SE301 - Project

SE301 Teaching Team
v0.4

October 17, 2025

Contents

1 Purpose

Upon successful completion of this project, students should be able to do the following,

1. Perform multithreaded programming using the Java programming language.
2. Apply the clean code practices.
3. Fine-tune the JVM for optimal performance.

Note: Students are free to use LLM (Large Language Models) as a tool, but expected to fully understand any code or concepts generated using these tools.

2 Overview

Your team is a group of elite senior engineers brought in to handle a critical incident at "Legacy Systems Inc." A former employee developed a password auditing tool years ago, written as a single, monolithic Java file. This tool is now mission-critical for an upcoming security audit, but it is dangerously flawed: it is slow, unmaintainable, crashes under load, uses insecure practices, and is written with severely outdated Java idioms.

Provided auditing tool executes a dictionary attack on a list of password hashes. Dictionary attack is a password-cracking method where an attacker tries every word from a prepared list ("dictionary") of likely passwords instead of random guessing.

- The list can contain real dictionary words, names, keyboard patterns, or common passwords ("123456", "password").
- It's faster than brute-force because it skips unlikely combinations.

It's like guessing a friend's password by trying all the words you know they might use. But on a much larger scale.

Your mission is to deconstruct, refactor and improve the provided code into a high-performance, concurrent, and robust system. The goal is to create an application that can audit a provided list of insecure password hashes in the shortest possible time. This is a competitive challenge. Your team's final submission will be benchmarked against others. Performance is important, but it has to be built upon a foundation of excellent software design.

3 Requirements

Refactor the provided DictionaryAttack.java code into a modern, concurrent, and architecturally sound application that correctly and rapidly finds all crackable passwords from a given list of target hashes.

Your final application must address all the following issues present in the provided code.

3.1 Architectural Refactoring

3.1.1 Deconstruct the Monolith

You must break the single DictionaryAttack.java file into a logical, multi-component system. A submission that remains a single file will be disqualified.

3.1.2 Apply SOLID Principles

Your new architecture must demonstrate a clear separation of concerns. At a minimum, you should have distinct components for:

1. Target Hash Loading/Management
2. Live Status Reporting (See the Section 3.4.1)
3. The Core Concurrent Cracking Engine

3.2 Performance & Concurrency

3.2.1 Fix the Lookup Algorithm

Replace the original O(N*M) lookup with an efficient algorithm (e.g., using a HashSet for O(1) lookups). This would yield a considerable performance increase.

3.2.2 Implement High-Performance Concurrency

The single-threaded model must be replaced with a robust concurrent solution. You must design and justify your task division strategy.

3.2.3 Eliminate Race Conditions

All shared mutable state (e.g., counters for found passwords, hashes computed) must be made thread-safe using atomic classes (AtomicLong, etc.) or other appropriate concurrency controls.

3.3 Code Modernization & Readability

3.3.1 Modernize Legacy APIs

Replace the outdated Java idioms.

- Reduce the boilerplate in data classes.
- Use flexible collections instead of concrete classes (i.e.: Map instead of HashMap).
- Make sure the resources handling is safe and tidy.

3.3.2 Use Modern Java Features

Imperative for loops should be replaced with the Java Streams API and Lambdas where it improves clarity and conciseness.

While replacing imperative for loops with the Java Streams API and lambdas can often be a significant improvement in terms of code clarity and conciseness, it's not a universal rule. The decision to refactor should be made on a case-by-case basis, considering the complexity of the operation and the overall readability

3.4 Feature Enhancement & Security

3.4.1 Implement a Live Status Reporter

The final application must provide non-blocking, status updates to the console. Use the same format and frequency provided in the original code. But the legacy method of printing inside the main loop is not acceptable. Status reporter needs to be run in a separate thread.

4 Instructions

4.1 Group Formation

You've already formed and enrolled in groups on e-learn.

4.2 JDK version

Use JDK 21 for the development (Except for section 5.3).

4.3 Evaluation

You're provided with two data sets 1) small 2) large. Each set contains 1) dictionary.txt 2) in.txt (contains username and hashed password) 3) out.txt (contains username, hashed password and plain password).

You may use the small set to check the correctness of the code and large set to measure the speed. For the grading, a different set of files similar to the size of the large set will be used.

Each team will be provided with Linux VM with same amount of resources. At the evaluation your code will be benchmarked on a similar VM. You are not allowed to change any OS parameters. All the optimizations should be done on JVM or the Java code.

4.4 Submission

You must submit a single .zip archive containing (Check the section 5.3 for additional submission requirements for bonus category):

1. src/: The complete source code for your refactored application in a **Maven** project structure.

2. run.jar: A single runnable JAR file that can be executed from the command line (You may use the provided pom.xml file to build the jar file. Upload the jar with all the dependencies included):

```
java [JVM parameters] -jar run.jar in.txt dictionary.txt out.txt
```

3. REPORT.pdf: A 3-5 pages (A4 sized in 11pt font size, single line spacing) report in pdf format containing:

   (a) Cover Page: Include a cover page that shows the group number and names of all the team members.

   (b) Initial Diagnosis: A brief analysis of the key architectural, performance, security, and legacy flaws you identified in the original monolith.

   (c) Final Architecture: An explanation of your new system design. A high level architectural/component diagram showing the main components and their interactions is required. Use UML diagrams where possible (high level collaboration/sequence diagrams, component diagrams).

   (d) Performance & Concurrency Strategy: A detailed explanation of your approach to making the application fast. Justify your choice of concurrency model, task division, and algorithmic optimizations.

   (e) Appendix:

      i. Complete command (including the JVM parameters) to run the jar file on Linux VM.

      ii. Link to the GitHub repository: You are required to use **git** to manage the source code and work collaboratively on this project. Every team member is expected to code and commit. Teaching team may check the commit history of the project.

      iii. Link to the recorded presentation: Include a recorded presentation (up to 10 min) posted anywhere with a public URL to present your solution. In your presentation, convince viewers that your solution follows good architectural principles/clean code principles & is easily extensible for future changes as well as decisions that improve performance. Assume your viewers include a Tech Lead (very good technical knowledge) and a Business Analyst(minimal technical knowledge). Selected teams will be asked to present in class in week 12. Bonus marks may be given for presenting teams.

   The cover page, appendix, references, and acknowledgments do not count toward the page limit.

   You must acknowledge all code done by 3rd parties, usage of GenAI, usage of ideas from other sources. Reference to use APA7. You may be penalized if you don't follow APA7 format.

5 Grading

This project will contribute 25% to the overall course grade. The grading components are as follows, This is a competitive project. Your final grade is determined by both the performance of your application and the quality of its design.

5.1 Technical Design & Refactoring Score (60 Points)

A fast solution built on a poor design will not score well.

1. Architectural Design & SOLID Principles (**25 Points**): How well did you deconstruct the provided code? Is the new architecture clean, decoupled, and testable?
2. Concurrency Model (**20 Points**): Quality and justification of your concurrency implementation. How did you divide tasks? Is the solution free of race conditions and deadlocks? How was the live status reporter implemented?
3. Memory & Algorithmic Strategy (**15 Points**): How did you solve the OutOfMemoryError? How did you fix the O(N*M) lookup? How well do you explain these optimizations?

5.2 Performance Score (40 Points)

Your program will be executed in the evaluation VM, and has to produce the our.txt file that correctly identifies all crackable passwords, and **must not** report any false positives. The order of rows in out.txt is not important.

Execution Speed: The total time to process in.txt will be benchmarked.

- 1st Place Team: 40 Points
- 2nd Place Team: 36 Points
- 3rd Place Team: 32 Points
- 4th Place Team: 29 Points
- 5th Place Team: 26 Points
- 6th Place Team: 23 Points
- 7th Place Team: 21 Points
- 8th Place Team: 19 Points
- 9th Place Team: 17 Points
- All Other Correctly Finishing Teams: 15 Points

Except for extraordinary situations, all team members will receive the same grade.

5.3 Bonus (5 Points)

To earn bonus points, your team is encouraged to experiment with and integrate features from the newly released JDK 25. To qualify for the bonus,

1. Configure JDK 25 on the provided VM and ensure your application runs successfully.
2. Leverage JDK 25 features to improve performance
3. Document your enhancements clearly in the report,
   (a) Describe each JDK 25 feature used.
   (b) Explain why it was chosen and how it improved your solution.
   (c) Provide before-and-after benchmarks to show measurable performance gains.
4. Upload additional src folder, jar file and java command for JDK 25.

Bonus marks will be awarded based on,

- Depth of exploration.
- Clarity of explanation.
- Demonstrated impact on performance.

5.4 Late Submissions

No extensions will be given for the submission. Late submissions,

- Within 24 hours - 50% penalty of final mark.
- Beyond 24 hours - 0 marks.

6 Guidance & Hints

Strategic Guidance & Hints

1. Order of Operations is Critical: Do not try to parallelize a broken algorithm.
   (a) First, fix the algorithm. Replacing the O(N*M) lookup will give you the single biggest performance boost.
   (b) Second, refactor the architecture. A clean design will make it much easier to add concurrency correctly.
   (c) Third, add concurrency. Parallelize your clean, efficient algorithm.
2. Think About the Bottleneck: The primary bottleneck is CPU (hashing). Your concurrency strategy should focus on keeping all CPU cores 100
3. The path to victory is through superior engineering, not just brute force. Good luck.

7 Note on Use of AI Tools

You are allowed to use Large Language Models (LLMs) such as ChatGPT, Google AI Studio or GitHub Copilot to assist with your project work. However, it is **your responsibility to fully understand** any code or concepts generated using these tools.

You may be asked to **explain your code and design decisions** during reviews or presentations. Failure to demonstrate understanding will impact your grade.

Use the AI tools to learn and accelerate your development—not as a shortcut.

8 Honor Code

SMU expects students to abide by the honor code and The SMU Student Code of Conduct. Most importantly, your answers must be your own! So, do not discuss these specific problems with other teams, seniors etc. Remember, plagiarism and conferring with your peers is academic dishonesty and carries severe consequences.

9 Appendix

9.1 Flow Chart

Figure 1: Main Flow

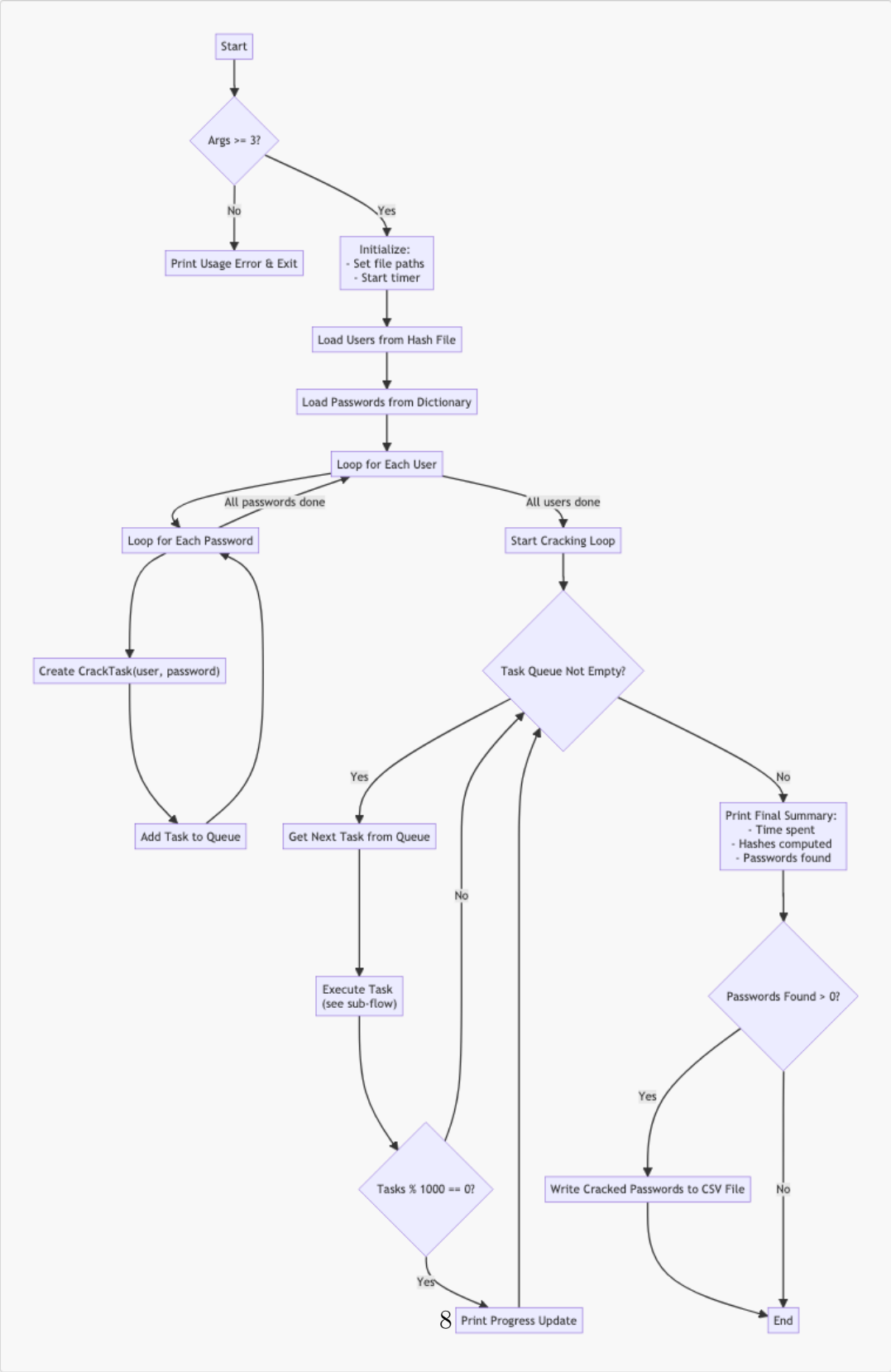

Figure 2: Execute Task Flow