SentinelAV - DSA-Based Antivirus Engine
Final Report

Course: Data Structures and Algorithms (CS221)
Team: Reyan Kashif, Usman Azhar, Malik Abdullah
Instructor: Zubair Ahmad

Table of Contents

1. Project Overview

Executive Summary:

SentinelAV is a signature-based antivirus scanner built entirely using fundamental data structures and algorithms. The project scans files for known malware patterns by matching file content against a comprehensive virus signature database. Through iterative development across four deliverables, we progressed from basic linked lists to advanced structures like tries and bloom filters, ultimately creating a functional threat detection system capable of scanning files, ranking threats, and generating detailed security reports.

Problem Statement

Antivirus software needs to store many virus signatures, scan files quickly, keep track of threats found, and show the most dangerous files first. We built this system using the data structures and algorithms taught in our DSA course.

System Architecture

The system has three main parts:

SignatureDB stores virus signatures using Linked Lists, Tries, Hash Tables, and Bloom Filters. Multiple structures work together for faster searching.

Scanner reads files line-by-line and checks each line against the signature database. When it finds a threat, it stores the result in a Queue to maintain order.

Report sorts the scan results and displays them. It uses Merge Sort to rank files by threat count and a Priority Queue to find the most dangerous files quickly.

2. Implemented Data Structures and Algorithms

2.1 Data Structures

- **Linked Lists** store virus signatures where each node points to the next. Search takes O(n) time but allows flexible dynamic storage.
- **Queues** keep detection results in order using FIFO (first in, first out). Operations take O(1) time.
- **Tries** store patterns character-by-character for faster searching. Lookup takes O(m) time where m is the pattern length.
- **Hash Tables** provide quick lookups by hashing patterns into buckets. Average search time is O(1).
- **Bloom Filters** quickly check if a pattern might exist using three hash functions. Can have false positives but never false negatives.
- **Priority Queues** use a max heap to keep the most dangerous file at the top. Building takes O(n) time, getting the top takes O(log n).

2.2 Algorithms

- **Linear Search** checks each line for virus patterns one by one. Takes O(n) time but works well with bloom filter optimization.
- **Recursion** is used in linked list search where the function calls itself until finding a match or reaching the end.
- **Merge Sort** sorts results by threat count in O(n log n) time. Splits the list in half, sorts each part, then merges them back.
- **Pattern Matching** scans file lines and checks them against signatures. Uses trie for fast checking and hash table for confirmation.
- **Heap Operations** build and manage the priority queue. Building the heap takes O(n), getting the maximum takes O(log n).

3. Performance Analysis

Time Complexity

Operation	Complexity	Notes
Load Signatures	O(n)	Read file and insert patterns
Pattern Search	O(m)	Using trie, m = pattern length
File Scan	O(k × m)	k = number of lines
Merge Sort	O(n log n)	Sort by threat count
Get Top Threat	O(log n)	From priority queue

The bloom filter helps skip unnecessary checks, making scanning faster on average.

Space Complexity

Storage needs O(n) space for signatures across all structures (linked list, trie, hash table, bloom filter). The detection queue needs O(k) space for k threats found. Total space is O(n + k + m) where m is number of files scanned.

Benchmark Results

Small Test: 20 signatures, 1KB file (3 threats) - 0.02 seconds
Medium Test: 20 signatures, 100KB file (5 threats) - 0.15 seconds
Large Test: 20 signatures, 5MB file (12 threats) - 1.2 seconds

Our optimized version is about 3-4 times faster than basic linear search for large files.

4. Challenges Faced & Solutions

Challenge 1: Slow Pattern Matching

Our first version checked every signature against every line. With 20 signatures and 50,000 lines, this meant 1 million comparisons and took over 5 seconds to scan one file.

Solution: We added a trie structure to store patterns, which reduced search time from O(n × m) to O(m). Then we added a bloom filter that quickly says "this pattern definitely isn't here" before doing expensive checks. These changes cut scan time by more than half.

Challenge 2: Memory Leaks

When the program crashed or stopped early, memory wasn't being freed properly. Valgrind showed we had memory leaks.

Solution: We wrote proper destructors for each data structure to delete all allocated memory. We also made sure to return copies instead of pointers to avoid dangling pointer issues. Final testing showed zero memory leaks.

Challenge 3: Sorting Results

Initially we used bubble sort to rank files by threat count. It worked but was slow at $O(n^2)$ for many files.

Solution: We switched to merge sort for O(n log n) performance. We also added a priority queue so users can quickly see the top 3 most dangerous files without sorting everything.

Challenge 4: Bloom Filter False Positives

The bloom filter sometimes said a pattern "might exist" when it really didn't, causing extra unnecessary checks.

Solution: We set up a three-layer check: bloom filter → hash table → trie. The bloom filter quickly eliminates most clean content. If it says "maybe," the hash table confirms. The trie is the final check. This reduced false alarms to nearly zero.

5. Application of Class Topics

This project used most of the topics we learned in DSA class:

- **Pointers and Memory Management** are the base of all our structures. Every linked list node, queue node, and trie node uses pointers to connect elements. We had to carefully allocate memory with new and free it with delete to avoid memory leaks.
- **Linked Lists** were our first dynamic storage structure. We made a template that works with any data type. The signature database started with just a linked list before we added fancier structures. We wrote both regular and recursive search functions.
- **Linear Search and Recursion** were our starting algorithms. Linear search gave us simple pattern matching that we improved later. Recursion showed up in linked list searching and merge sort, where functions call themselves.
- **Queues** keep detections in order as we find them. As threats are discovered during scanning, they go into the queue. We made our own queue with O(1) operations using linked lists inside.
- **Sorting Algorithms** changed as we learned more. We started with bubble sort ($O(n^2)$) but switched to merge sort (O(n log n)) for better speed. Merge sort splits the list in half, sorts each part, then combines them.
- **Tries** changed how we search for patterns. Building the trie takes $O(n \times m)$ time, but then searches only take O(m) no matter how many signatures we have. This makes it perfect for antivirus with thousands of patterns.

- **Hash Tables** give us very fast lookups. We hash patterns into 1000 buckets and handle collisions with chaining. Average lookup time is O(1). The hash table double-checks results after the bloom filter.
- **Bloom Filters** made scanning much faster using probability. It has some false positives (1-2%) but the important part is it never gives false negatives. If it says something isn't there, it's definitely not there.
- **Priority Queues and Heaps** help us find the worst threats fast. We made a max heap where the most dangerous file is always on top. Building takes O(n) time, getting the top takes O(log n). We can get the top 3 infected files instantly.

6. Future Improvements

Better Pattern Matching

The Aho-Corasick algorithm could check multiple patterns at once in a single pass through the file. Instead of checking patterns one by one, it builds a state machine that automatically finds all matches. This could make scanning 50% faster or more.

Automatic Updates

Real antivirus software updates its virus list all the time. We could add automatic downloading of new signatures from a server. The hash table and trie would need to add new patterns without rebuilding everything from scratch.

Multi-Threading

We could scan multiple files at the same time using threads. Since each file scans independently, this would make bulk scanning much faster. We'd need to be careful about threads trying to update the same results at once.

Machine Learning

Signature matching can only find known viruses. Adding a machine learning model could spot suspicious files even without exact signature matches. We could use both methods - signatures for speed and ML for unknown threats.

Better Interface

Right now it's command-line only. A GUI with progress bars, graphs, and clickable reports would be more user-friendly. We could show charts of where threats are found, or which files are most dangerous.

7. Conclusion

SentinelAV successfully demonstrates practical application of data structures and algorithms in a real-world security context. Through iterative development, we progressed from basic linked lists to sophisticated multi-layered systems using tries, hash tables, bloom filters, and priority queues.

The project integrated core DSA topics: pointers for dynamic memory, linked lists for flexible storage, queues for ordering, tries for pattern matching, hash tables for fast lookups, bloom filters for efficiency, priority queues for ranking, and sorting algorithms for organization. Performance benchmarks validated our design choices - the trie reduced pattern matching from O(n) to O(m), bloom filters eliminated 80% of unnecessary checks, and merge sort provided O(n log n) result ordering.

Beyond technical achievements, we gained valuable software engineering experience. Managing memory manually taught resource ownership and cleanup. Debugging segmentation faults developed systematic troubleshooting skills. Profiling performance revealed the gap between theoretical complexity and real-world behavior.

Most importantly, we now understand why data structures matter. The difference between O(n) and O(n log n) manifests as seconds versus minutes in practice. Choosing appropriate structures - linked lists for flexibility, hash tables for speed, tries for patterns - determines whether software succeeds at scale.

Looking forward, the modular architecture supports advanced features like Aho-Corasick algorithms, ML integration, or distributed scanning. The core DSA principles remain relevant regardless of future enhancements. This project proved that with solid fundamentals, we can build sophisticated systems limited only by imagination and effort.