Confronto delle implementazioni sequenziale e parallela dell'attacco all'algoritmo di cifratura DES

Lorenzo Macchiarini
7045205

Andrea Leonardo
7070775

Abstract

In questo progetto abbiamo analizzato diversi algoritmi per l'attacco di un sistema di cifratura DES e come il loro tempo di esecuzione migliori con la parallelizzazione. Abbiamo svolto i test implementando una versione sequenziale in C++ parallelizzata tramite C++Threads e una versione sequenziale in Java parallelizata tramine JavaThreads.

1. Introduzione al DES

Il Data Encription Standard (DES) è un algoritmo di cifratura a chiave condivisa, ovvero prevede una chiave segreta e condivisa solo tra emittente e destinatario del messaggio. La chiave essendo unica è utilizzata sia per la cifratura sia per la decifratura del messaggio. Il DES è un algoritmo di cifratura a blocchi, il quale prende in ingesso un blocco di 64 bit che viene permutato diverse volte tramite funzioni apposite come la funzione XOR e la funzione di Feistel. Quest'ultima funzione sfrutta la chiave per generare le permutazioni più complesse, anche se è proprio la chiave stessa che è considerata uno degli anelli deboli dell'algoritmo, soprattutto per la lunghezza standard la quale consta di 56 bit. In questo progetto non analizzeremo l'algoritmo come tale, ma vedremo l'implementazione di diversi attacchi al sistema:

- *Attacco su password note*: in questo attacco verrà inteso un tentativo di ricerca di una password specifica contenuta in un insieme di password comunemente usate racchiuse nel file "rockyou.txt". L'esempio può essere identificato come una specie di tentativo di login ad un social network. La chiave di cifratura è conosciuta dal sistema, che la usa per cifrare il tentativo, ma è sconosciuta all'utente che può solo tentare password fino a che il ciphertext in suo possesso non corrisponde a quello generato dal sistema. L'utente tenta le password contenute nel file di rockyou (poiché diamo per scontato che la password sia presente tra esse).

- *Attacco su PINs generici*: questo attacco intende una ricerca su un PIN di 8 numeri, dunque è intesa una qualsiasi disposizione con ripetizione dei numeri da 0 a 9 in un gruppi di 8. Le possibilità sono dunque $10^8 = 100000000$ (100 mln) e possono rappresentare sia PINs generici sia date in qualsiasi formato. Anche in questo caso la struttura della ricerca è simile a quella del punto precedente, ovvero una ricerca ciclica su tutte le possibili stringhe di 8 numeri cifrate in una "black box". La cifratura ottenuta viene dunque paragonata con quella posseduta e l'uguaglianza ne identifica la correttezza

- *Attacco sulla chiave*: questo attacco rappresenta invece il tipico attacco forza bruta che viene effettuato sui cifrati DES. Viene tentata una chiave qualsiasi fino a che la decryption (effettuata con l'utilizzo della chiave generata) del ciphertext in possesso non somiglia abbastanza a una password realistica. Questa distanza viene stabilita da una unione tra n-grams comuni e distanza di Jaccard con le parole all'interno di "rockyou.txt". Anche se la chiave reale dei cifrari DES, composta dunque da 56 bit, è rintracciabile in tempi comprensibili noi utilizzeremo chiavi nettamente più corte per realizzare esperimenti più semplici.

2. Implementazione Java

Definiamo prima di tutto come è strutturato il programma e quali siano le funzioni utilizzate per la cifratura DES. Tutti i metodi utilizzati sono contenuti all'interno della classe pubblica *DES*. Essa presenta due cifratori (oggetti della classe Cipher) *ecipher* e *dcipher*, i quali hanno il compito di cifrare o decifrare una stringa assegnata rispettivamente, e una chiave (oggetto della classe SecretKey). I metodi utilizzati nella ricerca sono:

1. *encrypt()*: prende in ingresso una stringa rappresentante la parola da cifrare e restituisce un'altra stringa rappresentante la parola cifrata.

2. *decrypt()*: prende in ingresso una stringa rappresentante la parola cifrata e restituisce un'altra stringa rappresentante la parola decifrata.

3. *readTxt()*: prende in ingresso un array di parole e un alfabeto, e modifica il dato array eliminando tutte le parole più lunghe di 8 caratteri e contenenti caratteri non presenti nell'alfabeto. Questo metodo viene utilizzato per la modifica del file rockyou, in modo che presenti le password più realistiche.

Vediamo ora le implementazioni, sequenziale e parallela, degli attacchi nel linguaggio Java.

2.1. Implementazione sequenziale

Nell'implementazione Java sono stati elaborati tutti gli attacchi presentati precedentemente. Sono presenti nel metodo *main()* della classe DES, nel quale viene scelto l'attacco da utilizzare tramite gli argomenti passati:

- $-m$: identifica il messaggio da cifrare o decifrare
- $-k$: identifica la chiave se conosciuta
- $-d$: indica al programma di effettuare una decryption
- $-ry$: indica al programma di effettuare la decription tramite rockyou (attacco su password note)
- $-p$: indica al programma di effettuare l'algoritmo di attacco in parallelo

2.1.1 Attacco su password note

L'attacco su password note è definito dai parametri $-m$"$String$" $-$ k"$String$" $-$ $d-ry$. Se la chiave viene definita nei parametri viene utilizzata per definire i cifratori del DES. In seguito viene eseguito l'algoritmo di attacco:

```
readTxt(data, A);
long start = System.currentTimeMillis();
System.out.println("\nSearching for password
   in Rockyou...");
for (String datum : data) {
   password = new StringBuilder();
   password.append(datum);
   if (Objects.equals(encrypt(
         password.toString()), message)) {
      long finish = System.currentTimeMillis();
      System.out.println("LA PASSWORD
         CORRETTA E': " + password);
      System.out.println("time: " +
         (finish - start) / 1000f + " sec");
      break;
   }
}
```

L'algoritmo presenta un ciclo for su tutti gli elementi di $rockyou.txt$, filtrati inizialmente dall funzione $readTxt()$ e memorizzati nella variabile $data$. Ogni elemento viene poi cifrato e confrontato con il ciphertext posseduto dal comando
$if(Objects.equals(encrypt(password.toString()), message))$
In caso la condizione sia soddisfatta viene stampata la password corrispondente e il tempo di ricerca.

2.1.2 Attacco su PINs generici

L'attacco su PINs generici è definito dai parametri $-m$"$String$" $-$ k"$String$" $-$ d. Anche in questo caso la chiave passata viene utilizzata per definire i cifratori. Ovviamente come prima davamo per scontato che la password fosse una di quelle contenute nel file $rockyou.txt$, ora consideriamo la password composta esattamente da 8 cifre $\in [0, 9]$. In seguito viene eseguito l'algoritmo di attacco:

```
System.out.println("Searching among all
   possible PINs...");
long start = System.currentTimeMillis();
for (int j = 0; j <
      Math.round(Math.pow(N.length, 8)); j++) {
   password = new StringBuilder();
   for (int x = 0; x < 8; x++)
      password.append(N[Math.toIntExact(
         j / Math.round(Math.pow(N.length, x)))
         % N.length]);
   if (Objects.equals(encrypt(
         password.toString()), message)) {
      long finish = System.currentTimeMillis();
      System.out.println("LA PASSWORD CORRETTA
         E': " + password);
      System.out.println("time: " +
         (finish - start) / 1000f + " sec");
      break;
   }
}
```

In questo attacco abbiamo un ciclo per $j \in [0, 10^8)$ ovvero un ciclo su tutte le disposizioni possibili. Tramite un sistema di resti (*for (int x = 0; x < 8; x++)*{..}) è possibile, a seconda di j, definire un numero a 8 cifre per cui:

$$j = 0 \Rightarrow password = 11111111$$
$$j = 10^8 - 1 \Rightarrow password = 00000000$$

Vengono in questo modo considerate tutte le disposizioni, e quando, come nell'if dell'attacco precedente, la cifratura del PIN corrente corriponde al ciphertext in possesso allora vengon stampati PIN corretto e tempo di ricerca.

2.1.3 Attacco alla chiave

L'attacco alla chiave è definito dai parametri $-m$"$String$" $-$ d. Rispetto agli attacchi precedenti la chiave non viene definita nei parametri, poichè essa viene cercata dall'algoritmo. Inizialmente vengono calcolati i trigrammi delle password all'interno di $rockyou.txt$:

```
readTxt(data, A);
StringBuilder tempKey;
long start;
System.out.println("Building trigrams...");
for (String datum : data) {
   List<String> triarray =
      Ngrams.ngrams(3, datum);
   for (String value : triarray) {
      triMap.computeIfAbsent(
         value, k1 -> new ArrayList<>());
      triMap.get(value).add(datum);
   }
}
```

Essi vengono registrati in una mappa $(trigramma, password[])$ la quale verrà utilizzata in seguito. Dopodiché viene eseguito l'algoritmo di ricerca. Esso si divide in due fasi. Nella prima viene calcolata la chiave e applicata ai cifratori del DES:

```
start = System.currentTimeMillis();
System.out.println("Searching for the key...");
Jaccard jaccard = new Jaccard();
for (int count = 0; count <
      Math.round(Math.pow(Q.length, 10));
      count++) {
   tempKey = new StringBuilder();
   for (int j = 0; j < 10; j++)
      tempKey.append(Q[Math.toIntExact(
         count / Math.round(
         Math.pow(Q.length, j)))
         % Q.length]);
   tempKey.append("0");
   tempKey.append("=");

   // generate secret key using DES algorithm
   byte[] encodedKey =
      Base64.getDecoder().decode(
      tempKey.toString());
   key = new SecretKeySpec(encodedKey,
      0, encodedKey.length, "DES");
   ecipher = Cipher.getInstance("DES");
   dcipher = Cipher.getInstance("DES");
   // initialize the ciphers with
   // the given key
   ecipher.init(Cipher.ENCRYPT_MODE, key);
   dcipher.init(Cipher.DECRYPT_MODE, key);
```

Tramite il sistemi di resti descritto nello scorso attacco vengono ciclate le possibili chiavi. Per semplicità non vengono utilizzate tutte le possibili stringhe di 10 caratteri, ma solo quelle che contengano determinati caratteri compresi nell'alfabeto Q. La seconda fase consiste nel confronto della decifratura del ciphertext in possesso e del processo di identificazione di correttezza:

```
result = decrypt(message);
if (result != null) {
   List<String> triArray;
   triArray = Ngrams.ngrams(3, result);
   ArrayList<String> possiblePassword =
      new ArrayList<>();
   for (String value : triArray) {
      if (triMap.get(value) != null)
         possiblePassword.addAll(
            triMap.get(value));
   }

   for (String value : possiblePassword) {
      if (jaccard.similarity(result, value)
            > 0.75){
         long finish = System.currentTimeMillis();
         System.out.println("Probably key: ...");
         System.out.println("time: " +
            (finish - start) / 1000f + " sec");
         count = (int) Math.round(
            Math.pow(Q.length, 10));
         break;
      }
   }
}
```

Nella variabile $triArray$ vengono registrati i trigrammi della parola decifrata, e in seguito nell'array $possiblePassword$ vengono registrati tutte le parole con almeno un trigramma comune con la password trovata. In seguito vengono controllati gli elementi di $possiblePassword$ e se uno di essi ha similarità di Jaccard superiore a 0,75 con la password decifrata allora consideriamo quest'ultima corretta, cosi come la chiave utilizzata. Questo attacco è significativamente più potente degli altri poiché trovare la chiave permette di potere cifrare qualsiasi plaintext cosi come decifrare qualsiasi ciphertext vista la perfetta invertibilità dell'algoritmo di cifratura.

2.2. Implementazione parallela

L'implementazione parallela di ogni attacco viene definita come in precedenza con l'aggiunta del parametro $-p$. La struttura dell'algoritmo parallelo è indipendente dal tipo di attacco, poichè tutti si basano su un ciclo for su un numero elevato di dati, il quale è banale da parallelizzare.

L'idea comune è dividere il suddetto ciclo in frazioni, ognuna delle quali viene eseguita da un thread della CPU:

```
long start = System.currentTimeMillis();
ArrayList<SearchThread> threads =
   new ArrayList<>();
int inizio , fine ;
for (int i = 0; i < numThreads; i++) {
   inizio = (data.size() / numThreads) * i;
   if (i == numThreads - 1) {
      fine = data.size();
   } else {
      fine = (data.size() / numThreads)
         * (i + 1);
   }
   threads.add(new SearchThread(inizio,
      fine, data, start, message));
}
for (int i = 0; i < numThreads; i++) {
   threads.get(i).start();
}
```

Viene innanzitutto inizializzato un array di threads. dopodichè tramite il ciclo for vengono distribuiti in parti uguali gli intervalli di indici che vanno divisi fra i vari threads, i quali vengono cosi inizializzati. infine vengono fatti partire. Quello che cambia tra un tipo di attacco e un altro è l'override del metodo run del thread. Sono stati implementati dunque tre tipi di thread, ognuno con il compito di eseguire uno dei tre specifici attacchi:

1. **SearchThread** per l'attacco su password note
2. **SearchAllThread** per l'attacco su PINs generici
3. **SearchKeyThread** per l'attacco alla chiave

Il metodo $run()$ dei vari thread ripresenta il codice già presentato nella parte sequenziale, a differenza dell'aggiornamento della variabile volatile $found$ la quale viene portata a $true$ una volta risolto l'algoritmo in modo da interrompere l'esecuzione di tutti i threads. la variabile numThread stabilisce quanti thread generare per svolgere l'algoritmo in parallelo.

3. Implementazione C++

L'implementazione dell'attacco al cifrario i C++ è stata fatta utilizzando la funzione della libreria *crypt.h* in cui sono presenti le funzioni di cifratura *crypt* e *crypt_r*. L'attacco implementato in questo linguaggio e quello che cerca di trovare il plaintext utilizzando un set di password note, nel nostro caso il file *rockyou.txt*. In particolare vengono cifrate tutte le parole del set comparando questa cifratura con il ciphertext dato. Entrambe le funzioni *crypt* e *crypt_r* non richiedono l'utilizzo di una chiave specifica da parte dell'utente, simulando quindi un possibile attacco ad una pagina di login, come riportato in precedenza.
In entrambe le versioni, sequenziale e parallela, viene creato un vettore stringhe contenente le parole selezionate dal file *rockyou.txt*. Inizialmente viene calcolata la lunghezza necessaria del vettore, quindi vengono qui aggiunte tutte le parole con un numero di caratteri ≤ 8. Quest'ultimo vincolo è dato dalla funzione di cifratura usata che impone che il plaintext sia di questo tipo. Il plaintext può essere fornito da linea di comando dall'utente o in modo statico dal programma. Il plaintext in ingresso viene quindi cifrato nello stesso modo da entrambe le versioni:

```
std::string salt = "LM";
std::string ct = crypt(pt.c_str(),
                       salt.c_str());
```

3.1. Implementazione sequenziale

Per trovare il plaintext corretto relativo al ciphertext in ingresso, è stata implementata la funzione *find_plaintext*:

```
std::string find_plaintext(std::string* vec,
                           std::string ct,
                           std::string salt)
```

Tale funzione prende in ingresso il vettore *vec* in cui cercare la il plaintext, il ciphertext *ct* obiettivo ed il *salt*. Questa funzione itera su tutti i dati del vettore cifrandoli e comparando la cifratura ottenuta con il ciphertext obiettivo *ct*. Appena viene trovato il ciphertext corrispondente, la funziona ritorna il plaintext associato.

```
int i = 0;
while (i < vec->length()) {
   if(ct.compare(crypt((vec[i]).c_str(),
                 salt.c_str())) == 0){
      return (vec[i]).c_str();
   }
   i++;
}
return "";
```

3.2. Implementazione parallela

Tale implementazione si basa sul fatto che questo problema di ricerca è imbarazzantemente parallelo in quanto questo tipo di ricerca non necessita di sincronizzazioni fra thread, se non quando uno di questi trova il plaintext corretto e notifica tutti gli altri. L'idea quindi è stata quella di assegnare ad ogni thread un upper e un lower bound del vettore in cui svolgere la ricerca. Per fare questo abbiamo utilizzato una versione di C++ ≥ 11

che ha il native support al multithreading. Abbiamo fatto eseguire ad ogni thread una sola funzione, *find_plaintext*, chiamata con *std::async* in modo da permettere l'utilizzo dei **futures** per ritornare il plaintext trovato.
La funzione *find_plaintext* prende in ingresso gli stessi parametri della funzione sequenziale aggiungendo anche il lower e l'upper bound del vettore di possibili plaintext in cui il thread deve cercare. Inoltre è condivisa fra tutti i thread una variabile booleana atomica *found* che indica se il plaintext è stato trovato o meno. Per garantire che l'esecuzione della funzione di cifratura fosse corretta per tutti i thread eseguiti in modo concorrente, abbiamo utilizzato la versione *reentrant* della funzione *crypt* usata nel programma sequenziale, ovvero *crypt_r*. La funzione implementata quindi presenta un ciclo sui plaintext assegnati al thread fino a quando nessun altro thread ha trovato il plaintext corretto. Questo check viene fatto controllando ad ogni iterazione che la variabile *found* sia *false* e che quando viene trovato il plaintext corretto questa venga modificata a *true*. Quindi viene comparato il valore del ciphertext in ingresso con il valore ottenuto dalla cifratura del plaintext corrente; se la comparazione da esito positivo viene ritornato il plaintext corrispondente oltre a settare *found* a *true*. Il codice ottenuto quindi è:

```
std::string find_plaintext(std::string* vec,
                           std::string ct,
                           std::string salt,
                           int lower,
                           int higher){
   struct crypt_data data;
   data.initialized = 0;

   int itr = lower;
   while (itr < higher && !(found.load())) {
      if(ct.compare(
               crypt_r((vec[itr]).c_str(),
                       salt.c_str(),
                       &data)) == 0){
       if(!(found.load())){
            found.store(true);
            return (vec[itr]).c_str();
         }
      }
      itr++;
   }
   return "";
}
```

Per gestire la sezione critica nell'assegnamento della variabile *found* l'abbiamo dichiarata atomica in modo che tale gestione fosse fatta dal compilatore. Per evitare che più thread possano ritornare un plaintext corretto, abbiamo aggiunto un ulteriore check su *found*; in questo modo l'ordinamento di eventuali letture o scritture multiple di *found* possono essere gestite direttamente dal codice e quindi non è necessario modificare il *memory_order* di default lasciando *memory_order_relaxed*.
Nel main viene lanciata la funzione su un numero stabilito di thread dividendo il set in blocchi di lunghezza uguale. Quindi viene atteso il risultato di ogni thread utilizzando dei *std::futures*. Questo ci ha permesso di rendere asincrono l'ottenimento del valore di ritorno del thread e di non dover scrivere il plaintext trovato in un riferimento di uno specifico parametro. In questo caso quindi abbiamo fatto eseguire ogni thread con *std::async* e lanciandolo in maniera asincrona invece che in maniera lazy aspettando che venisse fatto il *.get()* sul valore del future. Il codice ottenuto nel main quindi è:

```
int threadNumber;
if(argc > 1) threadNumber = atoi(argv[1]);
else threadNumber = 8;

std::vector<std::future<std::string>>futures;
   int incrAmount = (int)floor(length/
                               threadNumber);
   for(int itr=0; itr<threadNumber; itr++){
      int lower = incrAmount*itr;
      int higher = itr == threadNumber-1 ?
                   length :
                   incrAmount*(itr+1);
      futures.push_back(std::async
                        (std::launch::async,
                        find_plaintext, vec,
                        ct, salt,
                        lower, higher));
   }

   std::string foundPt;
   std::string tmpStr;
   for (int itr=0; itr<threadNumber; itr++){
      tmpStr = futures[itr].get();
      if(tmpStr != "") foundPt = tmpStr;
   }
```

4. Test

I Test effettuati sono stati divisi a seconda del linguaggio utilizzato, poiché i metodi di cifratura del des non potevano corrispondere esattamente l'uno con l'altro e i tempi sarebbero stati relativamente diversi.

4.1. Test Java

Questi test consistono nel confronto fra i tempi dei vari attacchi nei seguenti casi:

- $sequenziale$
- 2 $Threads$

PASSWORD	Key	Seq	2	4	8	Jaccard Coefficient	Similar Word
andrea98	55555555550=	18,613	11,254	6,990	7,168	0,8333	andrea9
Unicorno	11111111110=	0,097	0,090	0,96	0,102	0,8333	Unicorn
TurtlE	34455554430=	9,347	2,091	2,039	3,817	0,6	Turtle

Table 1. Tempi per l'attacco sulla chiave

- 4 $Threads$
- 8 $Threads$

I test inoltre sono stati eseguiti per elementi all'inizio della ricerca, in fondo alla ricerca e in posizioni centrali, in modo da verificare l'efficacia della parallelizzazione.

PASSWORD	Seq	2	4	8
parallel	2,367	1,416	1,012	1,349
123456	0,005	0,008	0,008	0,009
havushi	0,496	0,555	0,768	0,542
ac3127	1,773	0,697	0,946	0,963
mritaly	1,061	1,075	0,720	0,503

Table 2. Tempi per l'attacco su password note

PIN	Seq	2	4	8
11111111	0,001	0,003	0,004	0,005
00000000	59,991	30,191	16,541	14,836
24101998	44,418	16,417	2,614	5,278
11111116	30,596	0,003	0,003	0,004

Table 3. Tempi per l'attacco su PINs generici

Per quanto riguarda i test dell'attacco su password note sono state scelte: $parallel$, parola finale del file di rockyou, 123456, parola iniziale del file di rockyou, e tre parole in posizioni casuali nel file. La vera efficacia della parallelizzazione si può notare per i tempi della parola in fondo all'elenco poichè e necessario visitare tutti gli elementi dell'array. Questa situazione vantaggiosa si può facilmente notare anche per gli esperimenti degli altri attacchi.

Lo stessa filosofia è stata utilizzata per gli altri due attacchi. Nell'attacco su PINs generici 11111111 rappresenta il primo elemento di ricerca, 00000000 rappresenta l'ultimo, mentre 11111116 rappresenta l'elemento esattamente al centro della ricerca. Quest'ultimo ha difatti un particolare comportamento nell'esecuzione parallela, avendo in ogni esecuzione il tempo relativamente nullo poiché uno dei thread lanciati partirà la ricerca esattamente da quell'elemento.

L'ultimo attacco prende come esempi le chiavi agli estremi della ricerca e una casuale (34455554430=) ognuna affiancata da una password differente per verificare il funzionamento della ricerca via trigrammi. Per la verifica dell'ultima password la soglia di similarità è stata abbassata a 0.5 poiché per parole più corte la somiglianza di trigrammi diventa molto più difficile.

4.2. Test C++

Sono stati svolti 3 tipi di test: uno sulla prima parola del set **123456**, uno sull'ultima parola del set **parallel** e uno su varie parole scelte in modo casuale **hawker1 ac3127 mritaly**. Tutti questi test sono stati eseguiti con la versione sequenziale e con la versione parallela con 2, 4, 8, 16, 32 thread. I risultati ottenuti sono riportati nella Table 4.

- **123456**: vediamo che la versione sequenziale ha un tempo di esecuzione molto minore in quanto non necessita della gestione dei thread. Più il numero aumenta, più aumenta anche il tempo di esecuzione in quanto deve essere lanciato un thread in più;
- **parallel**: il tempo di esecuzione in questo caso diminuisce all'aumentare dei thread impiegati. Infatti dato che parallel è l'ultima parola, il fatto che i thread lavorino su un set grande $\#set/numero\_thread$ implica che ogni thread debba ricercare in un set molto più piccolo. Notiamo infatti che i tempi di computazione all'incirca si dimezzano all'aumentare dei thread, coerentemente con le attese;
- **hawker1, ac3127, mritaly**: in questi tre casi vediamo come i tempi di computazione varino molto in base alla posizione della parola nel set: se il plaintext obiettivo è molto vicino all'indice di inizio di un thread, allora il thread eseguirà molto più velocemente delle versioni precedenti. I risultati non sono totalmente conformi a quest'ultima considerazione. Probabilmente ciò è dato dal fatto che i cores all'interno di una stessa CPU sono diversi e con diverse capacità computazionali.

4.3. Hardware utilizzato

L'hardware utilizzato per la computazione in Java è:

- Chip Apple M1: 8 cores;
- Ram: 16 gb.

L'hardware utilizzato per la computazione in C++ è:

- **Intel Core i7-1065G7**: 4 cores, 8 threads, 1,30 GHz;
- Ram: 16 gb.

PLAINTEXT	Seq	2	4	8	16	32
123456	8.7e-06	3.3e-04	4.2e-04	6.1e-04	7.6e-04	1.0e-03
parallel	21.3	11.7	6.9	4.7	4.5	4.4
hawker1	11.7	1.3	1.5	2.1	4.2	3.6
ac3127	15.2	5.4	6.3	4.0	3.0	1.3
mritaly	8.6	9.7	4.5	1.2	2.4	0.3

Table 4. Tabella che mostra i risultati ottenuti nei test in C++. Sulle righe sono riportate le parole cercate, sulle colonne la versione eseguita: se sequenziale o a 2, 4, 8, 16, 32 thread.

5. Conclusioni

In linea con il fatto che il problema dell'attacco al cifrario DES è imbarazzantemente parallelo, come atteso, le versioni parallele sono molto più veloci delle versioni sequenziali. Le performances dei cores nei processori possono talvolta influenzare tali risultati, ma fondamentalmente l'evidenza dei miglioramenti rimane.