Semester course Project

Malicious Login Detector

University: Euromed University of Fes (UEMF).

Department: EIDIA - Ecole d’Ingénierie Digitale et d’Intelligence Artificielle.

Module: Advanced Algorithmics.

Instructor: DEGU WORKNEH Abebaw.

Group Members:

- AITAOUICHA Yassine - 2500845
- EL KHOUDARI Marwa - 2300542
- KHLIFI TAGHZOUTI Adam - 2500940
- BOULAHBACH Malak - 2300431
- EL MOURABIT Hamza - 2300697
- HANNOUN Haytam -

Date: December 24, 2025

Project Overview/Objectives

This project simulates a **login monitoring system** that detects suspicious or malicious login attempts such as brute-force or credential stuffing.

The main objectives of this project are:

- To implement fundamental data structures such as arrays, linked lists, stacks, queues, trees, and graphs from scratch.
- To apply these data structures to a realistic cybersecurity scenario involving malicious login detection.
- To analyze and compare **time and space complexity** for different algorithms and structures.
- To experimentally evaluate performance through timing comparisons.
- To integrate all modules into a single **command-line based Malicious Login Detector system**.

Dataset Description

The system processes login attempts provided through a **CSV dataset**. Each record represents a single login attempt with attributes such as:

- timestamp
- username
- status (success or failure)
- ip address

System Architecture Overview

The project contains 3 main folders:

- data folder which contains datasets.

- include folder which contains header files to unify the work and have a general view of the project.

- The src folder contains the implementation of the header files.

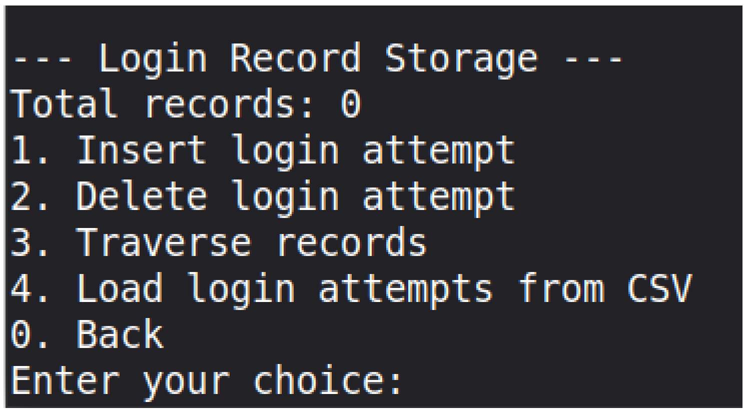
Modules and Tasks

1)Arrays and Linked Lists – Login Record Storage

Login attempts can be loaded from a CSV file or inserted one by one. Each record from the file is inserted into both the array and the linked list. This ensures that both data structures store the same data, making performance comparison fair and accurate.

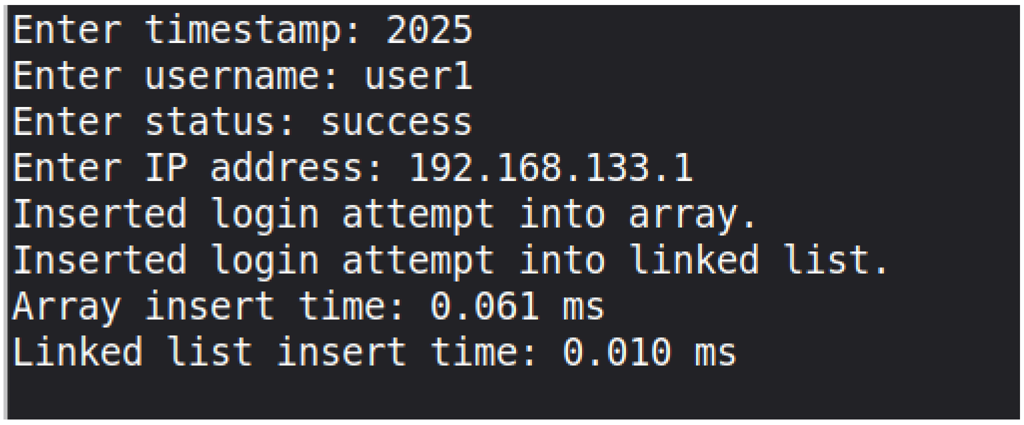
An array has a fixed size and provides very fast access to elements using an index. However, inserting or deleting elements in the middle requires shifting data, which takes linear time. An ArrayList behaves similarly in terms of access and traversal but supports dynamic resizing.

This resizing operation occasionally requires copying all elements to a new memory location, which introduces additional overhead. Overall, both structures have similar time complexity for most operations, but ArrayLists provide more flexibility at the cost of extra memory usage.



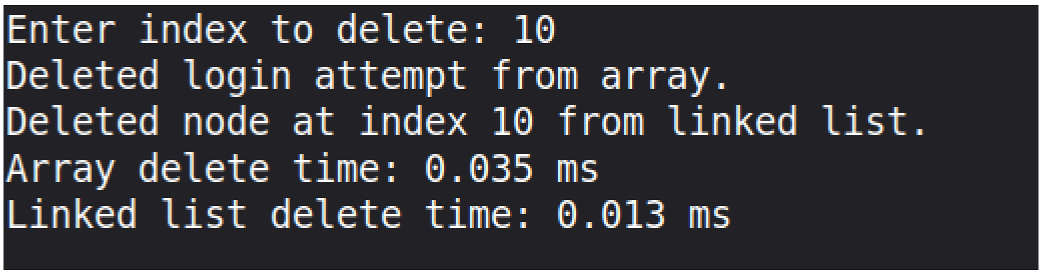
```
--- Login Record Storage ---
Total records: 0
1. Insert login attempt
2. Delete login attempt
3. Traverse records
4. Load login attempts from CSV
0. Back
Enter your choice:
```

insertion:



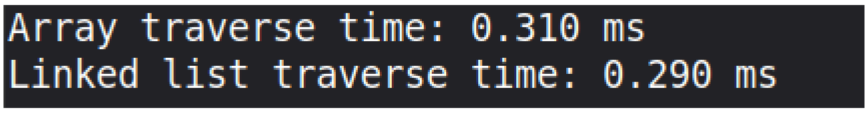
```
Enter timestamp: 2025
Enter username: user1
Enter status: success
Enter IP address: 192.168.133.1
Inserted login attempt into array.
Inserted login attempt into linked list.
Array insert time: 0.061 ms
Linked list insert time: 0.010 ms
```

deletion:



```
Enter index to delete: 10
Deleted login attempt from array.
Deleted node at index 10 from linked list.
Array delete time: 0.035 ms
Linked list delete time: 0.013 ms
```

traversal:

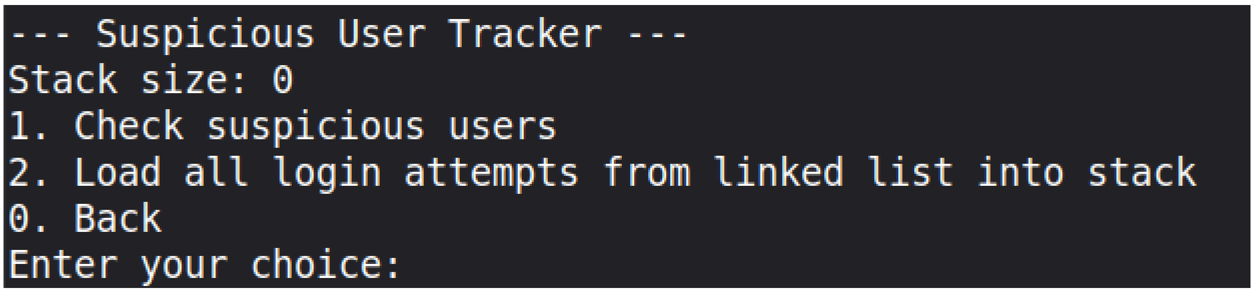


```
Array traverse time: 0.310 ms
Linked list traverse time: 0.290 ms
```

2) Stack – Suspicious User Tracker

This module uses a stack to track recent login attempts and identify suspicious users based on consecutive failures. The stack follows the Last-In-First-Out principle and is implemented using a fixed-size array with a top index. The **push()** function adds a new login record to the stack, while the **pop()** function removes the most recent record.

Login attempts stored in the linked list are transferred to the stack using the **populate_stack_from_linkedlist()** function, ensuring the stack reflects the actual order of login activity. The **check_suspicious_users()** function scans the stack and counts consecutive failed logins for each user. If a user reaches a predefined threshold k, the user is flagged as suspicious and displayed.



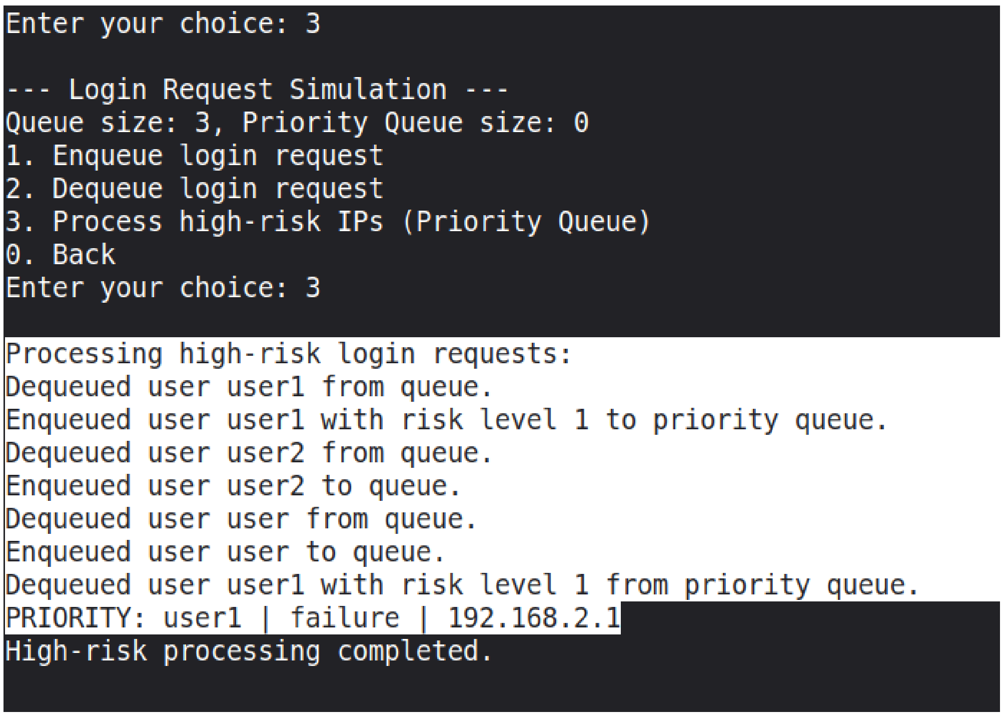
```
--- Suspicious User Tracker ---
Stack size: 0
1. Check suspicious users
2. Load all login attempts from linked list into stack
0. Back
Enter your choice:
```

3) Queue → Login Request Simulation

This module simulates incoming login requests using a FIFO queue. Login attempts are added to the queue using the **enqueue()** function and processed in the order they arrive using the **dequeue()** function. The queue is implemented as a circular array, allowing efficient use of memory while maintaining constant-time enqueue and dequeue operations.

To prioritize potentially dangerous activity, a priority queue is added for high-risk login attempts. **The process_high_risk_ips()** function removes each request from the normal queue and checks its risk level. Failed login attempts are treated as high-risk and inserted into the priority queue using **pq_enqueue()**, where elements are ordered by risk level. Normal login attempts are returned to the regular queue.

High-risk requests are processed first by removing them from the priority queue using the **pq_dequeue()** function and displaying their details. A command-line menu allows users to enqueue requests, dequeue them, and trigger high-risk processing. This module demonstrates how queues and priority queues can be combined to simulate real authentication systems that prioritize suspicious activty.



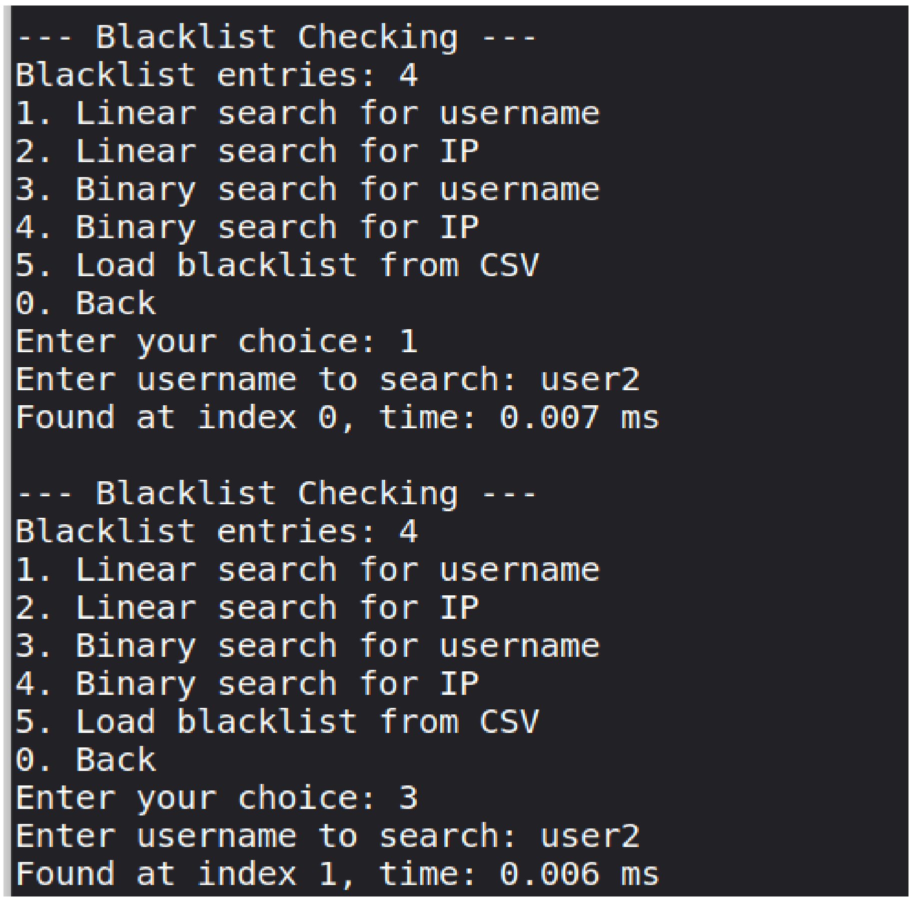
```
Enter your choice: 3

--- Login Request Simulation ---
Queue size: 3, Priority Queue size: 0
1. Enqueue login request
2. Dequeue login request
3. Process high-risk IPs (Priority Queue)
0. Back
Enter your choice: 3

Processing high-risk login requests:
Dequeued user user1 from queue.
Enqueued user user1 with risk level 1 to priority queue.
Dequeued user user2 from queue.
Enqueued user user2 to queue.
Dequeued user user from queue.
Enqueued user user to queue.
Dequeued user user1 with risk level 1 from priority queue.
PRIORITY: user1 | failure | 192.168.2.1
High-risk processing completed.
```

4) Searching → Blacklist Checking

This module allows the system to check whether a username or IP address is present in a blacklist, using both linear and binary search methods. Blacklist entries are loaded from a CSV file using the **load_blacklist()** function, which stores usernames and IPs in separate arrays. For linear search, the functions **linear_search_username()** and **linear_search_ip()** iterate through the arrays to find a match.

For faster search on sorted data, the module implements binary search with the functions **binary_search_username()** and **binary_search_ip(),** and the blacklist is sorted first using **sort_blacklist_by_username()** or **sort_blacklist_by_ip()**.

A command-line menu allows users to load the blacklist and perform both types of searches while measuring execution time for performance comparison.



```
--- Blacklist Checking ---
Blacklist entries: 4
1. Linear search for username
2. Linear search for IP
3. Binary search for username
4. Binary search for IP
5. Load blacklist from CSV
0. Back
Enter your choice: 1
Enter username to search: user2
Found at index 0, time: 0.007 ms

--- Blacklist Checking ---
Blacklist entries: 4
1. Linear search for username
2. Linear search for IP
3. Binary search for username
4. Binary search for IP
5. Load blacklist from CSV
0. Back
Enter your choice: 3
Enter username to search: user2
Found at index 1, time: 0.006 ms
```

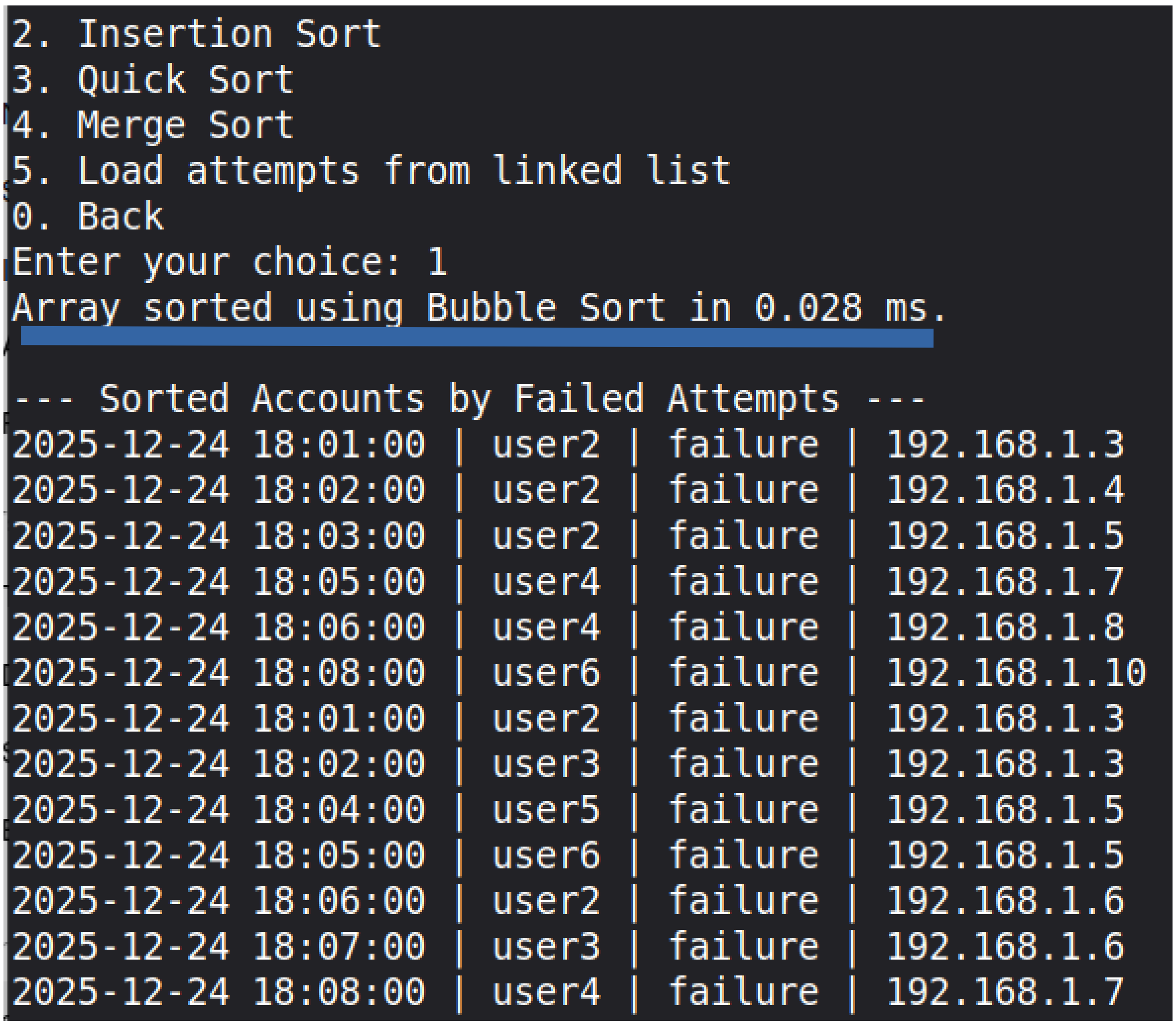
5) Sorting → Ranking of Accounts

This module ranks user accounts based on the number of failed login attempts by sorting login records stored in an array.

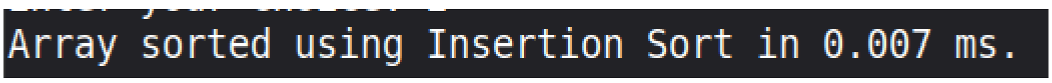
The l**oad_from_linkedlist()** function copies login attempts from the linked list into the array for sorting. Four classical sorting algorithms are implemented: **bubble_sort()**, **insertion_sort()**, **quick_sort(),** and **merge_sort().**

Each algorithm orders the array so that users with the highest number of failed logins appear first. Execution time is measured for each algorithm to allow performance comparison.

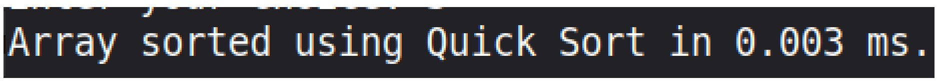
The **sort_menu()** function provides a simple command-line interface to load records, choose a sorting method, and display the sorted results.



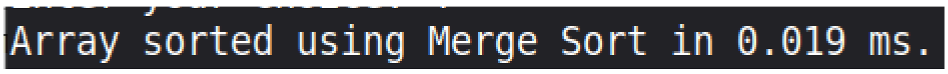
```
2. Insertion Sort
3. Quick Sort
4. Merge Sort
5. Load attempts from linked list
0. Back
Enter your choice: 1
Array sorted using Bubble Sort in 0.028 ms.

--- Sorted Accounts by Failed Attempts ---
2025-12-24 18:01:00 | user2 | failure | 192.168.1.3
2025-12-24 18:02:00 | user2 | failure | 192.168.1.4
2025-12-24 18:03:00 | user2 | failure | 192.168.1.5
2025-12-24 18:05:00 | user4 | failure | 192.168.1.7
2025-12-24 18:06:00 | user4 | failure | 192.168.1.8
2025-12-24 18:08:00 | user6 | failure | 192.168.1.10
2025-12-24 18:01:00 | user2 | failure | 192.168.1.3
2025-12-24 18:02:00 | user3 | failure | 192.168.1.3
2025-12-24 18:04:00 | user5 | failure | 192.168.1.5
2025-12-24 18:05:00 | user6 | failure | 192.168.1.5
2025-12-24 18:06:00 | user2 | failure | 192.168.1.6
2025-12-24 18:07:00 | user3 | failure | 192.168.1.6
2025-12-24 18:08:00 | user4 | failure | 192.168.1.7
```



```
Array sorted using Insertion Sort in 0.007 ms.
```



```
Array sorted using Quick Sort in 0.003 ms.
```



```
Array sorted using Merge Sort in 0.019 ms.
```

6) Trees → Classification of Attempts

This module organizes login attempts into a binary search tree to classify users based on their login success and failure counts.

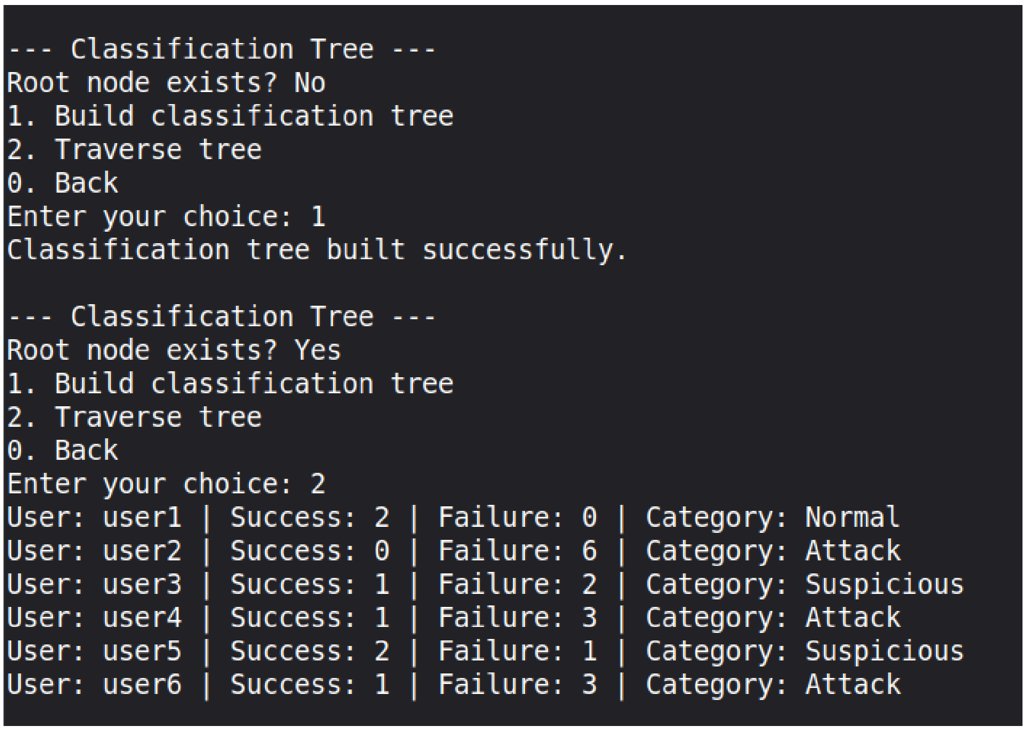
Each tree node represents a unique username and stores the number of successful and failed attempts.

The **`create_node()`** function initializes a new node, while **`insert_tree()` adds** login attempts to the tree, updating counts for existing users.

The **`build_classification_tree()`** function constructs the tree from the array of login attempts, ensuring that each user is represented once.

Users are classified using the **`classify_user()`** function into categories: Normal, Suspicious, or Attack, based on their failure count.

The **`traverse_tree()`** function performs an in-order traversal to display each user's login statistics and category. **Conclusion**



```
--- Classification Tree ---
Root node exists? No
1. Build classification tree
2. Traverse tree
0. Back
Enter your choice: 1
Classification tree built successfully.

--- Classification Tree ---
Root node exists? Yes
1. Build classification tree
2. Traverse tree
0. Back
Enter your choice: 2
User: user1 | Success: 2 | Failure: 0 | Category: Normal
User: user2 | Success: 0 | Failure: 6 | Category: Attack
User: user3 | Success: 1 | Failure: 2 | Category: Suspicious
User: user4 | Success: 1 | Failure: 3 | Category: Attack
User: user5 | Success: 2 | Failure: 1 | Category: Suspicious
User: user6 | Success: 1 | Failure: 3 | Category: Attack
```

7) Graphs → Attack Spread Simulation

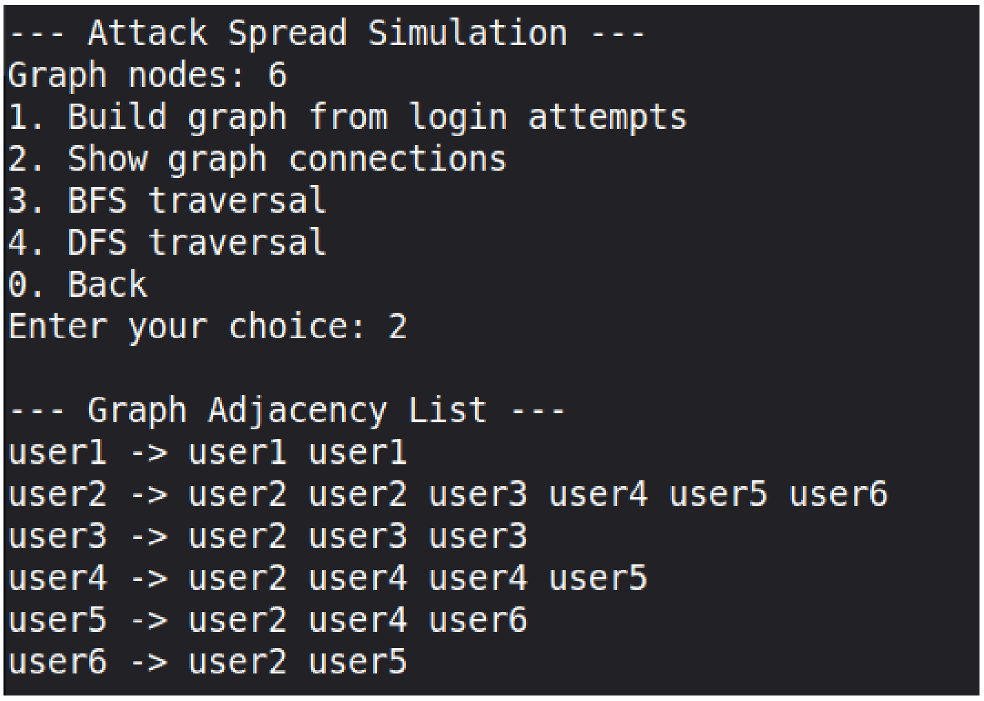
This module models login accounts as a graph to detect possible attack spread via shared IP addresses.

Each user is represented as a graph node **(add_node())**, and an edge connects two users if they share the same IP **(add_edge())**.

The graph is automatically constructed from login attempts using **build_graph_from_login_attempts()**.

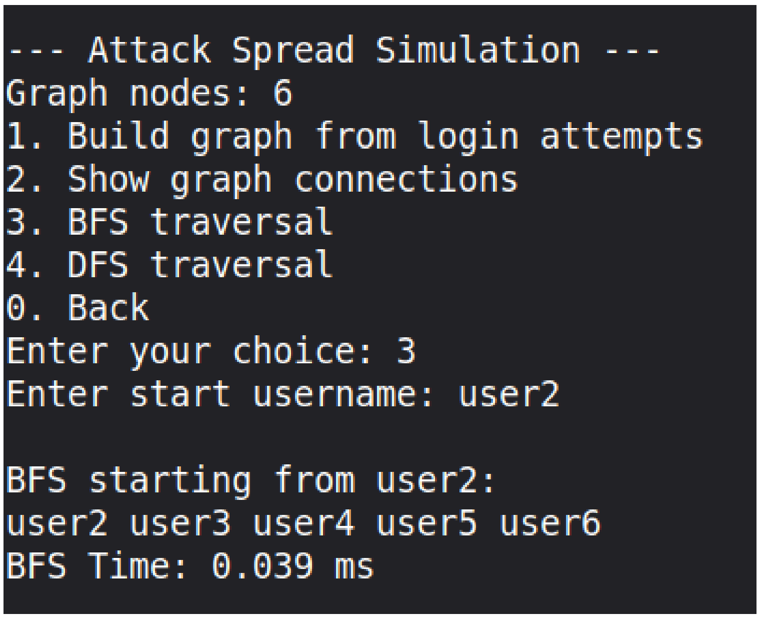
To explore potential attack paths, the module implements breadth-first search **(bfs())** and depth-first search **(dfs())** traversals, both measuring execution time.

The adjacency list of the graph can be displayed using **print_graph().**

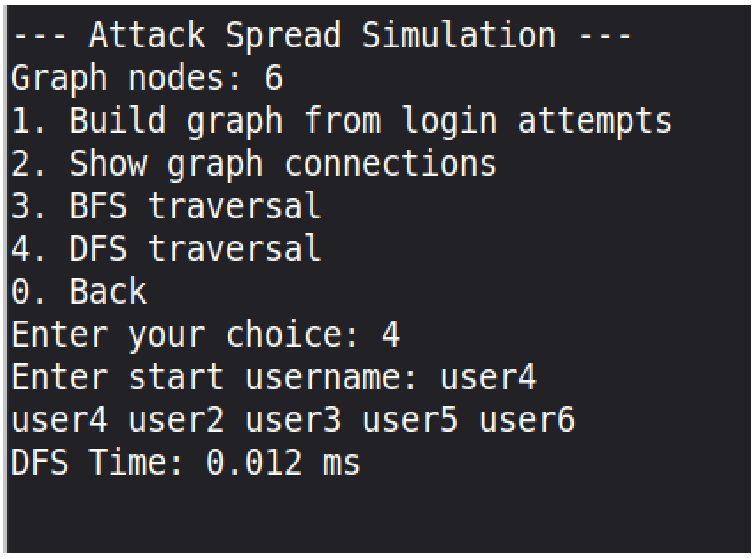


```
--- Attack Spread Simulation ---
Graph nodes: 6
1. Build graph from login attempts
2. Show graph connections
3. BFS traversal
4. DFS traversal
0. Back
Enter your choice: 2

--- Graph Adjacency List ---
user1 -> user1 user1
user2 -> user2 user2 user3 user4 user5 user6
user3 -> user2 user3 user3
user4 -> user2 user4 user4 user5
user5 -> user2 user4 user6
user6 -> user2 user5
```

BFS vs DFS:



```
--- Attack Spread Simulation ---
Graph nodes: 6
1. Build graph from login attempts
2. Show graph connections
3. BFS traversal
4. DFS traversal
0. Back
Enter your choice: 3
Enter start username: user2

BFS starting from user2:
user2 user3 user4 user5 user6
BFS Time: 0.039 ms
```



```
--- Attack Spread Simulation ---
Graph nodes: 6
1. Build graph from login attempts
2. Show graph connections
3. BFS traversal
4. DFS traversal
0. Back
Enter your choice: 4
Enter start username: user4
user4 user2 user3 user5 user6
DFS Time: 0.012 ms
```

Conclusion:

The semester project demonstrates how classical data structures and algorithms can be applied to a real-world cybersecurity scenario.

Each module highlights the strengths of different structures: arrays and linked lists for storage and traversal, stacks for detecting consecutive failures, queues and priority queues for simulating and prioritizing login requests, searching algorithms for blacklist checking, sorting for ranking accounts, trees for classifying users, and graphs for tracing attack spread.

Through this project, we learned to implement these structures into a real world case, analyze their performance, and integrate them into a complete system.