Password decryption with Java Thread

Giulio Bazzanti, Niccolò Biondi
Dipartimento di Ingegneria dell'Informazione (DINFO)
University of Florence, Florence, Italy
giulio.bazzanti@stud.unifi.it, niccolo.biondi1@stud.unifi.it

***Abstract*—In Network Security, a brute force attack generally indicates the method used by an Hacker to identify a login password analyzing exhaustively all the possible strings combinations allowed by the particular system. In this project we want to study the possible improvements that the use of parallel approach introduces in this kind of attack. To do this, we implement a Sequential approach and a Java Thread one to decrypt an alphanumeric 8-characters password. We evaluate the reported Speedup and Efficiency for different Java synchronization methods: Callable, Runnable, ReadWriteLock and Atomic.**

I. Introduction

In cryptography, a *brute-force attack* is a *cryptanalytic attack* that can be used, in theory, to attempt to decrypt any encrypted data. During this attack, the hacker submits many passwords or passphrase with the hope of guessing it. This kind of attack can be used when is not possible to take advantage of other vulnerability of the encrypting system that the hacker wants to attack.

Generally, eight is the standard number for the shortest length of a password. Furthermore, several alphabets are added within a password to make it more secure and complex. Considering an alphanumeric 8-characters password, the possible combinations, using the set [a-zA-Z0-9], that could be made are 68^8 (26 lower case English characters, 26 upper case English characters and 10 numeric digits). As we can assume, this approach takes a lot of time: it can take seven million years to crack an alphanumeric 8-characters password.

The aim of this project is to study a brute-force attack using a Java Thread approach to decrypt password protected with the *DES* crypting algorithm and comparing it with a Sequential (Java) standard implementation. Since the brute-force attack needs huge amount of computational time, as we describe above, to simulate the entire process we adopt another the *dictionary attack*. In this kind of attack the hacker has a set of dictionaries used to find the protected password.

II. Methods

In this section we present our approaches, from the adopted dictionaries generation to the experiments configuration. We also briefly describe the different proposed implementations for the parallelism technique.

A. Dictionary Generation

A brute-force attack takes a lot of time, since it analyze all possible combinations to find one password. In order to reduce the time spent in this search, the hacker can use a dictionary: a big file containing all possible password that a common user can adopt. Typically most of the adopted passwords are something that a user can easily memorize. Statistically, a 8-characters password can figure a date of birth or, for example, the name of a user pet with some number at the end. Following this reasoning, we create two different type of dictionary. The former is create with all possible date (we call it *Dictionary_data*) from the year 1600 to 2020 in the tree possible format: day-month-year, year-month-day, month-day-year. We decided to choose the year 1600 as the starting year to have an acceptable number of dates to analyze. The latter is create, instead, taking as a reference a dictionary of all 8 letters long English words, from witch we create three different type of dictionaries: one with capitalized words, one with numbers instead of vowels and one with four random numbers instead of the last four word letters (we call it *Dictionary_words*). Another difference from the Dictionary_data is the number of words that compose it. In the Dictionary_words, the words number is much more than the Dictionary_data one. This was done to see how the brute-force attack computational time depends on the dictionary size.

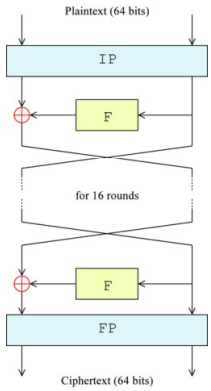

Fig. 1: DES algorithm structure.

B. Data Encryption Standard

Data Encryption Standard (*DES)* is a block cipher algorithm, which takes as input a plain text string and returns it encrypted. The algorithm is composed by sixteen identical phases called *round* or *cycle*. There are also an initial and final permutation called *IP* and *FP* respectively. Before the main cycle, the block is divided into two 32-bit halves and processed alternately. The rest of the algorithm remains identical. The

DES structure is represent in Figure 1. The DES algorithm is implemented using a Java extension called *javax*. This includes the extension *javax.crypto*, a package that defines classes and interfaces for various cryptographic operations. The central class is *Cipher*, which is used to encrypt and decrypt data. The *KeyGenerator* class creates the *SecretKey* objects used by Cipher for encryption and decryption.

In this project, this algorithm is used to encrypt the password that a possible hacker wants to find. To achieve that, is created a *secret key* using a KeyGenerator object that a Cipher leverages to encrypt the target password. After it, a brute-force attack adopts a dictionary is performed to find this secret password.

III. Sequential Implementation

In this section is presented the sequential implementation, that we develop to simulate a brute-force dictionary attack.

Giving a dictionary, the sequential implementation processes one password at once checking if it is the same as the target one. To do that, we suppose that the hacker knows somehow the secret key used by the DES algorithm. So after loading the entire Dictionary used to the attack, the sequential implementation is composed by a simple for loop with which the passwords are processed until the hacker not find the secret one. A password p is encrypted through the secret key finding the encrypted password ep. After it, is checked if ep is equal to the encrypted target password. If it is true, the for loop is stopped, returning eventually the password.

The problem of this implementation is the dictionary length. As we introduce previously, the brute-force attack is a temporally ineffective algorithm because the hacker has to check all possible combinations to find a single password. The computational time can be reduced using a dictionary containing a certain number of passwords but the longer the dictionary is, the longer it will take to check it all. For more detail check Section V.

So the total time for this implementation is $O(l)$, where l is the dictionary length.

IV. Parallel Implementation

In this section we introduce our brute-force attack implementations with CPU parallel computation, based on the Java threading model. The Java threading model is based on two fundamental concepts: shared, visible mutable state and preemptive thread scheduling. Following these two concepts a thread can change an object and also an object is easily shared between all threads. Furthermore, a thread scheduler can swap threads on and off cores at any time, more or less. There is two way to execute a pool of thread. The former is to manually create and start all single thread. The latter is to use a special class called *Executor*. The Executor separates threads management and creation from the rest of the application. It creates them from a pool of threads keeping the overhead low avoiding the creation of an excessive number of threads. The parallel implementation follows this second threads creation approach: given a maximum threads number, an Executor creates threads from a pool assigning them a task to perform. In this scenario, the task is a brute-force attack following the same sequential implementation idea. Each thread has to find the encrypted target password using a dictionary chunk, that is different for every threads. So taking the original dictionary, we divide it in several chunks, depending on the threads number, giving them a chunk to analyze. When a thread finds the target password has to communicate to other threads to stop their job. To do this, we implement three different synchronization approaches: *Atomic*, *ReadWriteLock* and *Thread Synchronization*.

A typical problem using threads is the *race condition*: more than one thread tries to access a shared resource. To solve this kind of problem we have to make sure that when a thread uses the shared resource, no other thread can use this. However, in our approaches, it is difficult to produce a race condition, because only one thread (the one which finds the password) will communicate to the other threads to stop. Anyhow, it is good practice to use mechanisms to synchronize access to shared resources, so we adopt different kind of solutions to avoid a possible race condition.

In all the approaches we use Callable and Runnable that are both interfaces designed to represent a task that can be executed by multiple threads.

The Runnable interface has a single *run()* method which does not accept any parameters and does not return any values. The Callable interface, instead, contains a single *call()* method, which returns a generic value V. So the main different between these two kinds of approach is the possibility to return the task result. To encapsulate this value, it is used a *Future*: a sort of container that take the result of a Callable task.

A. Thread Synchronization

Synchronization in java is the capability to control the access of multiple threads to any shared resource. The synchronization is mainly used to prevent threads interference and to prevent consistency problems. To do this, the keyword *synchronized* is used, that leverages the lock that is built into every Java Object. A method can be defined as synchronized, so one thread can use this method at once. Until the method do not end, the lock is not release.

In our approach, we define a class called *State* where we put the shared resource with which the threads can communicate. Inside the State class we define a boolean variable (True/False variable) which indicates if the password was found (True) or not (False).

To synchronize the threads access to the shared resource, we define two different methods in the State class. The former, called *getState()*, it is used to read the shared variable and check if the password is found or not. The latter, called *setState()*, it is used by the thread that finds the target password. The function set the boolean variable to True, stopping the other threads.

B. ReadWriteLock

ReadWriteLock is a Java interface that gives an advanced thread lock mechanism. Using the class *ReentrantReadWrite-*

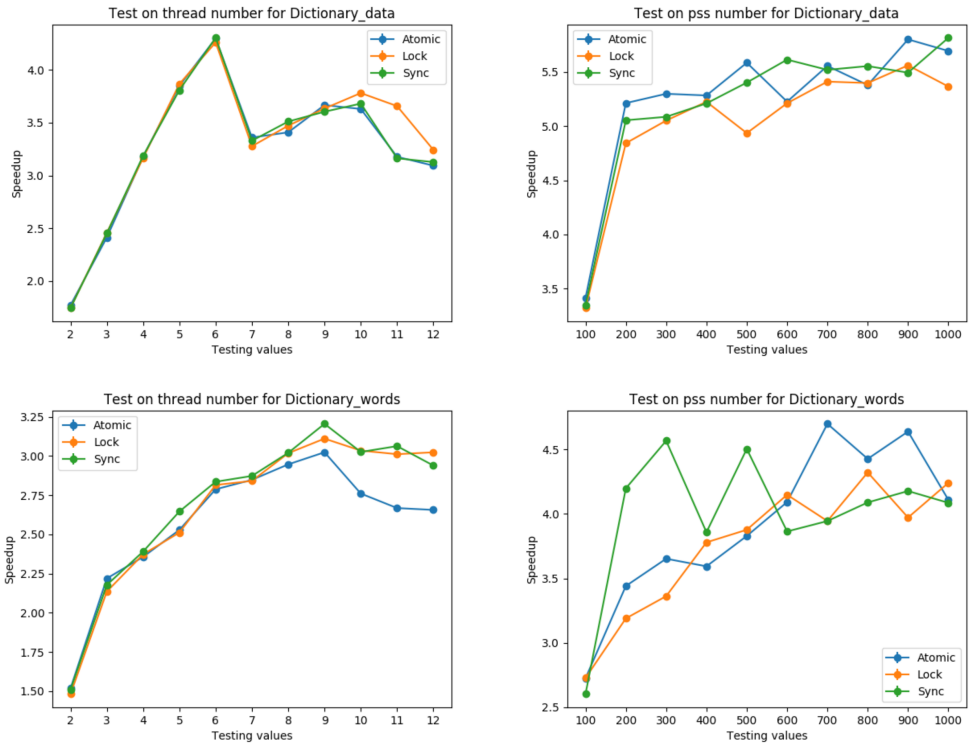

Fig. 2: In this four plots is reported the experiments Speedup. With *pss* we indicate the passwords number increasing and with *thread* the thread numbers increasing. The *Dictionary_data* is a dictionary with only birth date and *Dictionary_words* is a dictionary with only English words.

Lock, it is possible to allow multiple threads to read a certain resource, but allow only one to write it at once. This class has two types of lock: one for read and one for write. The idea is that more than one thread can use read operations simultaneously, but only one thread can use write operations. When a thread is doing a write operation, there can't be any threads doing read operations. In this way a thread can read the true shared resource state. When a thread wants to access to a shared resource, it has to acquire the lock and then it has to release it. The main problem of this kind of approach is that if one thread do not release the lock, due to a possible error, the other threads will not be able to acquire the lock, causing a *dead lock*.

Like is describe in Section IV-A, we define, also in this approach, the same class called *State*. This class has the same features as the Thread Synchronization class, but in this case we use a ReentrantReadWriteLock class to acquire and release the lock both in reading and writing.

C. Atomic Variable

An *Atomic* is another kind of threads synchronization. A variable declared as Atomic can be accessed and modified atomically, without need of synchronization or locks. The main different with a *Volatile* variable is that this ensures the visibility of changes of the shared variables across threads, while Atomic ensures that operations on those are performed atomically. In both of these two approaches, the reading and writing operations are atomic but with a Volatile variable it is not guarantee mutual exclusion. Thus the main problem in Volatile is the variable update: a thread with one operation has to read the shared resource state, changing the variable value and writing it in memory. So two threads at same time can each perform these operations and generating data inconsistency. Using the atomic variable, we can implement non-blocking, lock-free algorithms and solve a potential race condition problem.

In our approach, the Atomic variable is the boolean variable that is included and defined, in the other approaches, inside the State class. Following the previous reasoning, when this variable is True, it means that a thread found the target password. This condition stops the other threads, ending the entire brute-force attack. In this case, the Atomic class gives to us two different functions to interact with the shared resource. The former is the function *get()* that returns the value from the memory. The latter is the function *set()* that writes the value to memory.

V. Experimental Results

In this section we discuss about the experiments that we bring during this project. In addiction to these we talk about the evaluation metrics and our setup. In all the experiments we

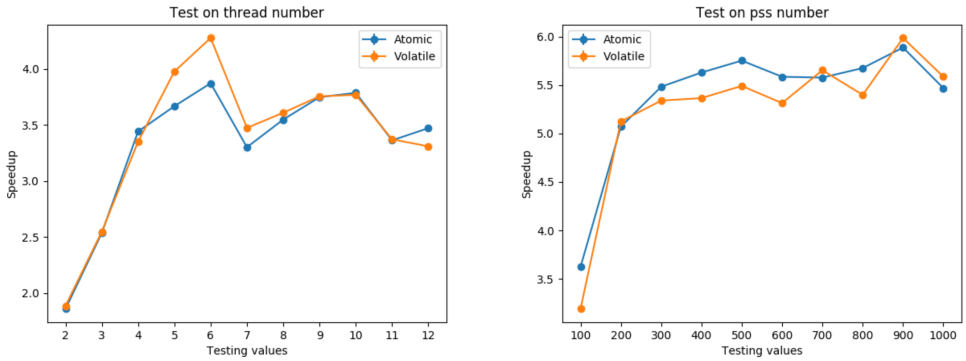

Fig. 3: Speedup plots to compare Atomic with Volatile approach using Dictionary_data

suppose that there are multiple passwords that an hypothetical hacker has to find.

A. Experiments Configuration

We conducted our Sequential and Java thread experiments on a Intel i7-8750H CPU which has $6+6$ (physical and virtual) cores.

To validate our results, we make two important assumptions. The former is that the hacker has to find more than one password. We make this decision because, in this way, we can report a mean of the computational times that all the approaches, defined in Section II, employ to find several passwords. Thus the reported values do not depend on the passwords positions in the dictionary giving a mean time of the entire process. The latter is based on the multiple replications of each one trial several times to report *Speedup* values those are means with their corresponding standard deviations (std).

B. Performances Evaluation Metrics

To evaluate properly an experiment, we report two different metrics, i.e Speedup and Efficiency. The former is computed as follows:

$$S_p = t_s/t_p \tag{1}$$

where t_s is the sequential time, t_p is the parallel time with p processors and S_p is the resulting Speedup.

The Speedup can be super-linear ($S_p > p$), linear ($S_p = p$), sub-linear ($S_p < p$) or without Speedup ($S_p < 1$). Our results often report sub-linear Speedup.

The Efficiency of an algorithm using p processors is:

$$E_p = S_p/p. \tag{2}$$

This is a value in $[0, 1]$ and it measures how well-utilized the processors are in solving the task.

C. Experiments

The brute-force attack can not be studied in a easily way, for the problematic described in the Section I. For this reason, we decide to study this type of attack considering: different number of passwords that a hacker has to find, different synchronization approaches, two different dictionaries and different threads number.

As we specified above, to validate our experiments we decide to replicate each one trial several times and also we decide to make two different kind of tests. The former where we change the threads number, giving a fixed passwords number (we call it *thread_test*). The latter where we change the passwords number, giving a fixed threads number (we call it *pss_test*). To do that, we set some hyper-parameters in order to specified the test to perform. One default parameter is 100 for the passwords number used in the thread_tests and for the starting number of passwords in pss_tests. We use two different types of dictionary: the *Dictionary_data* composed by 461313 of data from year 1600 to year 2020 and the *Dictionary_words* composed by 19958400 possible 8 letters long English words. These two dictionaries have different number of words because, in our experiments, we want to underline the time inefficiency of a generic brute-force attack. The number of threads ranging from 2, the starting threads number in the thread_test experiment, to 12 threads, maximum threads number used in both thread_test and pss_test experiments. The synchronization methods adopted are the three methods defined in Section IV. At the end, we always perform 10 runs with the same configuration for computing mean and std of reported Speedup to extract more stable results. Thanks to this, we can report in our experimental results values with small std.

First of all, in order to understand the differences between the use of Callable and Runnable, we perform some tests to discover the different performances adopting the two methods. In these experiments, we notice that using Callable or Runnable does not affect the reported values. So we decide to use the Runnable interface for all the following experiments.

As we report in Figure 2, we perform four type of experiments, two for each dictionaries. For the Dictionary_words, the two figures on the button in Figure 2, we obtain a Speedup values that are lower than the Speedup values we obtain for the Dictionary_data, the two figures on the top in Figure 2. This is caused by the time inefficiency of the brute-force attack, that to find a password it has to check all possible combination,

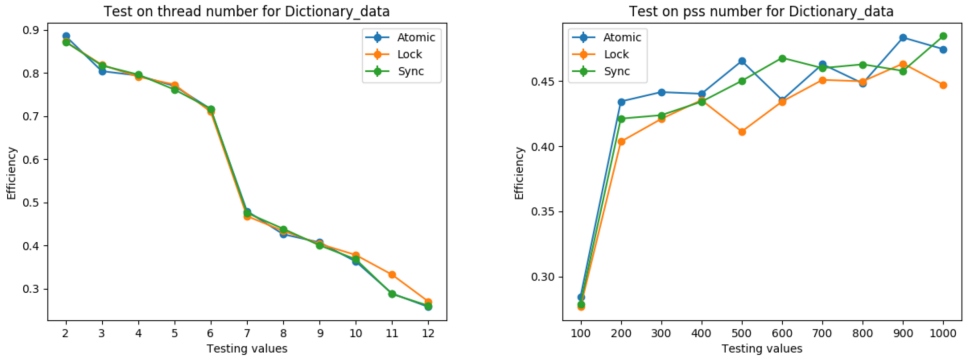

Fig. 4: Efficiency plots using Dictionary_data for different threads number (left plot) and different passwords number (right plot).

using a set of symbols. We obtain the same time inefficiency using different type of dictionaries. The longer the dictionary is, the more it will take to control it, reproducing the same behaviour as a normal brute-force attack.

The Dictionary_data has inside much fewer values than Dictionary_words, thus greatly reducing the time taken to find the individual password, because all single thread has to analyze a smaller dictionary chunk than the one analyzed for the Dictionary_words. Another important result is obtained going to change two hyper-parameters: the threads number and the passwords number. If we compare the two figures on the right with the two figures on the left in Figure 2, we can see that changing the passwords number, considering a fixed number of threads (the two left figures), we obtain a higher Speedup than changing the threads number, considering a fixed number of passwords (the two right figures).

Increasing therefore the passwords number, each thread will find within its dictionary chunk a greater passwords number. This leads to a reduction in the overall search time, in addition to a performance increase. At the other hand, increasing the thread number leads to a decrease in the time needed to analyze the entire dictionary, as the number of chunks into which it is divided increases.

We also compare the reported performances adopting a Volatile shared variable instead of an Atomic one. As we can see in Figure 3, the Volatile variable approach reports the same results of the Atomic variable ones. Since just one thread updates the shared variable, it is not possible for a race condition to occur. Moreover, the little threads synchronization of this task, that is a single update, makes the Speedups have the same trend. The little gap that our experiments report is due to the random passwords positions in dictionaries. Indeed if the passwords are at the beginning of the chuck, the thread that analyzes it will find those very quickly.

Finally we also evaluate our implementations in terms of Efficiency and we report the obtained results in Figure 4. Here we study this performances measure in the Dictionary_data both changing the threads and the passwords number. As regard the threads number with fixed passwords to find, the efficiency decrease at the increase of the used processors. This is because each thread will have a less chuck of data where there could be less passwords to retrieve. This leads the Efficiency to assume a descending trend. When we evaluate the Efficiency on the passwords number (with 12 threads for each test), the reported results are the opposite. The Efficiency plot has an ascending trend because increasing the passwords number we increase the probability of finding more passwords in a single chuck.

VI. Conclusions

In this paper we implement a Java Thread parallel approach to reproduce a brute-force attack using the DES algorithm. To do this we adopt two dictionaries with which we perform several experiments to study how their size affect the computational time of this task. As we imagine, the parallel approach breaks down the brute-force overall time with respect to the size of the adopted dictionaries. The sequential brute-force method has to analyze every single passwords contained in high dimension dictionary, unlike the parallel approach where the dictionary is fragmented into much smaller chunks.

An important obtained result is about the synchronization methods: changing them do not impact much to the final results. This is due to the fact that, in this problem, the shared resource is only write by the thread that finds the final target password, completing the attack. So in this scenario, it is impossible to have race condition or dead lock making possible to leverage a Volatile variable to synchronize threads. The performances gap between these implementations is caused by the random choice of the passwords to retrieve. They could be in good positions in the respective chunks making them find faster by threads. The real problem is inside the problem itself: brute-force attacks are computationally expensive. A parallel approach can reduce the computation time, but it all depends on the length of the password to be found and the size of the adopted dictionary.