 44 72 63 6c 71 11 2d 29 12 3b 0e 77 2b 0d 3a 31 20 7a 34 35 24 1d 49 78 65 78 28 30 46 7d 49 49 27 4c 4f 43 67 72 68 25 6c 65 7d 59 55
 22 3e 3c 27 53 01 7c 6a 67 76 24 3f 7c 4b 2a 37 32 7f 3c 50 45 50 26 38 2d 4f 3b 5e 2a 2b 0c 6b 57 57 18 25 34 74 3b 50 5a 5b 4e 6a 4d 5f 51 7e 72 23 23 7f 4c 0a
 68 6d 6e 65 60 7e 44 28 2c 0c 6e 0a 27 3c 17 3b 7e 24 3f 34 28 39 5f 5d 79 75 61 2a 30 75 7a 46 49 27 43 57 53 60 27 74 6a 63 21 67 48 5b 38 29 21 20 47 01 75 6f
 6c 67 78 70 1e 1e 28 34 3d 7f 29 40 5f 4a 30 3e 61 40 72 43 2a 76

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 5
Enter message to decrypt (in hex): 44 72 63 6c 71 11 2d 29 12 3b 0e 77 2b 0d 3a 31 20 7a 34 35 24 1d 49 78 65 78 28 30 46 7d 49 49 27 4c 4f 43 67 72 68 25 6c 65 7
d 59 55 22 3e 3c 27 53 01 7c 6a 67 76 24 3f 7c 4b 2a 37 32 7f 3c 50 45 50 26 38 2d 4f 3b 5e 2a 2b 0c 6b 57 57 18 25 34 74 3b 50 5a 5b 4e 6a 4d 5f 51 7e 72 23 23 7
f 4c 0a 68 6d 6e 65 60 7e 44 28 2c 0c 6e 0a 27 3c 17 3b 7e 24 3f 34 28 39 5f 5d 79 75 61 2a 30 75 7a 46 49 27 43 57 53 60 27 74 6a 63 21 67 48 5b 38 29 21 20 47 0
1 75 6f 6c 67 78 70 1e 1e 28 34 3d 7f 29 40 5f 4a 30 3e 61 40 72 43 2a 76
Decrypted Message: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla vehicula nisl at ex scelerisque, in vestibulum ipsum vestibulum. Phasellus non s
agittis libero, non cursus nisi.
```



```
=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 6

=== Encryption Key Report ===
Key Length: 500
Key Sequence: 8 29 17 9 28 49 68 89 97 78 99 87 79 98 86 94 82 90 71 92 80 61 40 21 0 12 4 16 37 18 39 58 66 47 59 38 19 7 26 5 13 1 20 41 60 81 93 85 73 52 33 25
 6 14 2 10 31 50 62 70 91 83 95 74 53 45 57 69 77 65 46 27 48 67 88 96 75 54 35 56 64 76 84 72 51 63 55 43 24 36 44 32 11 23 15 3 22 34 42 30 8 29 17 9 28 49 68 8
9 97 78 99 87 79 98 86 94 82 90 71 92 80 61 40 21 0 12 4 16 37 18 39 58 66 47 59 38 19 7 26 5 13 1 20 41 60 81 93 85 73 52 33 25 6 14 2 10 31 50 62 70 91 83 95 74
 53 45 57 69 77 65 46 27 48 67 88 96 75 54 35 56 64 76 84 72 51 63 55 43 24 36 44 32 11 23 15 3 22 34 42 30 8 29 17 9 28 49 68 89 97 78 99 87 79 98 86 94 82 90 71
 92 80 61 40 21 0 12 4 16 37 18 39 58 66 47 59 38 19 7 26 5 13 1 20 41 60 81 93 85 73 52 33 25 6 14 2 10 31 50 62 70 91 83 95 74 53 45 57 69 77 65 46 27 48 67 88
96 75 54 35 56 64 76 84 72 51 63 55 43 24 36 44 32 11 23 15 3 22 34 42 30 8 29 17 9 28 49 68 89 97 78 99 87 79 98 86 94 82 90 71 92 80 61 40 21 0 12 4 16 37 18 39
 58 66 47 59 38 19 7 26 5 13 1 20 41 60 81 93 85 73 52 33 25 6 14 2 10 31 50 62 70 91 83 95 74 53 45 57 69 77 65 46 27 48 67 88 96 75 54 35 56 64 76 84 72 51 63 5
5 43 24 36 44 32 11 23 15 3 22 34 42 30 8 29 17 9 28 49 68 89 97 78 99 87 79 98 86 94 82 90 71 92 80 61 40 21 0 12 4 16 37 18 39 58 66 47 59 38 19 7 26 5 13 1 20
41 60 81 93 85 73 52 33 25 6 14 2 10 31 50 62 70 91 83 95 74 53 45 57 69 77 65 46 27 48 67 88 96 75 54 35 56 64 76 84 72 51 63 55 43 24 36 44 32 11 23 15 3 22 34
42 30
Hashed Passphrase: fa44935a6e80c1508a3b6661af71d6f4d0b6e6b4bcbaa5219633574d82a8fd3a
Starting Position: (0, 8)

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 7
Time to generate key: 0 ms
Time to encrypt message: 0 ms
Time to decrypt message: 0 ms

=== Knight's Tour Encryption System ===
1. Generate new key
2. Save key to file
3. Load key from file
4. Encrypt message
5. Decrypt message
6. Generate report
7. Measure performance
8. Exit
Choice: 8
Exiting...
○ venkat@Venkats-MacBook-Pro DAA Project %
```