PROJECT REPORT

Group Members: Muhammad Saad
Fatima Ali
Suleman Malik

Course: CS221 DSA

Project Overview:

The **Psych Defender** is a cybersecurity tool designed to analyze URLs and determine if they are safe, suspicious, or high-risk. Phishing attacks often rely on visual deception (homograph attacks) or slightly modified URLs (typo squatting) to trick users.

Data Structures and Algorithms:

1. Keyword Pattern Scanner:

- ***Data Structures Used:*** *Trie (Aho–Corasick Automaton)*
- ***Purpose:*** *Efficiently detect multiple phishing-related keywords in text.*
- ***Implementation:***
  - *A **trie** is built where each node represents a character.*
  - *Leaf nodes mark the end of a keyword.*
- ***How It Works:***
  - *The text is scanned once, and all matches are reported in linear time.*

2. Sentiment and Urgency Detector:

- ***Data Structures Used:*** *Hash Map, Arrays*
- ***Purpose:*** *Identify urgent or manipulative language in messages.*
- ***Implementation:***
  - *A **hash map** (using an array of structures) stores words/phrases with urgency scores.*
  - ***Arrays** are used for n-gram generation (1-gram, 2-gram, 3-gram).*
- ***How It Works:***

    - *A sliding window generates n-grams, which are looked up in the hash map.*
    - *Cumulative urgency scores determine the message’s risk level.*

3. Attachment Risk Analyzer

- ***Data Structures Used:*** *Trie*
- ***Purpose:*** *Identify and classify dangerous file extensions.*
- ***Implementation:***
    - *A **trie** is built where each node represents a character in a file extension.*
    - *Leaf nodes store a risk level (1–5) instead of a simple end-of-word flag.*
- ***How It Works:***
    - *The trie is traversed to detect file extensions in the text.*
    - *Detected extensions are classified by their risk score.*

4. Confidence Explanation System:

- ***Data Structures Used:*** *Singly Linked List*
- ***Purpose:*** *Provide users with clear, rule-based explanations of detected risks.*
- ***Implementation:***
    - *Each node in the **linked list** contains:*
        - *Rule name (e.g., "too_many_links")*
        - *Explanation text*
        - *Pointer to the next node*
- ***How It Works:***
    - *The system traverses the linked list to fetch explanations for triggered rules.*
    - *Explanations are displayed in a clean, user-friendly format.*

5. Multi-Language Detection System

- ***Data Structures Used:*** *Linked List, Wide Strings (Unicode)*
- ***Purpose:*** *Detect homograph attacks using mixed-script characters.*
- ***Implementation:***
    - *A **linked list** stores mappings between Unicode characters and their ASCII equivalents.*
    - *Each node contains:*
        - *Unicode character (e.g., Cyrillic 'а')*
        - *ASCII equivalent ('a')*
        - *Next pointer*
- ***How It Works:***
    - *The input string is converted to a wide string for Unicode support.*
    - *Each character is checked against Unicode ranges (e.g., Latin, Cyrillic, Greek).*
    - *The linked list is used to identify suspicious mappings.*

6. Link Ratio Detector

- ***Data Structures Used:*** *Arrays, Strings parsing*
- ***Purpose:*** *Calculate the ratio of links to words in a message.*
- ***Implementation:***
    - ***Character arrays*** *and* ***string manipulation*** *functions are used for parsing.*
    - ***Integer counters*** *track:*
        - *linkCount*
        - *wordcount*
    - ***Floating-point*** *variables compute the ratio.*
- ***How It Works:***
    - *Word boundaries are detected using spaces, tabs, and newlines.*
    - *URL patterns are matched using string::find() and length checks.*
    - *The ratio is compared against thresholds to determine risk.*

URL and Domain Analysis Tools:

1. URL Heuristic Analysis

- ***Data Structures Used:*** *Strings and Boolean flags*
- ***Purpose:*** *Evaluate URLs based on suspicious features like length, symbols, and keywords.*
- ***Implementation:***
    - *Utilizes C++ Standard Library's string class for URL storage and manipulation*
    - *Employs character-level parsing algorithms for pattern detection*
    - *Implements heuristic scoring through integer accumulation and string concatenation for result formatting.*
- ***How It Works:***
    - *The system applies multiple heuristic checks including length validation, symbol detection, IP address parsing, and keyword matching*
    - *Each heuristic violation contributes weighted scores to a cumulative risk assessment*
    - *Final classification follows threshold-based categorization into LOW RISK (0-14), SUSPICIOUS (15-29), or HIGH RISK (30+) categories.*

2. Visual Spoof Analysis

- ***Data Structures Used:*** *Linked List*
- ***Purpose:*** *Detect homoglyph attacks by normalizing domains.*
- ***Implementation:***
    - *A* ***singly linked list*** *stores mappings of visually similar characters (e.g., 'o' → '0', 'rn' → 'm').*

- ***How It Works:***
    - *The normalize Domain function traverses the linked list to replace suspicious characters.*
    - *Normalized domains are compared using a simple character-by-character match.*

3. Domain Similarity Analysis

- ***Data Structures Used:*** *2D Array (Dynamic Programming Table)*
- ***Purpose:*** *Quantify similarity between domains using the Levenstein distance.*
- ***Implementation:***
    - *A **2D array** is used to store intermediate values for edit distance calculations.*
    - *The user input is checked against a file for trusted domains.*
- ***How It Works:***
    - *The algorithm fills the DP table to compute the minimum edits needed to transform one domain into another.*
    - *The result is converted into a similarity percentage.*

4. Hash Table

- ***Data Structures Used:*** *Array (The main table), Linked List (To handle collisions)*
- ***Purpose:*** *To instantly check if a domain exists in the trusted list (like google.com) without searching through the entire file line by line.*
- ***Implementation:***
    - *A fixed-size **Array** of 1,009 pointers is created to act as "buckets" for data storage.*
    - ***Linked List** node structure is used so multiple domains can live in the same bucket if they share the same index (Chain collision resolution).*
    - *A **Hash Function** is implemented to convert string text into unique integer indices.*
- ***How It Works:***
    - *The system runs the domain name through a hash function (mathematical formula) to calculate a specific index number, identifying exactly which "bucket" the domain belongs to.*
    - *It jumps directly to that bucket in the array and traverses the linked list to check if the domain matches a trusted entry, providing an instant result without scanning the whole list.*

Performance Analysis:

Heuristic Single-Pass Loop:

The code iterated through the URL string multiple times (once to count dots, once for '@', once for '%', etc.), resulting in **O(5N)**.

Optimized: Consolidated these checks into a single loop, reducing operations and complexity to **O(N)**.

Keyword Search in analyzeURL(string url):

Instead of calling a search function 5 times for different keywords, we integrated the check into the main analysis flow, reducing function call overhead.

Levenshtein Distance:

Previously accepted string by value, creating a copy of the string in memory every time. We updated function signatures to const string &, passing by reference. This avoids creating copies.

SentimentAndUrgency:

Removed the (Extra lowering text) function.

Separated the words, with their scores, for variety.

Main:

Removed the Extra Loops in main reducing time complexity from $O(n^2)$ to $O(n)$.

Replace the calling function 5 times to only once and stored its value for comparison.

Changed the hard code input to user input.

Challenges Faced & Solutions:

Handling Multi-Word Phrases (N-Grams):

One significant challenge we encountered was that our initial keyword scanner treated spaces as delimiters, restricting detection to single words only. This meant the system failed to flag common phishing phrases like "limited time" or "act now." To resolve this, we implemented an **N-gram** technique. This allowed the system to generate and analyze sequences of words, ensuring that suspicious multi-word context was accurately detected alongside individual keywords.

Trusted Database: Another challenge involved selecting the efficient data structure for the trusted domain database. Initially, we implemented a **Binary Search Tree (BST)**. However, because our domain data was often sorted alphabetically, the tree became skewed (unbalanced), effectively degenerating into a Linked List with a worst-case search time of **O(N)**. To fix this inefficiency, we migrated to a **Hash Table** with collision handling. This optimization reduced the average lookup time to **O(1)**, ensuring instant verification even as the database grows.

Future Improvements:

-Real-Time Browser Integration ("Grammarly for Security"):

We plan to evolve this tool into a browser extension that functions like Grammarly for web security. Instead of manual checks, it will run in the background, analyzing URLs in real-time as users navigate. By monitoring the address bar and cross-referencing our database, the extension will provide immediate, on-screen warnings if a user visits a suspicious or high-risk site.

-Live API Integration

Connecting to real-time blacklists (like Google Safe Browsing API) instead of a static text file.

-Machine Learning:

Replacing the static keyword heuristics with a trained ML model for better accuracy on novel phishing phrases.

Conclusion:

The project extensively utilized Data Structures and Algorithms studied to build efficient detection mechanisms. **Arrays and string manipulation** served as the foundation for parsing URLs and calculating link ratios. **Linked Lists** were implemented to handle collisions within our Hash Table and to dynamically store rule-based explanations and character mappings for visual spoof analysis. Furthermore, **Tries (Prefix trees)** were employed to efficiently detect multiple phishing-related keywords in text.