PC-2024/25 Password Decryption

Alessio Bugetti
7132744
alessio.bugetti@edu.unifi.it

Abstract

This report details the implementation of a brute-force password decryptor targeting passwords encrypted with the Data Encryption Standard (DES). Three versions of the decryptor were developed: one sequential implementation and two parallel implementations using OpenMP and PThread. The average execution times for each version were measured, and the speedup achieved by the parallel implementations was calculated in comparison to the sequential version. For benchmarking, the Rockyou database was utilized, extracting only 8-character passwords composed of the alphabet [a-zA-Z0-9./].

1. Introduction

The goal of this project is to design and evaluate a brute-force password decryption system targeting passwords encrypted with the Data Encryption Standard (DES) algorithm. By leveraging parallel computing techniques, the project seeks to accelerate the decryption process and measure the speedup achieved by different parallelization approaches. DES, one of the earliest widely adopted encryption algorithms, uses a symmetric 56-bit key and operates on 64-bit data blocks [4]. Despite its historical significance, DES's relatively short key length makes it susceptible to brute-force decryption with modern computational capabilities, especially when parallelization is employed.

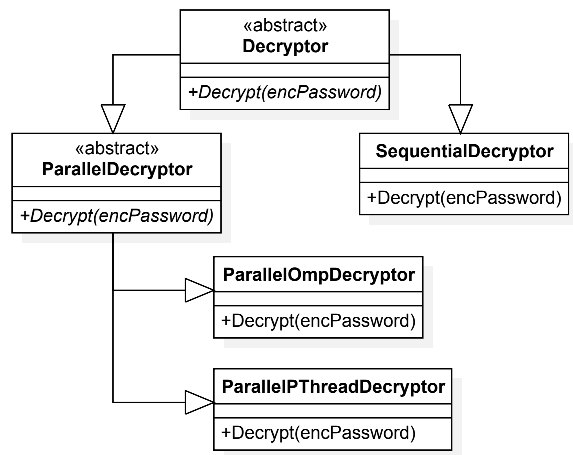

Figure 1. UML class diagram of the defined classes.

2. Project Structure

As shown in the UML class diagram in Figure 1, the project comprises an abstract class `Decryptor`, which exposes the virtual method `Decrypt`, implemented by all concrete classes defining the password decryptors. The sequential decryption is handled by the class `SequentialDecrypter`, while parallel decryption using the OpenMP framework[3] and PThread[2] is implemented in the classes `ParallelOmpDecrypter` and `ParallelPThreadDecrypter`, respectively. Unit tests were created using Google Tests to validate and support the development of the decryptors.

The project supports both Linux and macOS platforms with some differences. For parallel decryption, the `crypt_r` function, a thread-safe version of `crypt`, is used on Linux. This function generates an encrypted password from

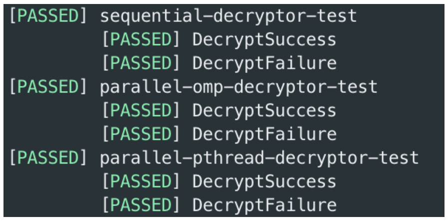

Figure 2. Output showing all tests passing successfully.

a plaintext password (key) and a salt. The salt serves two purposes: generating a more secure encrypted password and selecting the specific algorithm used for encryption. For DES-based encryption, the salt consists of two characters from the `[./0-9A-Za-z]` alphabet, followed by 11 more characters of the same alphabet, resulting in a 13-character encrypted password.

`crypt_r` takes an additional user-defined data structure that must be initialized to `0` before its first use. The result is stored in this structure, making it thread-safe, unlike `crypt`. A sample usage is as follows:

```
struct crypt_data data;
data.initialized = 0;
char *enc = crypt_r(key, salt, &data);
```

Since `crypt_r` is a GNU extension and not part of the POSIX standard, it is unavailable on all UNIX-like systems, including macOS. On macOS, the thread-safe `DES_fcrypt` function from *OpenSSL* is used, which also employs a user-defined structure to store results.

The codebase is fully documented using *Doxygen*, enabling easily accessible documentation of classes and methods.

Two scripts were developed: one for automating test execution and another for automating project compilation via *Ninja*, documentation generation, and execution of the generated binaries. Specifically:

```
./test.py
```

executes all defined tests, outputting the results. Figure 2 shows an example where all tests pass successfully.

Additionally, the project is compiled using *Ninja*:

```
./passwordcracker build
```

Once compiled, the binaries can be executed directly, with the executable names matching the source file names. For instance, to execute a file named `example.cc`, compile the project first, and then run the corresponding binary:

```
./passwordcracker run example
```

Documentation is compiled using:

```
./passwordcracker doc
```

2.1. Sequential Implementation

The sequential implementation of the brute-force attack is straightforward, involving a `for` loop that iterates over a list of the most common 8-character passwords in the chosen alphabet. For each password, DES encryption is performed and compared with the target encrypted password. If a match is found, the password is successfully decrypted, and a tuple containing `true` and the decrypted password is returned. If the list is exhausted without finding a match, a tuple containing `false` and an empty string is returned. The specific code is as follows:

```
const std::vector<std::string> &passwords =
    GetPasswords();
const int numPasswords = passwords.size();
std::string salt = encryptedPassword.substr(0,
    2);

#ifdef __linux__
struct crypt_data data;
data.initialized = 0;
#else
char data[14] = {0};
#endif

for (int index = 0; index < numPasswords;
    index++) {
#ifdef __linux__
  std::string encryptedTmpPassword =
      crypt_r(passwords[index].c_str(),
          salt.c_str(), &data);
#else
  std::string encryptedTmpPassword =
      DES_fcrypt(passwords[index].c_str(),
          salt.c_str(), data);
#endif
  if (encryptedTmpPassword == encryptedPassword)
      {
    return {true, passwords[index]};
  }
}

return {false, ""};
```

The `#ifdef` directive ensures that `crypt_r` is used on Linux, while `DES_fcrypt` is used on macOS.

2.2. Parallel Implementation using OpenMP

This problem is embarrassingly parallel, making it well-suited for parallelization with *OpenMP*:

```
const std::vector<std::string> &passwords =
    GetPasswords();
int numPasswords = passwords.size();
std::string salt = encryptedPassword.substr(0,
    2);

int index = -1;

int numThreads = GetNumThreads();

#pragma omp parallel default(none) shared(index,
    passwords)                          \
    firstprivate(encryptedPassword,
        numPasswords, salt)
                                    \
    num_threads(numThreads)
{
#ifdef __linux__
  struct crypt_data cryptBuffer;
  cryptBuffer.initialized = 0;
#else
  char cryptBuffer[14] = {0};
#endif
#pragma omp for
  for (int i = 0; i < numPasswords; i++) {
#ifdef __linux__
    std::string encryptedTmpPassword =
        crypt_r(passwords[i].c_str(),
            salt.c_str(), &cryptBuffer);
#else
    std::string encryptedTmpPassword =
        DES_fcrypt(passwords[i].c_str(),
            salt.c_str(), cryptBuffer);
#endif

    if (encryptedTmpPassword ==
        encryptedPassword) {
#pragma omp atomic write
      index = i;
#pragma omp cancel for
    }
#pragma omp cancellation point for
  }
}

if (index == -1) {
  return {false, ""};
} else {
  return {true, passwords[index]};
}
```

The parallel implementation uses the `default(none)` clause, a good programming practice that avoids the default behavior of treating all variables as shared. Here, only `index` (storing the decrypted password's index) and `passwords` (the candidate password list) are declared shared.

Another notable feature is the use of the `omp cancel` directive introduced in OpenMP 4.0. This directive allows termination of the loop as soon as a thread decrypts the password. The `omp cancellation point` serves as a checkpoint, terminating the loop if the cancellation flag is set. The environment variable `OMP_CANCELLATION` is automatically set to `true` when running executables via the `passwordcracker` script.

2.3. Parallel Implementation using PThread

The PThread-based parallel brute-force implementation uses data parallelism. The password list `passwords` is divided into chunks based on the number of threads. For instance, with 4 threads, the list is ideally divided into 4 equal parts. Each thread processes a specific portion of the list and updates the shared `index` variable upon successfully decrypting the password, signaling other threads to stop searching.

Each thread is provided with a specific data chunk using a `struct` defined as:

```
struct ThreadData {
  const std::vector<std::string> *passwords;
  const std::string *encryptedPassword;
  const std::string *salt;
  std::atomic<int> *index;
  int start;
  int end;
};
```

This structure includes pointers to the full password list, the target encrypted password, the salt, and the shared `index` variable, as well as two integers marking the thread's data chunk boundaries.

The `DecryptWorker` method attempts password decryption, with the following implementation:

```
ThreadData *data = static_cast<ThreadData
    *>(arg);

#ifdef __linux__
struct crypt_data cryptBuffer;
```

```
cryptBuffer.initialized = 0;
#else
char cryptBuffer[14] = {0};
#endif

for (int i = data->start; i < data->end; ++i) {
  if (data->index->load() != -1)
    return nullptr;

#ifdef __linux__
  std::string encryptedTmpPassword =
      crypt_r((*data->passwords)[i].c_str(),
          data->salt->c_str(), &cryptBuffer);
#else
  std::string encryptedTmpPassword = DES_fcrypt(
      (*data->passwords)[i].c_str(),
          data->salt->c_str(), cryptBuffer);
#endif

  if (encryptedTmpPassword ==
      *data->encryptedPassword) {
    data->index->store(i);
    return nullptr;
  }
}

return nullptr;
```

Threads are created and managed in the `Decrypt` function:

- Candidate passwords are divided into nearly equal chunks based on the thread count
- Each thread is initialized with its chunk and executes `DecryptWorker` via `pthread_create`
- The program waits for all threads to finish using `pthread_join`, ensuring synchronization

The code for the `Decrypt` function is provided below:

```
const std::vector<std::string> &passwords =
    GetPasswords();
int numPasswords = passwords.size();
std::string salt = encryptedPassword.substr(0,
    2);

std::atomic<int> index(-1);
int numThreads = GetNumThreads();
std::vector<pthread_t> threads(numThreads);
std::vector<ThreadData> threadData(numThreads);

int chunkSize = (numPasswords + numThreads - 1)
    / numThreads;

for (int i = 0; i < numThreads; ++i) {
  int start = i * chunkSize;
  int end = std::min(start + chunkSize,
      numPasswords);
  threadData[i] = {&passwords,
      &encryptedPassword, &salt, &index, start,
      end};
  pthread_create(&threads[i], nullptr,
      DecryptWorker, &threadData[i]);
}

for (pthread_t &thread : threads) {
  pthread_join(thread, nullptr);
}

if (index.load() == -1) {
  return {false, ""};
} else {
  return {true, passwords[index.load()]};
}
```

3. Benchmarking and Performance Analysis

To perform the benchmarks, specific passwords were attacked at different positions in a list of commonly used passwords of 8-character length from the alphabet `[a-zA-Z0-9./]`. This list was generated using the Rockyou[1] database, a well-known collection of compromised passwords containing over 32 million passwords leaked during a 2009 hacking attack. Only passwords of the desired length and character set were filtered for use. The resulting file contains a total of 2,829,164 passwords.

The passwords attacked during the benchmarks are determined by the number of executions. Defining N as the number of executions and $\delta = \lfloor \frac{2829164}{N} \rfloor$, the attacked password positions are as follows:

$$P = \{i \cdot \delta \mid i \in \{0, 1, \ldots, N-2\}\} \cup \{N-1\}$$

The code used to execute the benchmarks is located in `decryption-benchmarks.cc` within the `benchmarks` folder and can be run using:

```
./passwordcracker run decryption-benchmark
--numExecutions=<number of executions>
```

The parallel implementations were tested with thread counts of 2, 4, 6, 8, 12, 16, 32, 64, resulting in the following decryptors being tested:

- Sequential decryptor
- OpenMP-parallelized decryptor with 2 threads

- ...
- OpenMP-parallelized decryptor with 64 threads
- PThread-parallelized decryptor with 2 threads
- ...
- PThread-parallelized decryptor with 64 threads

For each version, the benchmarks involved attacking 10^3 passwords at various positions, as specified above, and recording:

- Minimum execution time
- Maximum execution time
- Average execution time

Execution times were measured as wall-clock time using the `omp_get_wtime` function. Specifically, the benchmarks were conducted on two machines:

1. First Machine:
   - CPU: Intel(R) Core(TM) i5-8257U
   - Total Number of Physical Cores: 4
   - Total Number of Logical Cores: 8
   - RAM: 8 GB of 2133 MHz LPDDR3 RAM
   - Storage: SSD
   - OS: macOS Sequoia 15.0.1
2. Second Machine:
   - CPU: Intel(R) Core(TM) i5-12400F
   - Total Number of Physical Cores: 6
   - Total Number of Logical Cores: 12
   - RAM: 16 GB DIMM DDR4 3600 MHz CL17
   - Storage: PCIe 4.0 NVMe SSD
   - OS: Ubuntu 24.04.1 LTS

The following subsections present the results obtained from benchmarks on these two machines.

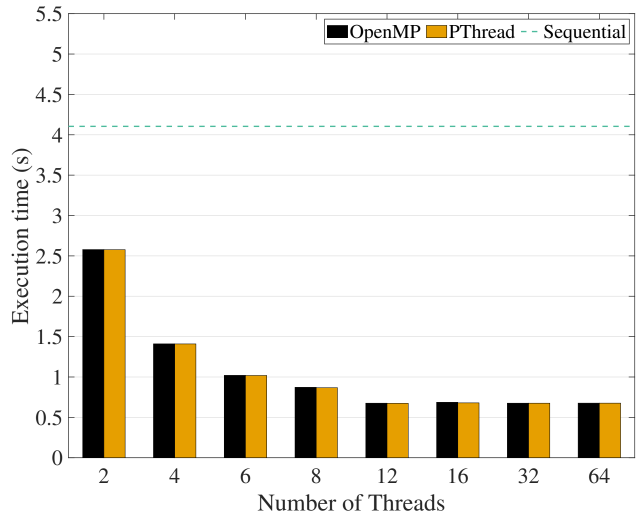

Figure 3. Execution times of parallel implementations using OpenMP and PThread with increasing thread counts on Ubuntu.

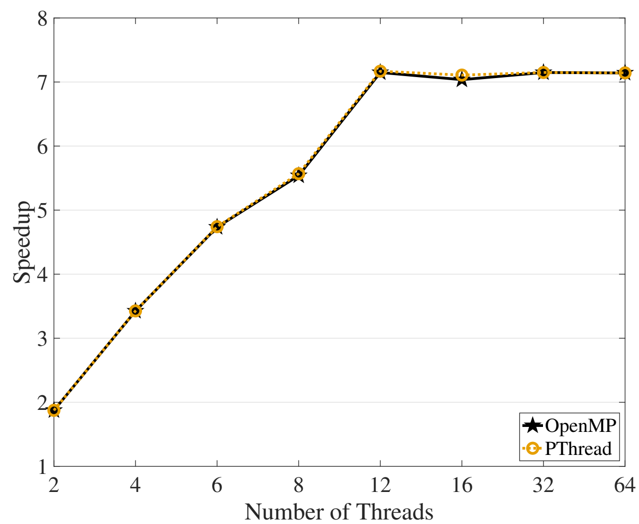

Figure 4. Speedup of parallel implementations using OpenMP and PThread with increasing thread counts on Ubuntu.

3.1. Ubuntu

The execution times on Ubuntu are shown in Figure 3 and in Table 1. The sequential implementation had an average execution time of $4.83118\,\mathrm{s}$, with minimum and maximum times of $0.15781\,\mathrm{s}$ and $9.51283\,\mathrm{s}$, respectively. The discrepancy between minimum and maximum times is attributed to the varying positions of the attacked passwords in the list.

With OpenMP, the speedup consistently improved as thread counts increased, peaking at approximately 7.14833 when utilizing 12 threads.

Num Threads	OpenMP*				PThread*			
	Min Time	*Max Time*	*Avg Time*	*Speedup*	*Min Time*	*Max Time*	*Avg Time*	*Speedup*
2	0.158805	5.06169	2.57853	1.87294	0.158675	5.06704	2.57715	1.87394
4	0.158824	2.73756	1.41039	3.42418	0.158461	2.72463	1.40954	3.42624
6	0.159456	1.88232	1.02011	4.73422	0.160320	1.87759	1.01862	4.74114
8	0.158664	1.66464	0.871962	5.53858	0.158837	1.67212	0.86732	5.56822
12	0.163764	1.19418	0.675602	**7.14833**	0.163721	1.19597	0.673749	**7.16799**
16	0.166303	1.31758	0.686362	7.03626	0.165651	1.28010	0.679468	7.10765
32	0.162851	1.20929	0.675626	7.14808	0.164484	1.18753	0.675585	7.14851
64	0.162062	1.19111	0.676298	7.14097	0.160376	1.19030	0.676185	7.14216

*All times are expressed in seconds.

Table 1. Execution times of parallel implementations using OpenMP and PThread with increasing thread counts on Ubuntu.

Beyond this, performance plateaued, reflecting full CPU utilization. The system's architecture includes 6 physical cores and 12 logical cores enabled by Hyper-Threading.

The PThread implementation mirrored this trend, reaching a similar maximum speedup of 7.16799 with 12 threads. This indicates that both libraries successfully parallelized the workload, achieving comparable efficiency despite operational differences.

Figure 4 graphically illustrates the speedup obtained with OpenMP and PThread as the number of threads increases.

3.2. macOS

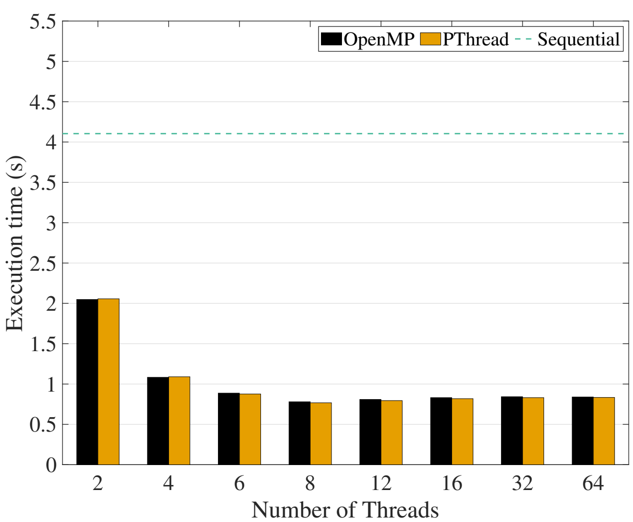

Figure 5. Execution times of parallel implementations using OpenMP and PThread with increasing thread counts on macOS.

Figure 5 and Table 2 presents the execution times measured on macOS. The sequential implementation had an average execution time of $4.10422\,\mathrm{s}$, with minimum and maximum times of $0.091218\,\mathrm{s}$ and $7.99175\,\mathrm{s}$, respectively.

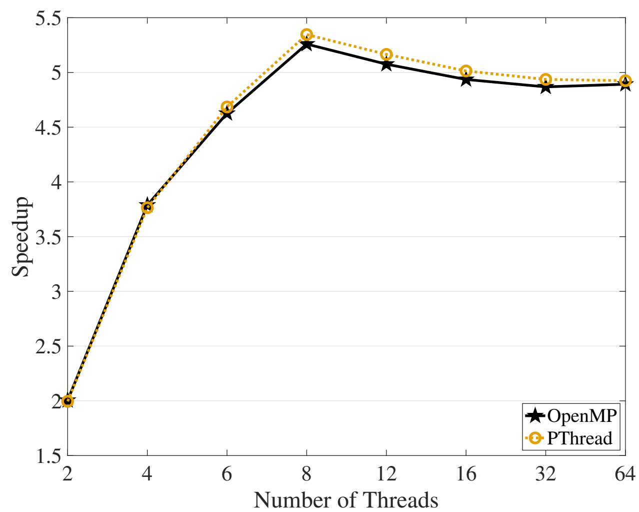

Figure 6. Speedup of parallel implementations using OpenMP and PThread with increasing thread counts on macOS.

The OpenMP implementation exhibited speedup as thread counts increased, peaking at 5.25914 with 8 threads. Beyond this point, speedup diminished slightly, indicating full utilization of the CPU's 4 physical cores and 8 logical cores with Hyper-Threading.

Similarly, the PThread implementation achieved a maximum speedup of 5.34608 with 8 threads, yielding results consistent with those of OpenMP.

Figure 6 depicts the speedup achieved by OpenMP and PThread on macOS.

Both libraries demonstrated similar scaling behavior until CPU saturation. The benefits of par-

Num Threads	OpenMP*				PThread*			
	Min Time	*Max Time*	*Avg Time*	*Speedup*	*Min Time*	*Max Time*	*Avg Time*	*Speedup*
2	0.0703931	4.01632	2.04778	2.00422	0.0680029	4.01877	2.05671	1.99552
4	0.0685871	2.42751	1.08362	3.7875	0.0672159	2.28545	1.09029	3.76433
6	0.0704	1.9228	0.88715	4.6263	0.0698168	1.8186	0.876023	4.68506
8	0.074645	1.94621	0.780398	**5.25914**	0.0735109	1.93615	0.767707	**5.34608**
12	0.0761511	1.9339	0.808661	5.07533	0.0762951	1.7504	0.794588	5.16522
16	0.079824	1.86555	0.831528	4.93576	0.0807939	1.87745	0.818606	5.01367
32	0.0807002	1.91449	0.843093	4.86805	0.080672	1.60726	0.831421	4.93639
64	0.0822098	1.7981	0.838919	4.89227	0.0815868	1.69562	0.833439	4.92444

*All times are expressed in seconds.

Table 2. Execution times of parallel implementations using OpenMP and PThread with increasing thread counts on macOS.

allelism decreased when the number of threads exceeded the available logical cores.

4. Conclusions

This project demonstrated how parallel computing techniques can significantly accelerate brute-force attacks on DES-encrypted passwords. Both parallel implementations, OpenMP and PThread, led to substantial speedups, with the speedup value approaching the number of logical cores available on the test machines.

In terms of comparison, although PThread showed slightly lower overhead, OpenMP proved to be more user-friendly, offering a simpler interface while maintaining almost identical performance. Another important observation was that the success of parallelization was strongly dependent on the hardware used, particularly the number of cores and threads. This dependency highlighted how much the effectiveness of parallel computing relies on the underlying hardware capabilities.

References

[1] Daniel Miessler. SecLists: rockyou.txt.tar.gz - Leaked Database Password List. `https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou.txt.tar.gz`.

[2] Enzo Mumolo. Introduzione ai POSIX Threads (libreria Pthread). `https://moodle2.units.it/pluginfile.php/119521/mod_resource/content/1/Pthread1617.pdf`.

[3] OpenMP. OpenMP Application Programming Interface. `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-6-0.pdf`.

[4] Wikipedia. Data Encryption Standard. `https://en.wikipedia.org/wiki/Data_Encryption_Standard`.