# DES decryption:
# A parallel approach using OPENMP

Niccolò Bellaccini
niccolo.bellaccini@stud.unifi.it

Tommaso Aldinucci
tommaso.aldinucci@stud.unifi.it

## Abstract

*In this paper we present a simple approach to Data Encryption Standard (DES) decryption using OPENMP framework. In order to allow a parallel idiom, we re-implemented a version of DES algorithm. Then we studied the behaviour of decryption process in two different strategies: a "brute force" version and a dictionary based version.*
*This OPENMP implementation has shown good results in speedup terms on 8-characters passwords using an ad-hoc dictionary.*



## 1. Introduction

The **Data Encryption Standard (DES)** is a simmetric-key algorithm for encryption of electronic data developed in the early 70s at IBM [6]. In 1977, after the acceptance of National Security Agency (NSA), it was published as an official Federal Information Processing Standard (FIPS) for the United States.

The algorithm is designed to encipher and decipher blocks of data consisting of 64 bits under control of a 64-bit key. A block, to be enciphered, is subjected to an initial permutation, then to a complex key-dependent computation and finally to a permutation which is inverse of the initial one. The key-dependent computation can be simply defined in terms of a function $f$ called cipher function.

More security has been added over the years using some salting mechanisms. In cryptography, a salt is a random additional data which can be used to perturb a user password increasing its security. It is widely used to store user's passwords in database, helping to hash same passwords to different encryptions.

### 1.1. Method overview

The classic algorithm aims to crypt a 64 bits data with a 64 bits key of which only 56 bits are used (the other 8 bits may be used for error detection). Generally [a-zA-Z0-9./] is supposed as the key characters set.

Whenever you want to protect a file with a password, you can split it into 64 bits blocks then use DES algorithm to encrypt each of them but, in this case, what we really want is to encrypt the key without a "real" 64 bits data. To allow it, usually a null 8-bytes (64-bits) string is used as data input.

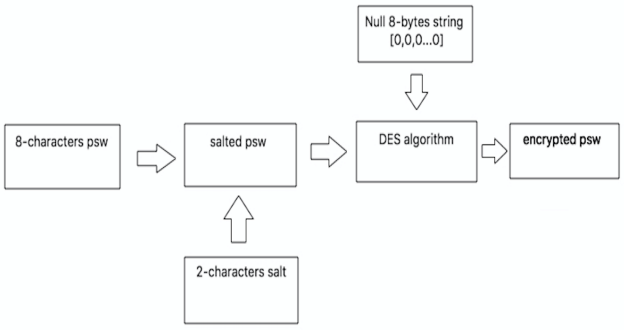


Figure 1. Crypting mechanism with salt

In this work we chose to use at most 8-characters passwords and 2-digits numeric salts. Under these assumptions we have a key space of $2^{72}$ possible values (56 bits key plus 16 bits salt).

We developed the password decryption process in two different ways: "brute force" and "dictionary". The brute force attack was implemented using a method able to generate all the possible combinations in the key space. More details will be provided in the next section.

## 2. The brute force approach

In this section we talk about brute force strategy implementation details. Since the key space has so many different values, it is computationally impraticable testing all keys in a reasonable time. In this context we want to examine only the behaviour of our solution in order to study the speedup, so we assumed [0-9] as possible characters set and 6-characters passwords.

Once we obtained the plain password and encrypted it, we start iterating on the key space testing all the possible combinations to find a match with the user encrypted password.

### 2.1. Mathematics

Iterating all over the key space, we aim to discover a function to map an index of the iteration to an unique 8-characters key (6-characters password and 2-characters salt). Using a mathematic formalism we introduce our method.

$Let$ :

- $k$ the length of password plus salt;
- $S$ the set of possible symbols with $n = |S|$;
- $I = \{0...n^k\}$ the index space;
- $K$ the space of all possible key combinations (salt and password).

so we want to find:

$$f : I \to K$$
$$i \in I \mapsto c \in K$$

such that $f$ is bijective.

The next figure shows a possible solution to determinate $f$ based on an algorithmic approach.

**Algorithm 1:** Index-to-key mapping

**Input:** $i \in I, n = |S|, k$
**Output:** $f(i) = v \in K$
**for** $j \in \{0...(k-2)\}$ **do**
  $v_j = \lfloor i/n^{k-1-j} \rfloor$ ;
  $i = i \; mod \; n^{k-1-j}$ ;
**end**
$v_{k-1} = i \; mod \; n$ ;

### 2.2. Implementation specifics

The GNU C Library provides a DES encrypt function [2] which takes a password (key) as a string and a salt char array, and returns a printable ASCII string. The function has the following signature:

**char** $*$ crypt (**const char** $*$key, **const char** $*$ salt )

Unfortunately the GNU provided crypt function is not able to handle multiple calls due to the use of pointers to static data; because of this, subsequent calls to crypt() will modify the same object.

In order to solve that, we implemented a crypt function that allowed us to use it in a multithread context.

The parallelism was achieved with OPENMP framework; more specifically we used the OPENMP "parallel for" directive to split the key-searching process. This directive made possible to automatically distribute all the workload to a fixed number of threads. There are different loop scheduling types provided by OPENMP [5], the most relevant ones are reported below:

- **static**: Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size;
- **dynamic**: Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue;
- **guided**: Similar to dynamic scheduling, but the chunk size starts off large and decreases

to better handle load imbalance between iterations.

Since we reasonably consider the workload for each iteration of the loop statement index-independent, we assumed that time-per-iteration is approximately constant. This led to choose **static** as scheduling strategy, without synchronizing any part of the code.

The parallel loop is summarised below in a general form.

**Algorithm 2:** Core of the searching function

```
Input: encrypted user-key x, Index space I
Parallel for i ∈ I do
    c = f(i) //f is the function defined in 2.1
    if x = crypt(c) then
        found = true
    end
end
```

In the next section we introduce an addictional key-search approach.

## 3. The dictionary approach

For practical purposes, if you wanted to crack a 8-characters password, the brute force would not be the best choice due to the computation time. Furthermore, a 8-characters password might suggests some semantic information *e.g.* dates. So, if you knew the victim's profile, you could have more possibilities to guess the right combination building an ad-hoc dictionary for that person.

There are many open source dictionaries which can be used for this kind of purpose. We used the "Cain & Abel" dictionary [1], adding all the date combination (from 1930) in ISO 8601 different formats, and deleting all the words with a number of characters other than 8 [1]. The final dictionary contains about 130000 different words.

### 3.1. Implementation specifics

The key-search process is approximately the same of the previous approach. Compared to the brute force method, in this case we have a significant reduction of the key space dimension. Moreover, this strategy is more usable as it tests only meaningful strings belonging to a vocabulary and it checks all of them in a reasonable computational time.

We initially load all the dictionary in a **std::vector** $\langle string \rangle$, then we effect a random permutation of the whole structure to make the search result independent by the word position.

Concerning what we said, down below we report the general form of the key-search procedure:

**Algorithm 3:** Core of the searching function

```
Input: encrypted user-key x, Dictionary space V
Parallel for s ∈ V do
    if x = crypt(s) then
        found = true
    end
end
```

This solution reported excellent results which will be shown in the next section.

## 4. Results

At this stage particular emphasis was given to the speedup concept to figure out if the taken multithreading approach brought benefits.

The speedup $S_p$ is defined as $S_p = t_s/t_p$ where, $P$ is the number of processors, $t_s$ is the completion time of the sequential algorithm and $t_p$ is the completion time of the parallel algorithm [4].

The different tests about our program timings are shown in the following tables. Each reported result was achieved on a machine with a quad-core Intel Core i7 processor @2.5 GHz, 16 GB DDR3 RAM @1600 MHz and MacOS Sierra 10.12 operating system.

We recall that, for computational time reasons, results in table 1 (bruteforce approach) were obtained using 6-characters password and 2-characters salt.

To decrease the full execution time and for portability reason we made an optimized dynamic library that provides our crypt function, compiled with Clang and "-0fast" optimization flag.

[1] We wrote a simple python-script to do it

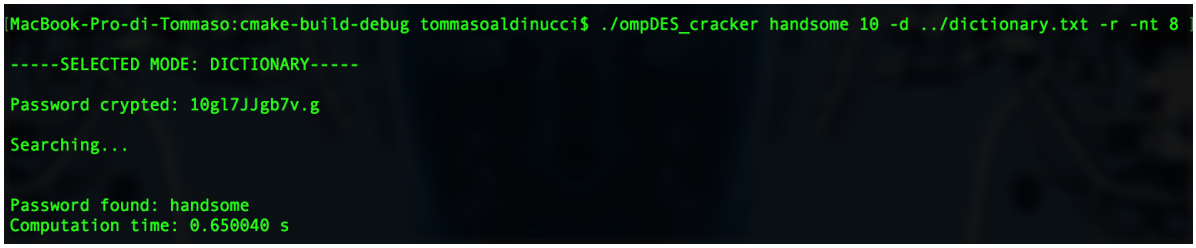


Figure 2. Example of usage: dictionary approach with 8 threads, password "handsome", salt "10" and random shuffled dictionary

| #Threads | Time ($s$) | Speedup |
|---|---|---|
| 1 | 1863 | 1.00 |
| 4 | 497 | 3.75 |
| 6 | 442 | 4.21 |
| 8 | 376 | 4.95 |
| 12 | 475 | 3.92 |

Table 1. Password matching time in bruteforce approach

| #Threads | Time ($s$) | Speedup |
|---|---|---|
| 1 | 1.38 | 1.00 |
| 4 | 0.40 | 3.45 |
| 6 | 0.36 | 3.83 |
| 8 | 0.34 | 4.06 |
| 12 | 0.38 | 3.63 |

Table 2. Password matching time in dictionary approach

All the results were obtained with the library previously mentioned, randomizing inputs and averaging times at the end of the execution.

The tables 1 and 2 show that an optimal speedup is reached with a number of threads equal to 8. It is reasonable due to the Hyper-Threading technology that makes possible to host resources of two threads in a single core. Indeed, in terms of the operating system, a single core i7 with HT seems like a bi-processor where CPUs share cache and main memory [3]. Some parallel programs can benefit from it using a number of threads equivalent to the number of logical processors.

## 5. Conclusion

We described a OPENMP based method to exploit parallelism in the search process of a DES encrypted password. Clearly, the dictionary approach turned out more effective results for practical purposes even if completeness is not guaranteed.

The obtained speedups are consistent with what was expected, being aware of the limits of our machine. Since our procedure operates on indipendent chunks of the key-space, we didn't need to syncronize any part of the code. So, we can reasonably assume a sub-linear speedup trend increasing the number of processors.

## References

[1] Cain and abel dictionary. `https://wiki.skullsecurity.org/Passwords`.

[2] Encrypting password. `https://www.gnu.org/software/libc/manual/html_node/crypt.html`.

[3] T. A. Andrew S. Tanenbaum. *Structured Computer Organization*. Pearson.

[4] L. S. Calvin Lin. *Principles of parallel programming*. Pearson.

[5] Intel. Openmp loop scheduling. https://software.intel.com/en-us/articles/openmp-loop-scheduling.

[6] W. Tuchman. Internet besieged. chapter A Brief History of the Data Encryption Standard, pages 275–280. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.