Password Decryption

Parallel Computing 2025-2026

Mathilde Patrissi
mathilde.patrissi@edu.unifi.it

Abstract

This project provides a comparative analysis of computational performance when employing a brute-force approach to decrypt an eight-character password encrypted with the Data Encryption Standard (DES) which is a symmetric-key algorithm for the encryption of data. By utilizing both sequential and parallel implementations using OpenMP, the project evaluates the efficiency gains achieved through parallelization.

1 Introduction

This study focuses on decrypting an eight-character password from the [a-zA-Z0-9./] character set. The methodology employs a dictionary-based attack that iteratively hashes entries from a common password database using the Data Encryption Standard (DES). These generated hashes are then compared against the target hash to find a match. This implementation relies on the POSIX crypt() function, a standard for password hashing in Unix-based system.

1.1 Data Encryption Standard

The Data Encryption Standard (DES) is a symmetric-key block cipher designed to process data in 64-bit blocks. While the initial key is provided as 64 bits, only 56 bits are utilized for encryption, as 8 bits are reserved for parity checking to ensure key integrity. The algorithm's core security is built upon a 16-round Feistel network, which iteratively applies non-linear substitutions through S-boxes and bitwise permutations to achieve high levels of confusion and diffusion.
For this project, the encryption logic is handled by the POSIX crypt() function, which implements this traditional DES-based hashing scheme.
The crypt function takes a password, key, as a string, and a salt character array which is described below, and returns a printable ASCII string which starts with a salt. Because it is a one-way hash function, identifying the correct input requires an exhaustive key search, or brute-force attack, where candidate keys are iteratively hashed and compared against the target output until a match is found.
The salt parameter serves two primary cryptographic functions. First, it acts as a structural identifier, determining whether the system employs the legacy DES-based cipher or the MD5-based hashing algorithm based on its prefix. Second, it serves as a defense-in-depth mechanism against bulk password cracking. By introducing a salt for each entry, an attacker is prevented from using a single computed hash to verify multiple passwords simultaneously. Instead, they are forced to perform a unique cryptographic calculation for every distinct salt in the target file, significantly increasing the computational overhead required for a successful attack.
For the MD5-based variant, the salt must begin with the 1 identifier, followed by an optional string of up to eight characters and terminated by either a second $ or the end of the string. The resulting crypt() output concatenates the salt (appending a $ if one was absent) with a 22-character hash derived from the ./0-9A-Za-z alphabet. This yields a final string of up to 34 characters. Unlike legacy DES implementations, every character in the input key is cryptographically significant in the MD5-based process.
In the DES-based implementation, the salt is restricted to a two-character string derived from the ./0-9A-Za-z alphabet. The resulting output is a 13-character string, comprising the original salt followed by an 11-character hash. A critical limitation of this method is that only the first eight characters of the key are processed; any subsequent characters are discarded. Conversely, the MD5-based algorithm supports arbitrary password lengths and offers enhanced cryptographic security, making it the preferred standard. For initial password creation, a cryptographically random salt should be generated. To verify subsequent login attempts, the previously generated hash string is passed back as the salt parameter to ensure a matching output.
The traditional C/Unix crypt() function typically returns a pointer to a statically allocated buffer. This design makes

the function fundamentally thread-unsafe due to shared resource interference. In a multi-threaded environment, concurrent calls to crypt() will attempt to write their output to the same memory address simultaneously. This creates a race condition where one thread can overwrite the results of another before they are retrieved, leading to data corruption and non-deterministic hashing results.
To mitigate thread interference, this project uses crypt_r(), the reentrant and thread-safe counterpart to the standard crypt() function. Unlike the original version, crypt_r() requires an additional argument—a pointer to a locally allocated buffer provided by the calling thread. By ensuring that each thread manages its own dedicated memory space for results, the implementation eliminates reliance on shared static buffers and prevents data corruption during parallel execution.

1.2 Dataset

Given that our project targets an eight-character password space within the specific [a-zA-Z0-9./] character set, we implemented a dedicated preprocessing pipeline for the RockYou dataset.

RockYou.txt:
Originating from a 2009 security breach, the RockYou.txt wordlist was born when the RockYou social application suffered a cyberattack exposing over 32 million credentials. The gravity of the leak was intensified by the fact that passwords were stored in plaintext—a fundamental security failure. Today, this dataset is an industry-standard resource for penetration testers and IT administrators. By employing this list within tools like Hashcat or John the Ripper, security experts can simulate real-world attacks to identify authentication vulnerabilities and enforce stronger password policies.

The preprocessing stage involved a three-part sanitization routine:

- **Data Ingestion:**
  Reading the source RockYou.txt file;
- **Validation:**
  Retaining only those strings that precisely match the 8-character length and use only characters from the [a-zA-Z0-9./] alphabet;
- **Deduplication:**
  Filtering out any redundant passwords to create a unique, optimized dictionary for the attack.

The refined password entries are preserved in a dedicated input file (password_file.txt), which serves as the standardized foundation for our decryption algorithms. By executing this preprocessing phase during the initial setup, we significantly optimize the dataset's size and relevance. This architectural choice not only enhances computational efficiency during the attack phase but also guarantees uniform testing conditions: an essential requirement for the comparative performance analysis between our sequential and parallel OpenMP implementations.

2 Proposed solutions

To address the password decryption problem, this project develops a comparative study between sequential and parallel computational models. All implementations are written in the C++ programming language. The proposed methodologies consist of the following distinct versions:

1. Sequential: A single-threaded baseline implementation.

2. Parallel OMP: A multi-threaded solution utilizing OpenMP directives for concurrent execution.

The system provides the option to choose the target password for the testing phase through an initial menu, which scrolls through the available passwords in the filtered dataset.
An example is shown below:

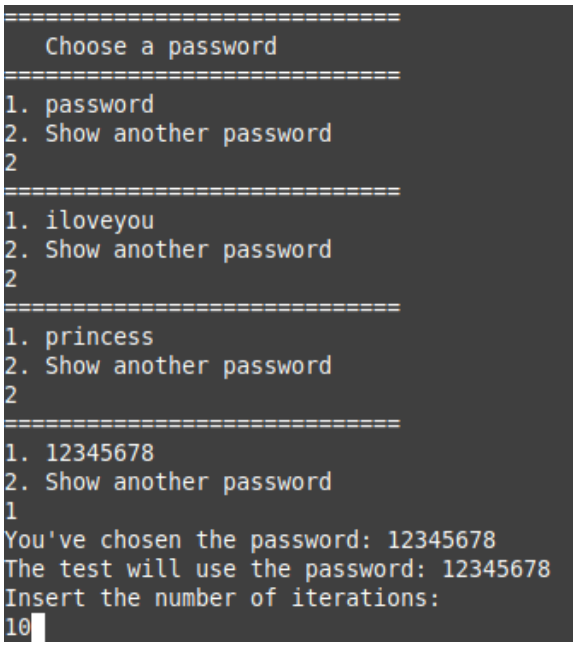

Figure 1: Initial choice

2.1 Sequential

The sequential implementation serves as the primary benchmark for our password decryption system. By processing candidate passwords linearly, this non-parallelized version establishes a functional baseline necessary for quantifying the performance gains achieved through parallelization. The core decryption logic is structured as follows:

```
int checkPwd(const std::string&
    crypted_password,const std::string&
    salt, const vector<std::string>&
    passwordList){
  std::string pwddecrypted;
  int pos=1;
  for (const auto& word : passwordList) {
      pwddecrypted =  cryptDES(word,salt);
      if (pwddecrypted ==
          crypted_password) {
        return pos;
      }
      pos++;
  }
  return -1;
}
```

In this version, the system iterates through the filtered dataset sequentially, leveraging the crypt_r() function to perform DES-based encryption. This process involves a distinct key schedule initialization followed by the encryption operation using a predefined salt. Because this architecture is inherently linear, the total execution time scales proportionally with the size of the password list. Because the execution flow is strictly serial, each cryptographic operation must terminate before the subsequent one can initiate.

This architecture fails to leverage the multi-core capabilities of modern processors, leading to significant underutilization of hardware resources. While this linear approach ensures operational reliability, its scalability is poor; the resulting computational inefficiency becomes a primary bottleneck as the dataset expands.

2.2 Parallel

The parallel implementations aim to optimize algorithmic performance through high-concurrency execution.

The following OpenMP version overcomes the limitations of the sequential model by utilizing multi-threading to process password candidates simultaneously. This approach shifts from a linear execution flow to a distributed workload model, significantly increasing throughput.

The structural logic of this implementation is detailed in the next snippet.

```
std::string ompDecryption(const
    std::string& encrypted_password, const
    std::string& salt,
    const std::vector<std::string>&
        pwdList, int threadNum, int
        chunkSize) {
    std::string word = "";
    bool found = false;
#pragma omp parallel
    num_threads(threadNum) shared(found,
    word)
    {
        std::string local_result;
#pragma omp for schedule(dynamic,
    chunkSize)
        for (int i = 0; i <
            static_cast<int>(pwdList.size());
            ++i) {
            if (!found) {
                std::string hash =
                    cryptDES(pwdList[i],
                    salt);

                if (hash ==
                    encrypted_password) {
#pragma omp critical
                    {
                        if (!found) {
                            found = true;
                            word =
                                pwdList[i];
                        }
                    }
                }
            }
        }
    }
    return word;
}
```

As demonstrated in the previous snippet, a dynamic scheduling strategy is employed to achieve optimal load balancing. This is particularly vital in scenarios where early termination (triggered by a successful match) creates non-uniform processing times across threads. To maintain data integrity, the #pragma omp critical directive is used to serialize updates to shared variables, preventing race conditions when a thread modifies the global state. Additionally, the implementation relies on thread-privatization; by defining variables inside the parallel loop, each thread operates on its own memory copy. This eliminates data races and ensures that the final result is determined accurately and efficiently.

Managing the shared state is a critical challenge within the parallel implementation. The *found* flag and the decrypted password variable are shared across all threads, necessitat-

ing strict synchronization via critical sections to mitigate race conditions. Once a thread identifies a match, it triggers a global signal for early termination, thereby optimizing computational throughput.
Lastly, dynamic scheduling is employed to improve load balancing because, first of all, in a real operating system, other processes can consume CPU resources unpredictably. This is a case of external interference: if a core is temporarily occupied by another system task, the threads assigned to that core would be suspended, while the others would continue their work. Furthermore, there is the situation where threads share the same core when their number exceeds the number of physical cores. This would result in different execution times for the threads.

3 Experimental setup and results

The following analysis describes the testing environment and the experimental procedures used to evaluate our sequential and parallel implementations. To provide a consistent baseline for performance analysis, all tests were carried out on a system with the following hardware specification:

- CPU: Intel Core i7 CPU 870 (4 cores - 8 threads, 2.93GHz)
- RAM: 32GB (4x8GB) DDR3-1600MHz PC3-12800
- Storage sda1: SSD 240GB KINGSTON A400
- Storage sdb1: SSD 240GB KINGSTON A400
- Storage sdc1: SSD 240GB KINGSTON A400
- Operating System: Linux Mint 20.3 MATE 1.26.0

3.1 Configuration and metrics

The experimental framework utilized a set of predefined target passwords to assess the application's decryption capabilities. To ensure statistical rigor and mitigate the impact of transient computational anomalies, each test iteration was performed ten times. The benchmarking protocol was structured to evaluate performance across diverse search scenarios by systematically adjusting the target password's position within the filtered dataset.
For each test cycle, the target was injected at positions derived from the following calculations:

$$position_i = \frac{pwdList\_size}{iterations} * i + \frac{pwdList\_size}{iterations * 2} \quad (1)$$

where *i* ranges from *0* to *iterations-1*.
This methodology ensures a uniform distribution of test cases throughout the search space, providing a comprehensive assessment of the algorithm's performance.

The application's performance is analyzed using the following metrics:

- **Temporal Metrics Time:**
  Minimum, maximum, and mean execution latency to establish a comprehensive timing profile;
- **Standard Deviation:**
  Utilized to quantify performance stability and ensure the reliability of the benchmarking environment.

3.2 Parallel Execution Parameters

The parallel version was tested under multiple configurations to identify the most efficient execution parameters, focusing on:

Thread Count Analysis: Aligning with our system's hardware architecture, we evaluated the implementation using various thread counts to conduct a rigorous scalability analysis. This range facilitates the observation of performance progression from initial parallelization to peak hardware thread saturation, allowing us to quantify the efficiency gains across different levels of concurrency.

Chunk Size Optimization: Various chunk sizes were evaluated to find the best balance between scheduling overhead and workload distribution.

- **Small Chunk Size** (500 passwords):
  This configuration provides high granularity, ensuring a responsive load balance;
- **Medium Chunk Size** (1000, 2000 passwords):
  This range represents a balanced configuration, effectively harmonizing the benefits of dynamic load balancing;
- **Large chunks Size** (4000 passwords):
  This approach prioritizes throughput by lowering the cost of thread orchestration. While it reduces the time spent on task assignment, it is less responsive to dynamic load balancing, which can result in some cores being underutilized during the final stages of the decryption cycle.

Adjusting the chunk size is critical when employing a dynamic scheduling strategy, as it directly governs the granularity of workload distribution across the thread pool. Small chunks maximize the system's ability to redistribute work but can lead to scheduling latency. Conversely, larger chunks mitigate this overhead but introduce the risk of load imbalance.

Performance Metrics : Performance data were captured using high-resolution temporal primitives (std::chrono::high_resolution_clock) and stored in a structured format for comprehensive analysis.
For the parallel execution the following parallel performance metrics were calculated:

- **Speedup:**
  The ratio of sequential execution time to parallel execution time, quantifying the performance gain;
- **Efficiency:**
  The ratio of speedup to the total number of threads, indicating the effective utilization of each processor core;
- **Granularity Analysis:**
  Assessment of the impact of chunk size on total execution latency and task distribution;
- **Scaling Efficiency:**
  Analysis of the system's ability to maintain performance as thread concurrency increases.

This testing methodology enables a multifaceted analysis of both absolute computational throughput and the efficacy of the parallelization strategy across diverse execution environments.

3.3 Sequential implementation

The sequential implementation serves as the performance baseline for this study. Experimental results indicate that execution times vary significantly depending on the password's position within the dataset:

Metric	Value
Mean Time (s)	9.5233s
Standard Deviation	5.3261s
Minimum Time (s)	0.9806s
Maximum Time (s)	17.8840s
Total Passwords	2965748

Table 1: Sequential Implementation Performance Metrics.

Execution times for the sequential implementation vary significantly, ranging from a few milliseconds to several seconds. This discrepancy stems from the linear nature of the search: finding a password at the beginning of the list allows for early termination, whereas searching toward the end requires a near-exhaustive traversal of the dataset.

3.4 OpenMP parallel implementation

The OpenMP implementation leverages shared-memory parallelism by distributing the image processing workload across multiple CPU threads. Evaluation of the parallel system involved testing various thread counts and chunk sizes, with key performance metrics detailed in Table 2. The resulting data highlights several notable patterns:

Scalability Profile: The system exhibits significant speedup as the thread count increases, reaching a peak performance gain of 4.5351. However, when increasing the number of thread over four, we observed a very small speedup improvement and a decrease of the efficiency, as shown in table 2. This degradation is likely attributable to resource contention, as the 8-thread workload exceeds the 4-core physical capacity of our processor.

Temporal Performance Analysis: Mean execution time follows a downward trend, decreasing from 9.5233s in the sequential baseline to a minimum of 2.0998s with eight threads. While this represents a substantial performance gain, the scaling is non-linear which confirms the presence of a little parallel overhead.

Efficiency Evaluation: The parallel efficiency (defined as speedup per thread) exhibits a downward trend as thread counts increase, dropping from 0.9669 at two threads to 0.5668 at eight threads. This highlights a classic scaling trade-off: although the total execution time is reduced, the marginal performance gain per thread diminishes, leading to less efficient use of the available CPU resources
To evaluate the performance profile of this implementation, an analysis of execution time patterns across various thread configurations was conducted. The comparison between sequential and parallel runtimes reveals significant insights into the scaling behavior.

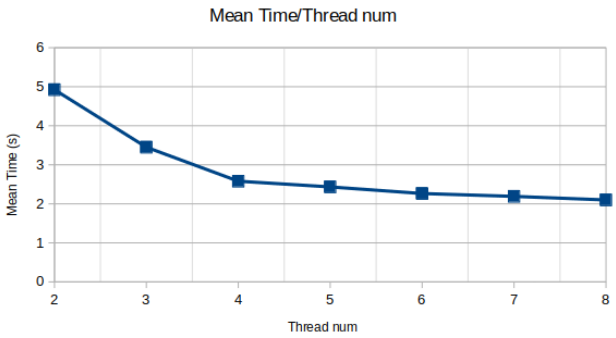

Figure 2: Mean execution time - number of thread

Figure n.2 illustrates the relationship between mean execution time and thread count, highlighting the performance gains achieved through parallelization. The data demonstrates a significant reduction in execution time

as the thread count increases, with the most substantial improvement observed between the sequential version and the four-thread configuration.
The efficiency of the parallel implementation is further evaluated through the speedup analysis in the following table and figure n.3.

num. of thread	speedup	ideal speedup	efficiency
2	1.9338	2	0.9669
3	2.7599	3	0.9199
4	3.6919	4	0.9229
5	3.9149	5	0.7829
6	4.2067	6	0.7011
7	4.3467	7	0.6209
8	4.5351	8	0.5668

Table 2: Speedup and efficiency - chunckSize 500

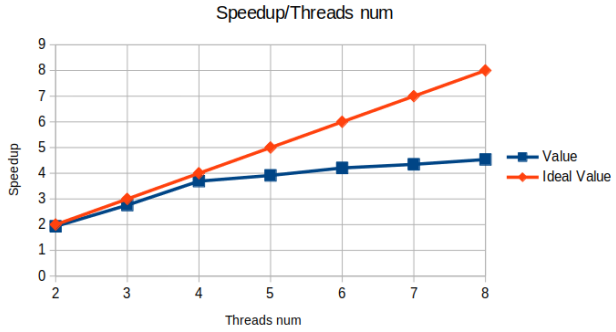

Figure 3: Speedup - number of thread

This visualization compares the measured speedup against the theoretical linear ideal, providing insights into the system's scalability.
The results indicate that while significant speedup is attained with more threads, efficiency begins to decrease at five threads.

Chunk Size Impact Analysis: To rigorously assess the impact of data granularity on system throughput, a systematic evaluation of various chunk sizes (500, 1,000, 2,000, and 4,000) was conducted.
This analysis focuses on identifying fluctuations across key performance metrics, the results of which are comprehensively detailed in the following tables.
Empirical data reveals how chunk size affects overall efficiency, specifically showing its impact on performance across different thread counts.

chunk	meanTime	stdDev	maxTime	minTime
500	4.9246s	2.8184s	9.3420s	0.4996s
1000	4.9225s	2.8259s	9.3204s	0.4941s
2000	4.9172s	2.8276s	9.3852s	0.4954s
4000	4.9245s	2.8236s	9.3359	0.4870s

Table 3: Parallel Implementation Performance - 2 threads.

chunk	meanTime	stdDev	maxTime	minTime
500	3.4505s	1.9799s	6.5387s	0.3461s
1000	3.4502s	1.9823s	6.5670s	0.3463s
2000	3.4462s	1.9791s	6.5467s	0.3423s
4000	3.4464s	1.9773s	6.5333s	0.3423s

Table 4: Parallel Implementation Performance - 3 threads.

chunk	meanTime	stdDev	maxTime	minTime
500	2.5795s	1.4790s	4.8952s	0.2603s
1000	2.7231s	1.6122s	5.0258s	0.2616s
2000	2.5804s	1.4793s	4.8975s	0.2556s
4000	2.5807s	1.4877s	4.9437s	0.2546s

Table 5: Parallel Implementation Performance - 4 threads.

chunk	meanTime	stdDev	maxTime	minTime
500	2.4325s	1.4072s	4.7290s	0.2438s
1000	2.4160s	1.3812s	4.5833s	0.2421s
2000	2.4121s	1.3848s	4.5924s	0.2418s
4000	2.4252s	1.4038s	4.5717s	0.2296s

Table 6: Parallel Implementation Performance - 5 threads.

chunk	meanTime	stdDev	maxTime	minTime
500	2.2638	1.3013s	4.3290s	0.2305s
1000	2.2998s	1.3144s	4.3452s	0.2338s
2000	2.3211s	1.3403s	4.4216s	0.2267s
4000	2.3186s	1.3258s	4.3369s	0.2269s

Table 7: Parallel Implementation Performance - 6 threads.

chunk	meanTime	stdDev	maxTime	minTime
500	2.1909s	1.2592s	4.1605s	0.2157s
1000	2.1956s	1.2643s	4.1516s	0.2202s
2000	2.1907s	1.2620s	4.1379	0.2118s
4000	2.2121s	1.2634s	4.1816s	0.2139s

Table 8: Parallel Implementation Performance - 7 threads.

chunk	meanTime	stdDev	maxTime	minTime
500	2.0998s	1.2063s	3.9896s	0.2114s
1000	2.1045s	1.2098s	4.0070s	0.2121s
2000	2.1032s	1.2108s	4.0073s	0.2068s
4000	2.1038s	1.2095s	3.9990s	0.1849s

Table 9: Parallel Implementation Performance - 8 threads.

Figure n.4 illustrates the impact of chunk size on performance across various thread configurations.

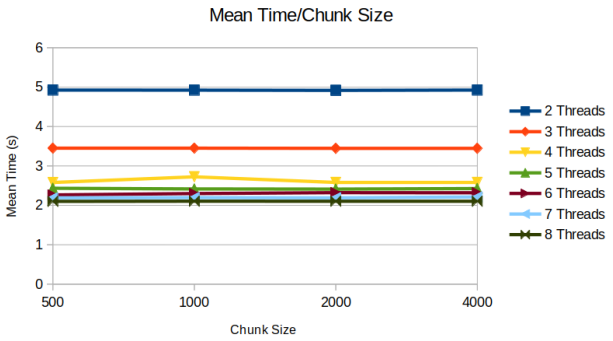

Figure 4: Mean Time - Chunk Size

Experimental results indicate that the impact of chunk size on performance is often marginal; however, the most stable and efficient results across various test cycles were observed using medium-sized chunks (1,000–2,000 passwords).
Regardless of the setup, decryption remains the most expensive operation, making the choice of chunk size almost inconsequential.
Because the system spends the vast majority of its resources on processing each unit, the impact of how those units are grouped is nearly invisible in the final results.

4 Conclusions

This project involved the development and analysis of sequential and parallel implementations for a DES-based password decryption system. From a computational complexity perspective, the sequential algorithm exhibits O(n) complexity, where n represents the number of entries in the dataset. Theoretically, under ideal parallelization conditions, the parallel implementation should achieve O(n/p) complexity, where p denotes the thread count. However, the experimental results reveal a more nuanced reality, highlighting the impact of real-world overhead and hardware constraints. Significant performance gains were achieved through parallelization, reaching a peak speedup of 4.5351 with eight threads. However, these results remain below the theoretical linear ideal. This discrepancy is largely attributable to the inherent sequential constraints of the DES implementation. This degradation is likely attributable to resource contention, as the 8-thread workload exceeds the 4-core physical capacity of our processor. Furthermore, despite optimizations in parallel workflow (including dynamic scheduling and meticulous thread management) the underlying library architecture remains a primary bottleneck. In addition, an evaluation of task granularity highlights the role of chunk size in performance optimization. The high cost inherent in decryption operations acts as a performance leveler, marginalizing the influence of workload granularity. Consequently, while medium-sized chunks (1,000–2,000 passwords) technically offer the best theoretical balance, the sheer weight of the decryption process ensures that any variations in chunk size remain statistically insignificant to the overall execution time. The performance plateau observed when scaling beyond four threads highlights how sequential components within the DES operations increasingly dominate execution time as parallelization expands. These findings demonstrate that while parallelization strategies provide substantial gains, the inherent characteristics of the chosen cryptographic library and the tuning of execution parameters remain decisive factors. Consequently, future research should evaluate alternative libraries optimized for parallel architectures or explore encryption algorithms with higher inherent concurrency.

References

[1] Material of Parallel Computing cource - University of Florence - 2025/2026 - Professor Bertini Marco

[2] Programming on Parallel Machines - Norm Matloff, University of California, Davis - GPU, Multicore, Clusters and More

[3] Parallel ProgrammingFor Multicore and Cluster Systems - Thomas Rauber, Gudula R¤unger - Springer-Verlag Berlin Heidelberg 2010

[4] OpenMP Programming Guide.
https://www.openmp.org/resources/refguides/

[5] RockYou Dataset Documentation.
https://www.kaggle.com/datasets/wjburns/commonpassword-list-rockyoutxt

[6] RockYou2024.txt— 180 Million ”Strong Passwords”
https://www.kaggle.com/datasets/bwandowando/strong-passwords-in-rockyou2024-txt

[7] Understanding RockYou.txt: A Tool for Security and a Weapon for Hackers
https://www.keepersecurity.com/blog/2023/08/04/understanding-rockyou-txt-a-tool-for- security-and-a-weapon-for-hackers/

[8] RockYou: An Example of Wordlists in Cybersecurity
https://www.packetlabs.net/posts/wordlists-in-cybersecurity/

[9] Password Pattern and Vulnerability Analysis for Web and Mobile Applications, hancang Li, Imed Romdhani, and William Buchanan School of Computing, Edinburgh Napier University, Edinburgh EH10 5DT, Scotland, UK
https://uwe-repository.worktribe.com/OutputFile/905982

[10] Tool Documentation: Wordlists Usage Examples,
https://www.kali.org/tools/wordlists/