DES Password Decrypter

Adriano Di Dio
E-mail address
adriano.didio@stud.unifi.it

Abstract

Purpose of this paper is to describe how to decrypt a DES hashed password using brute-force.
In particular, it will be shown how to implement the main algorithm in C and how it can be sped up using multiple threads through the pthreads library.

1. Introduction

Data Encryption Standard (DES) is a symmetric-key algorithm used for the encryption of data.
DES passwords can be generated using a library available for the C programming language called 'libcrypt' that exposes two versions of the same function named: crypt and crypt_r.
The first one is the standard and will be used in the sequential version, while the latter is the reentrant version of the same function which is required when dealing with multiple threads due to the possibility of the scheduler to interrupt any thread at any time, causing the library to not complete the encryption operation and returning invalid data.
Differences about the two function will be given in the chapters related to the sequential and parallel version.
In particular in chapter 2 a brief explanation of the algorithm is given,then in chapter 3 and 4 the sequential and parallel version is described, and finally, in the last chapter, the performance will be evaluated by showing the differences between the two approaches.

2. Algorithm

Every password created using the crypt function has the following format:

```
salt+hash
```

the salt is used as a countermeasure to specific attacks such as the Rainbow Table method where an attacker could generate a list of hashed password that can be combined to find the correspondent plain-text one, in fact, by adding a random salt,that should be removed from the final password, these kind of attack could be mitigated due to the necessity of knowing which salt has been applied.
In our case, the salt is always known since it is given by the user itself when running the decrypt program.
The brute-force attack, is a process where the attacker tries all the possible available combination of a specific charset.
As an example, this is the default charset that will be used in the implementation:

abcdefghilmnopqrstuvzABCDEFGHILMNOPQRSTUVZ0123456789./

Figure 1. Default Charset

the algorithm is quite simple and the main objective is to find all the possible combination of the letters available in the charset and run, for each combination, the crypt function to test if the hashed password matches the one that was given from the user.
Below, a pseudo-code implementation, to describe the algorithm:

Algorithm 1 Password Decrypter
Takes 4 parameters:Password,Salt,Charset,Password Length

```
1:  procedure GUESSPASSWORD(Password,Salt,Charset,Length)
2:      Let CurrentCombination a new list of length ← Length
3:      Let Position a new list of length ← Length
4:      for i ← 0 to Length do
5:          Combination[i] ← Charset[0]
6:          Position[i] ← Charset[0]
7:      while True do
            CRYPT(Combination,Salt) ← Hash
8:          if Hash == Password then
9:              return Combination
10:         Place ← Length − 1
11:         while Place >= 0 do
12:             Position[Place] ← Position[Place] + 1
13:             Len ← len(Charset)
14:             if Position[Place] == Len then
15:                 Position[Place] ← 0
16:                 Combination[Place] ← Charset[0]
17:                 Place ← Place − 1
18:             else
19:                 Letter ← Charset[Place]
20:                 Combination[Place] ← Letter
21:                 break
22:         if Place < 0 then
23:             break
24:     return ""
25:
26: procedure DECRYPT(MaxLength)
27:     for i ← 1 to MaxLength do
28:         GUESSPASSWORD(Password, MaxLength) ← Password
29:         if Password ≠ "" then
30:             return Password
31:     return ""
```

As we can see from the pseudo-code the algorithm is quite simple, and it works like a counter that produces all the available combination from the given charset.
In particular the entry function is called decrypt, which calls the GuessByLength function in a loop that goes from 1 to the maximum allowed password length,which calculates all the possible combinations.
Inside the GuessByLength function we have the main algorithm that generates the possible combinations.
In the first part, we initialize two arrays, Position and Combination, that are used respectively as the Counter and the corresponding combination based on the counter values.
Then the loop is started and the first combination is checked by calling the crypt function that produces the hash for the current combination and salt, and returns the hash value as a string that can be compared to the encrypted password, if it matches the function returns the plain-text password otherwise it creates a new combination by Starting from the maximum length of the password and adding to the counter the current position relative to the charset.
When the counter has reached the last letter of the charset, it wraps around, and start a new count on the other element of the array.
If all the values have been tried then the main loop is stopped and the function returns an empty string otherwise it will keep iterating until one of the previous condition is met.
When the function 'GuessByLength' returns, the decrypt function that called it, checks if the returned password is not empty and it will stop the iteration and returns the result to the user,otherwise it will keep iterating until the max length is reached.

3. Implementation

3.1. Tools

A tool has been written in C to generate a DES password, using the crypt function, and requires only two parameters:the plain-text password and the salt.
It returns the encrypted password that can be later used in the decrypt application to test it out.
E.G

```
$ ./Crypt foo aa
```

Sample Output:

```
$ Crypted Password for foo is ...
```

3.2. Main Program

The algorithm, seen in the pseudo-code,is implemented in both the sequential version and in the parallel one in C but the implementation is slightly different in the parallel one to optimize thread usages.

3.2.1 Sequential Version

The sequential version is made using two functions as seen in the pseudocode, one for iterating over the maximum allowed length while the other to create the possible combinations.
The following structure have been declared to initialize a decrypt session:

```
typedef struct DecypherSettings_s {
    int MaxLength;
    char Salt[3];
    char *EncryptedPassword;
    char *Charset;
    char *DecryptedPassword;
    int CharsetSize;
} DecypherSettings_t;
```

Listing 1. Data Structure Definition

this structure is initialized in the main function as it can be seen below:

```
DecypherSettings_t* DecypherSettingsInit(char *
    ↪ Key,int MaxLength,char *Charset)
{
    DecypherSettings_t *Out;
    if( strlen(Key) <= 2 ) {
        printf("DecypherSettingsInit:Invalid Key
            ↪ .\n");
        return NULL;
    }
    Out = malloc(sizeof(DecypherSettings_t));
    Out->MaxLength = MaxLength;
    Out->Salt[0] = Key[0];
    Out->Salt[1] = Key[1];
    Out->Salt[2] = '\0';
    Out->EncryptedPassword = StringCopy(Key);
    if( Charset != NULL ) {
        Out->Charset = StringCopy(Charset);
    } else {
        Out->Charset = StringCopy(DefaultCharset)
            ↪ ;
    }
    Out->CharsetSize = strlen(Out->Charset);
    return Out;
}
```

Listing 2. DecypherSettingsInit Function

and contains all the data needed to implement the algorithm, the Decrypted Password field will be populated once the algorithm has finished and a password was found,the charset can be set from the command line or it can use the default one which is:

```
char DefaultCharset[] = "
    ↪ abcdefghilmnopqrstuvzABCDEFGHILMNOPQRSTUVZ0123456789
    ↪ ./";
```

Listing 3. Default Charset

After initialization the decrypt function is called, passing the previous initialized data structure (DecypherSettings), and the algorithm starts:

```
void Decrypt(DecypherSettings_t *DecypherSettings
    ↪ )
{
    int Start;
    int End;
    int Found;

    Start = Sys_Milliseconds();
    Found = 0;
    for( int i = 1; i <= DecypherSettings->
        ↪ MaxLength; i++ ) {
        if( GuessPasswordByLength(
            ↪ DecypherSettings,i) != -1 ) {
            Found = 1;
            break;
        }
    }
    End = Sys_Milliseconds();
    if( Found ) {
        printf("Password %s took %i msec to be
            ↪ cracked\n",DecypherSettings->
            ↪ EncryptedPassword,End-Start);
    } else {
        printf("Password not found...try
            ↪ increasing the max length.\n");
    }
}
```

Listing 4. Decrypt Function

This function, as seen in the pseudo-code, calls 'GuessPasswordByLength' with an increasing MaxLength until either the password is found or the length has reached the maximum value.
The function can be seen in below snippet:

```
int GuessPasswordByLength(DecypherSettings_t *
    ↪ DecypherSettings,int PasswordLength)
{
    int *PositionIndex;
    char *CurrentCombination;
    int Place;
    int Found;
    int i;

    PositionIndex = (int*) malloc(PasswordLength
        ↪ * sizeof(int));
    CurrentCombination = (char*) malloc(
        ↪ PasswordLength * sizeof(char) + 1);

    for( i = 0; i < PasswordLength; i++ ) {
```

```
        CurrentCombination[i] = DecypherSettings
            ↪ ->Charset[0];
        PositionIndex[i] = 0;
    }
    CurrentCombination[i] = '\0';
    Found = -1;
    while(1) {
        if( ComparePassword(DecypherSettings,
            ↪ CurrentCombination) ) {
            DecypherSettings->DecryptedPassword =
                ↪  StringCopy(CurrentCombination
                ↪ );
            Found = 1;
            break;
        }
        Place = PasswordLength - 1;
        while( Place >= 0 ) {
            PositionIndex[Place]++;
            if( PositionIndex[Place] ==
                ↪ DecypherSettings->CharsetSize
                ↪ ) {
                PositionIndex[Place] = 0;
                CurrentCombination[Place] =
                    ↪ DecypherSettings->Charset
                    ↪ [0];
                Place--;
            } else {
                CurrentCombination[Place] =
                    ↪ DecypherSettings->Charset[
                    ↪ PositionIndex[Place]];
                break;
            }
        }
        if( Place < 0 ) {
            break;
        }
    }
    free(CurrentCombination);
    free(PositionIndex);
    return Found;
}
```

Listing 5. GuessPasswordByLength Function

where the function is implemented, in the same way as seen in the pseudo-code, it will generate all possible combination given the maximum length and for each combination it will call the 'ComparePassword function':

```
int ComparePassword(DecypherSettings_t *
    ↪ DecypherSettings,char *PasswordAttempt)
{
    char *HashedPasswordAttempt;
    HashedPasswordAttempt = crypt(PasswordAttempt
        ↪ ,DecypherSettings->Salt);
    return strcmp(HashedPasswordAttempt,
        ↪ DecypherSettings->EncryptedPassword)
        ↪ == 0;
}
```

Listing 6. ComparePassword Function

that, as it can be seen by the snippet above, calculates an hash with the same salt as the one the user passed to the program and if the two hash matches then the plain-text password is returned based on the current combination used to generate it.
When this function returns 1 then the password has been found and the loop in the Decrypt function is interrupted in order to print out the plain-text password along with the execution time in ms.
If the password is not found and the loop has reached his maximum value then a message will inform the user that the password could not be decrypted.

3.3. Usage

To use the application, user needs to specify only the encrypted password and the maximum length:

```
$ ./PasswordDecrypterSP asgCdZs 8
```

If the user want to change the charset, they can be modified by using the following command-line option:

```
$ /PasswordDecrypterMP --Charset abcd
```

By default the charset is the one seen in figure 2

3.4. Parallel Version

As we can see from the pseudo-code, the generation of all the possible combination can be done in parallel by distributing the available combination to multiple threads that can check if the password match in parallel without having to wait.
The following data structures have been declared to hold the status of the decrypter between multiple threads:

```
typedef enum {
    DECRYPTER_JOB_STATUS_NOT_COMPLETED = 0,
    DECRYPTER_JOB_STATUS_FOUND = 1,
    DECRYPTER_JOB_STATUS_REACHED_MAX_COMBINATION
        ↪ = 2
} DecrypterJobStatus;

```

```
typedef struct DecipherSettings_s {
    int MaxLength;
    char Salt[3];
    char *EncryptedPassword;
    char *Charset;
    char *DecryptedPassword;
    int CharsetSize;
    int CharsetIncrement;
} DecipherSettings_t;

typedef struct PoolWork_s {
    int CurrentLength;
    int CurrentPositionValue;
    int TargetPositionValue;
    int CharsetIterator;
} PoolWork_t;

typedef struct PoolJob_s {
    PoolWork_t GlobalWorkStatus;
    pthread_t *ThreadPool;
    DecipherSettings_t Settings;
    pthread_mutex_t JobStatusMutex;
    pthread_cond_t JobStatusCondition;
    int JobStatus;
    int ThreadPoolSize;
    char *DecryptedPassword;
} PoolJob_t;
```

Listing 7. Parallel Data Structure

in the DecypherSettings structure we have the same data as the one seen in the sequential version that holds the configuration for the decryption session that we need to run, and it is initialized in the same way:

```

    StaticDecipherSettings.EncryptedPassword =
        ↪ StringCopy(argv[1]);
    StaticDecipherSettings.Salt[0] = argv[1][0];
    StaticDecipherSettings.Salt[1] = argv[1][1];
    StaticDecipherSettings.Salt[2] = '\0';
    if( argv[3] != NULL ) {
        StaticDecipherSettings.Charset =
            ↪ StringCopy(argv[3]);
    } else {
        StaticDecipherSettings.Charset =
            ↪ StringCopy("
            ↪ abcdefghilmnopqrstuvzABCDEFGHILMNOP
            ↪  \
            QRSTUVZ0123456789./");
    }
    StaticDecipherSettings.CharsetSize = strlen(
        ↪ StaticDecipherSettings.Charset);
```

Listing 8. Decrypter Session Data Structure Initialization

then, the thread structure is initialized by defining the starting conditions:

```
    StaticDecipherSettings.CharsetIncrement = 2;

    PoolJob.Settings = StaticDecipherSettings;
    StaticGlobalWorkStatus.CurrentLength = 1;
    StaticGlobalWorkStatus.CurrentPositionValue =
        ↪  0;
    StaticGlobalWorkStatus.TargetPositionValue =
        ↪ 0;
    StaticGlobalWorkStatus.CharsetIterator = 0;
    PoolJob.GlobalWorkStatus =
        ↪ StaticGlobalWorkStatus;
    pthread_mutex_init(&PoolJob.JobStatusMutex,
        ↪ NULL);
    pthread_cond_init(&PoolJob.JobStatusCondition
        ↪ ,NULL);
    PoolJob.JobStatus =
        ↪ DECRYPTER_JOB_STATUS_NOT_COMPLETED;
    PoolJob.DecryptedPassword = NULL;
```

Listing 9. Parallel Data Structure Initialization

As we can see from the above snippet the program uses one mutex and one condition to handle thread synchronization, and a pool of four workers threads to generate the available combinations.
Note that the number of workers can be set from command-line.
The actual thread are then started:

```
    PoolJob.ThreadPool = (pthread_t *) malloc(
        ↪ PoolJob.ThreadPoolSize * sizeof(
        ↪ pthread_t));

    pthread_create(&Master,NULL,MasterWork,(void
        ↪ *) &PoolJob);
    for(i = 0; i < PoolJob.ThreadPoolSize; i++ )
        ↪ {
```

Listing 10. Thread Launch

there are two types of thread, the first one is the Master which handles thread synchronization when an exit event occur while the others are the workers thread which do the actual required job.
The master thread is only one and uses the following function:

```
}
void *MasterWork(void *Arg)
{
    PoolJob_t  *Pool;
    int i;
    int JobStatus;

    Pool = (PoolJob_t *) Arg;

    pthread_mutex_lock(&Pool->JobStatusMutex);

    while (Pool->JobStatus ==
        ↪ DECRYPTER_JOB_STATUS_NOT_COMPLETED )
        pthread_cond_wait(&Pool->
            ↪ JobStatusCondition, &Pool->
            ↪ JobStatusMutex);

    JobStatus = Pool->JobStatus;
    pthread_mutex_unlock(&Pool->JobStatusMutex);
```

```
17
18      if( JobStatus == DECRYPTER_JOB_STATUS_FOUND)
            ↪ {
19          for (i = 0; i < Pool->ThreadPoolSize ; i
                ↪ ++) {
20              pthread_cancel(Pool->ThreadPool[i]);
21          }
22      }
23      for(i = 0; i < Pool->ThreadPoolSize; i++ ) {
24          pthread_join(Pool->ThreadPool[i],NULL);
25      }
26      pthread_exit(NULL);
```

Listing 11. Master Thread

as we can see from the snippet, the master thread will wait until an exit condition occurs.
An exit condition happens when either one worker finds the password or when a worker has reached the maximum allowed combination given the maximum length, when one of these events occurs the worker broadcast a signal that wakes up the master thread that will look to the shared data structure to see what exit code the worker has put into the variable JobStatus.
If the JobStatus is equals to 'decrypter_job_Status_found' then the master thread will cancel all the running worker threads using the function 'pthread_cancel' and waits for their termination using the 'pthread_join' function otherwise, if the JobStatus is equals to 'decrypter_job_status_reached_max_combination' then it will wait for normal termination of all the worker threads by calling the 'pthread_join'.
When all the threads have joined, the master will exit, terminating itself.
The workers uses a different function which is called 'DoWork'.
It is important to note that in order to avoid any overhead when dealing with trivial passwords, the workers thread must be initialized as follows:

```
1       pthread_setcancelstate(PTHREAD_CANCEL_ENABLE,
            ↪  NULL);
2       pthread_setcanceltype(
            ↪ PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

Listing 12. Worker Thread Initialization

this ensures that the threads could be cancelled at any time without having to wait for a specific cancellation point.
Then every worker thread must initialize the data structure required by the reentrant crypt function as shown below:

```
1       struct crypt_data data;
2
3       data.initialized = 0;
```

Listing 13. Crypt Function Initialization

if the initialized value is not zero or is not set then the crypt function will not work properly.
After initialization the main worker thread code is executed as an infinite while loop, on every iteration the worker checks if there are any available jobs by calling the function 'PoolGetAvailableJob':

```
1           Work = (PoolWork_t *) PoolGetAvailableJob
                ↪ (Pool);
```

Listing 14. PoolGetAvailableJob Function

this function checks the current status of the decryption session and advances the counter till the maximum allowed length is reached.
Since the data structure is shared a mutex is used to avoid any data race as seen below:

```
1   void *PoolGetAvailableJob(PoolJob_t *Pool)
2   {
3       PoolWork_t *Work;
4       int RealCharsetSize;
5
6       pthread_mutex_lock(&Pool->JobStatusMutex);
7       RealCharsetSize = Pool->Settings.CharsetSize
            ↪ - 1;
8       if( Pool->GlobalWorkStatus.
            ↪ TargetPositionValue >= RealCharsetSize
            ↪  ) {
9           Pool->GlobalWorkStatus.CharsetIterator =
                ↪ 0;
10          Pool->GlobalWorkStatus.CurrentLength++;
11      }
12      if( Pool->GlobalWorkStatus.CurrentLength >
            ↪ Pool->Settings.MaxLength ) {
13          pthread_mutex_unlock(&Pool->
                ↪ JobStatusMutex);
14          return NULL;
15      }
16      Pool->GlobalWorkStatus.CurrentPositionValue =
            ↪  Pool->GlobalWorkStatus.
            ↪ CharsetIterator *
17          Pool->Settings.CharsetIncrement;
18      Pool->GlobalWorkStatus.TargetPositionValue =
            ↪ ((Pool->GlobalWorkStatus.
            ↪ CharsetIterator + 1)
19          * Pool->Settings.CharsetIncrement);
20      //Clamp it if we have gone out of bounds...
21      if( Pool->GlobalWorkStatus.
            ↪ TargetPositionValue > RealCharsetSize
            ↪ ) {
```

```
        Pool->GlobalWorkStatus.
            ↪ TargetPositionValue =
            ↪ RealCharsetSize;
    }
    Work = (PoolWork_t *) malloc(sizeof(
        ↪ PoolWork_t));
    Work->CurrentPositionValue = Pool->
        ↪ GlobalWorkStatus.CurrentPositionValue;
    Work->TargetPositionValue = Pool->
        ↪ GlobalWorkStatus.TargetPositionValue;
    Work->CurrentLength = Pool->GlobalWorkStatus.
        ↪ CurrentLength;
    Pool->GlobalWorkStatus.CharsetIterator++;
    pthread_mutex_unlock(&Pool->JobStatusMutex);
    return Work;
}
```

Listing 15. PoolGetAvailableJob Function

this function generates a new job that the worker can do by giving a range of combination that it can generates, the range size depends from the 'CharsetIncrement' variable that holds the size of each chunk.
When the maximum length is reached, it will return NULL and the worker thread will broadcast the exit condition to the master thread:

```
        if( Work == NULL ) {
            WorkerQuit(Pool,DECRYPTER_JOB_STATUS_
                ↪ REACHED_MAX_COMBINATION,NULL);
        }
```

Listing 16. Worker Exit

otherwise, the worker will generate all the combination in the given range:

```

        while( 1 ) {
            if( WorkerHasReachedMaxCombination(
                ↪ StartPositionIndex,
                ↪ TargetPositionIndex,Work->
                ↪ CurrentLength) ) {
                break;
            }
            if( ComparePassword(&Pool->Settings,
                ↪ CurrentCombination, &data) ) {
                DecryptedPassword = (char *)
                    ↪ malloc(strlen(
                    ↪ CurrentCombination) + 1);
                strcpy(DecryptedPassword,
                    ↪ CurrentCombination);
                WorkerQuit(Pool,
                    ↪ DECRYPTER_JOB_STATUS_FOUND
                    ↪ ,DecryptedPassword);
            }
            Place = Work->CurrentLength - 1;
            while( Place >= 0 ) {
                StartPositionIndex[Place]++;
                if( StartPositionIndex[Place] ==
                    ↪ Pool->Settings.CharsetSize
                    ↪  ) {
                    StartPositionIndex[Place] =
                        ↪ 0;
                    CurrentCombination[Place] =
                        ↪ Pool->Settings.Charset
                        ↪ [0];
                    Place--;
                } else {
                    CurrentCombination[Place] =
                        ↪ Pool->Settings.Charset
                        ↪ [StartPositionIndex[
                        ↪ Place]];
                    break;
                }
            }
            if( Place < 0 ) {
                break;
            }
```

Listing 17. Worker Combination Generator

As we can see from above there are two utilities function:'WorkerHasReachedMaxCombination' and 'ComparePassword'.
'WorkerHasReachedMaxCombination' simply checks if the worker has completed his job by comparing the current counter status with the target one:

```
int WorkerHasReachedMaxCombination(int *
    ↪ PositionIndexArray,int *
    ↪ TargetPositionIndexArray,int
    ↪ PositionIndexSize)
{
    int NumMaxCount;
    int i;

    NumMaxCount = 0;

    for( i = 0; i < PositionIndexSize; i++ ) {
        if( PositionIndexArray[i] ==
            ↪ TargetPositionIndexArray[i] ) {
            NumMaxCount++;
        }
    }
    return NumMaxCount == i;
}
```

Listing 18. WorkerHasReachedMaxCombination Function

while 'ComparePassword' calls the crypt_r function, along with his data structure, and check if the hash of the current combination matches the password:

```
int ComparePassword(DecipherSettings_t *
    ↪ DecipherSettings, char *PasswordAttempt,
    ↪ struct crypt_data *CryptReentrantData)
{
    char *HashedPasswordAttempt;
    HashedPasswordAttempt = crypt_r(
        ↪ PasswordAttempt, DecipherSettings->
        ↪ Salt, CryptReentrantData);
    return strcmp(DecipherSettings->
        ↪ EncryptedPassword,
        ↪ HashedPasswordAttempt) == 0;
```

```
}
```

Listing 19. Compare Password Function

if it does then the worker thread will save it to the shared data structure and will wake up the master in order to terminates all the running threads.
The exit function is shown below:

```
void WorkerQuit(PoolJob_t *Pool,
    ↪ DecrypterJobStatus Reason,char *
    ↪ DecryptedPassword)
{
    pthread_mutex_lock(&Pool->JobStatusMutex);
    Pool->JobStatus = Reason;
    if( DecryptedPassword != NULL ) {
        Pool->DecryptedPassword =
            ↪ DecryptedPassword;
    }
    pthread_cond_broadcast(&Pool->
        ↪ JobStatusCondition);
    pthread_mutex_unlock(&Pool->JobStatusMutex);
    pthread_exit(NULL);
}
```

Listing 20. Worker Exit Function

Finally the program will exit when the master thread exit which is detected using the 'pthread_join' statement.
The parallel implementation has been profiled using Intel's Vtune-profiler which showed no data races and full CPU cores usage,altough as we can see from the image it warned about some issues about being memory bound.

3.5. Usage

To use the application, user needs to specify only the encrypted password and the maximum length:

```
$ ./PasswordDecrypterMP asgCdZs 8
```

If the user want to increase the number of workers threads or change the charset, they can be modified by using two command-line options:

```
$ /PasswordDecrypterMP
--NumThreads 8 --Charset abcd
```

By default the number of threads is four and the charset is the one seen in figure 2

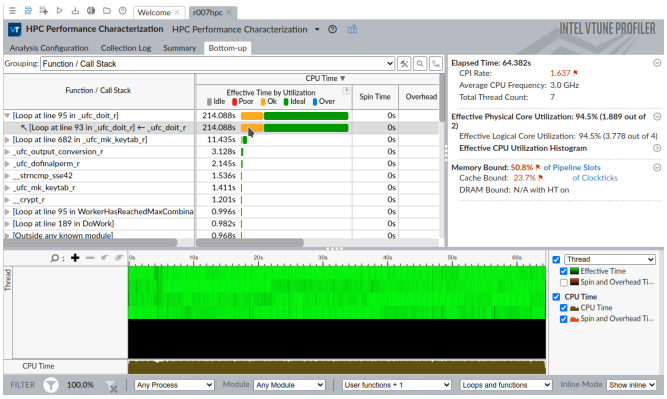

Figure 2. Intel's VTune-Profiler

These issues were actually related to the libCrypto, which the program uses to test if the password matches, and cannot be directly fixed.

4. Metrics

4.1. Run-Time Comparison

In order to see what advantages the parallel implementation has brought we need to measure the execution time of both implementations by using the same passwords.
The processor, used to collect the data, is an 'Intel(R) Core(TM) i5-4200M CPU' that has two cores and 4 hardware threads.
The execution time is measured using the same function:

```
int StartSeconds = 0;
int SysMilliseconds()
{
    struct timeval tp;
    int CTime;
    gettimeofday(&tp, NULL);
    if ( !StartSeconds ){
        StartSeconds = tp.tv_sec;
        return tp.tv_usec/1000;
    }
    CTime = (tp.tv_sec - StartSeconds)*1000 + tp.
        ↪ tv_usec / 1000;
    return CTime;
}
```

Listing 21. Time Function

that measures the elapsed time in milliseconds.
Under the hood it uses the function 'gettimeofday()' that returns the Wall Clock and the results are available in the following table:

Password	SP Time	MP Time	SpeedUp
foo	60 ms	21 ms	2.85
fooB	3005 ms	1157 ms	2.59
foobB	164411 ms	64443 ms	2.55
foobbB	171,612 m	72.84 m	2.35

Table 1. Execution Times

4.2. Parallel Run-time vs Number Of Threads

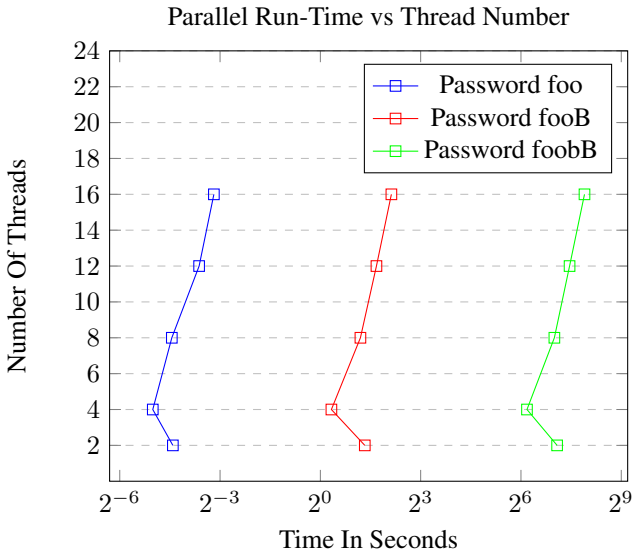

Figure 3. Run-Time vs Number Of Threads Graph

As we can see from the above graph, the performance of the parallel version depends from the number of the thread used.
In particular since the test were done using a CPU that has two cores and four hardware threads then the optimal number of required workers threads is four.
In fact, if we use two threads then the application takes a little longer to complete and it can be seen from the Vtune profiler that the Core usage has dropped:

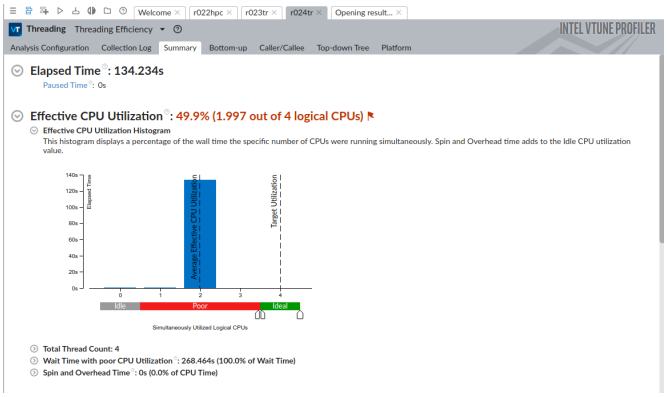

Figure 4. Intel's VTune-Profiler showing non-optimal thread usage (Under-Usage)

if we increase the number of worker threads then the logical core usage is almost the same but the performance starts to drop due to the overhead required to manage the software threads that have to be scheduled by the system:

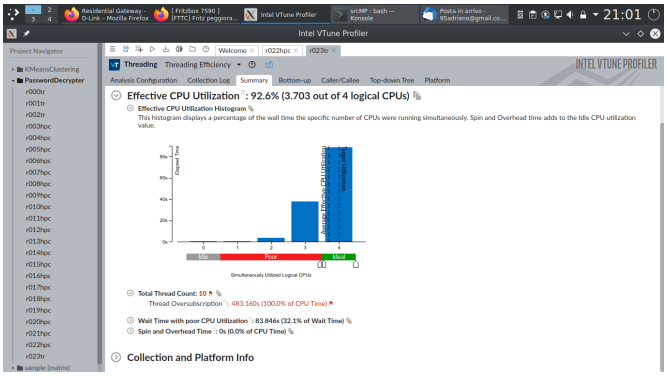

Figure 5. Intel's VTune-Profiler showing non-optimal thread usage (Over-Usage

When using a number of worker threads equal to the processor's hardware threads the usage is optimal:

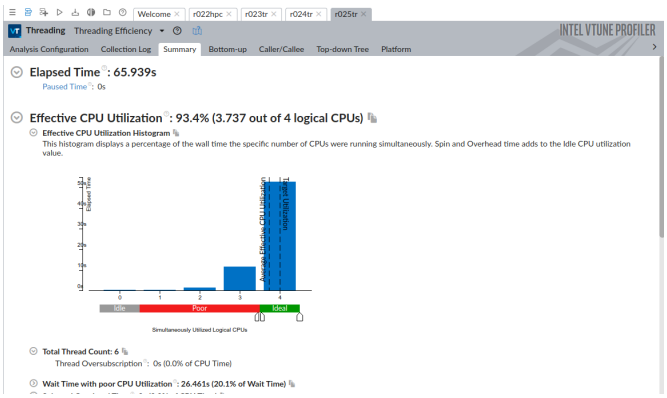

Figure 6. Intel's VTune-Profiler showing optimal thread usage (Normal)

5. Conclusions

As we have seen in the metric section the overall performance of the parallel version is two times faster than the sequential version, and even in simple cases, like the first result, the parallel version is still faster than the sequential one thanks to the usage of multiple threads.
It is important to note,as seen in section 4.2, that in case of higher core CPU, the program should be instructed to use them by specifying the right number of workers threads.