LEBANESE UNIVERSITY
FACULTY OF ENGINEERING III
ELECTRICAL AND ELECTRONIC DEPARTMENT

Parallel Password Hash Cracker — Project Proposal

By

Kassem Elhajj 6360

Mahdi Abou Hamdan 6359

Spring 2024-2025

Supervisor: Dr. Mohammad Aoudi

Performance Evaluation of Sequential and Parallel Password Cracking in Java

1. Introduction

In modern cybersecurity, understanding the performance of brute-force password attacks is essential for designing secure systems. This project explores and benchmarks three password cracking strategies in Java:
- Sequential (single-threaded)
- Recursive Parallel (Fork/Join)
- Full Parallel (Thread Pool)

Each approach targets a numeric 8-digit password formed by combining a 2-digit prefix with a 6-digit suffix. SHA-256 hashing is used for password verification.

2. Hashing Method

All password candidates are verified using SHA-256 hashing via the following method:

```
public static String sha256(String input) {
   MessageDigest md = MessageDigest.getInstance("SHA-256");
   byte[] hash = md.digest(input.getBytes());
   StringBuilder sb = new StringBuilder();
   for (byte b : hash) sb.append(String.format("%02x", b));
   return sb.toString();
}
```

This ensures consistent cryptographic validation across all implementations.

3. Cracking Approaches

3.1 SequentialCracker

- Implements a single-threaded brute-force attack.
- Loops through all prefixes and their 6-digit combinations.
- Acts as the baseline for speed-up comparison.

3.2 ParallelCracker (Fork/Join)

- Implements recursive parallelism using Java's ForkJoinPool.
- Splits prefix array into smaller chunks recursively.
- Efficient CPU usage, with dynamic work stealing.

3.3 FullParallelCracker (ExecutorService)

- Assigns one thread per prefix.
- Uses a fixed-size thread pool (Executors.newFixedThreadPool()).
- Each prefix is independently processed.
- Offers the best raw performance due to independence.

4. Experimental Setup

Parameter	Value
Password Example	84429446
Prefix Set	{"03", "70", "71", ...}
SHA Algorithm	SHA-256
JVM Threads Used	up to 12 (depending on CPU)
Runs per Strategy	5

System Specs:

- CPU: 12 Logical Cores
- RAM: 16 GB
- OS: Windows/Linux
- Java: OpenJDK 17+

5. Results Summary

Average of 5 Runs:

Strategy	Password Found	Avg Time (s)	Avg Mem (MB)	Avg CPU(%)
Sequential	84429446	26.580	75.45	25.56
Parallel (FJ)	84429446	8.501	316.23	40.58
Full Parallel	84429446	7.431	219.67	41.89

6. Performance Analysis

- Recursive Parallel reduced runtime by ~63% over sequential.
- Full Parallel achieved the fastest performance due to maximum thread utilization.
- Fork/Join provided dynamic load balancing but had more overhead than fixed thread pool.
- CPU usage was highest in Full Parallel due to complete core saturation.

7. Speed-Up Calculations

Strategy	Speed-Up over Sequential
Parallel (FJ)	~3.13×
Full Parallel	~3.58×

8. Implementation Snippets

FullParallelCracker task:

```
Callable<String> task = () -> {
  for (int i = 0; i <= 999999; i++) {
    String candidate = prefix + String.format("%06d", i);
    if (HashUtil.sha256(candidate).equals(targetHash)) return
candidate;
  }
  return null;
};
```

Fork/Join Tree:

```
crack(0,10)
├── fork -> crack(0,5)
└── compute -> crack(5,10)
```

9. About Rainbow Tables

A rainbow table is a precomputed database of password–hash pairs. Attackers use it to reverse hashes without recalculating them.

However, they become useless when:
- Passwords are salted
- Hashing is slow (e.g., bcrypt, PBKDF2)

Our brute-force strategy is not rainbow-table-based; it dynamically hashes candidates at runtime.

10. Summary Report

This project includes:
- Runtime measurements
- CPU time and memory usage
- Timestamped logs
- Automatically written to CrackReport.pdf

11. Conclusion

- Brute-force attacks are CPU-heavy but highly parallelizable.
- Recursive parallelism offers good balance.
- Full parallel execution is optimal for uniform and independent tasks.

- Java's concurrency tools (ForkJoinPool, ExecutorService) are powerful for performance-critical applications.

GITHUB: https://github.com/Kassem-Elhajj/ParallelPasswordCracker

12. Future Improvements

- Add early termination signals to cancel threads faster.
- Try GPU-based parallel hashing (e.g., via OpenCL or CUDA).
- Benchmark with non-numeric passwords and larger search spaces.
- Add progress bar or real-time monitoring.

Appendix

A. Prefix Set Used

"03", "70", "71", "76", "78", "79", "81", "82", "83", "84"

B. Sample Hash

SHA-256("84429446") =
8b7a0dbdd4cd2b01aef7d9ab88a1f0cbb2a711d94b5efabe8f6f152b494f4f7d

13. Why we didn't use Rainbow table:

Rainbow tables are optimized for fast password lookup by storing massive precomputed databases of hashes.
But in our project:

- We're cracking a password with an **unknown hash**, but from a **very limited and numeric-only space** (prefix + 6 digits).
- It was simpler and more educational to **generate hashes on the fly** and focus on:
  - Sequential vs. parallel computation
  - CPU and memory usage
  - Speed-up with threads