COMP522 Privacy and Security

Assignment 1: Brute-force search attack on the password-based encryption

Maciej Lechowski

m.j.lechowski@liverpool.ac.uk

1 Description of the classes

Program I have written consists of four public classes.

`Main.java` - where settings can be specified and methods from other classes are called,

`Encryption.java` - where code regarding encrypting and decrypting a message is located; there is also an inner class called `EncryptionData`, where certain parameters concerning encryption are kept,

`BruteForce.java` - here is where brute-force search attack is carried on; also in this class all possible combinations of given character set are generated (algorithm for doing that is explained in section 4),

`Utils.java` - utility class in which method converting byte array to hexadecimal representation is declared.

2 Running the program

The program was tested on Linux (Ubuntu 14.04).

To run the program in Linux go to bin/ directory:

```
$ cd bin
```

Then, call the main class specifying the package in which it is stored:

```
$ java com.lchsk.Main
```

After that, program should run with default settings.

3 Settings

To complete different parts of the assignment and run experiments I have declared a number of settings to specify the way I want the program to work. All of them can be easily changed in `Main.java` file at the top of the Main class.

Here's a summary of all the settings I have used.

Setting name	**Description**
`characters`	It tells the program which characters should be used to generate all possible combinations. By default only digits are used, but all other characters are possible, including letters.
`stopAtSuccess`	It is a boolean variable, which if set to true, stops the execution of the program when the password is broken. If set to false, all possible combinations are used even after finding out what the password was (useful when measuring how long it takes to generate all possible combinations).
`askForPassword`	If set to true, user will be asked to specify the password that will be used to encrypt the message. Then the user will be asked to provide the password again and the message will be decrypted. This setting is essentially for demonstrational purposes only.
`defaultPassword`	If parameter `askForPassword` is set to false, the `defaultPassword` will be used for encryption. I used this settings when conducting experiments.
`secret`	Message that will be encrypted.
`printInformation`	If set to true, all combinations used to break the password will be printed to console (for the user to check if everything is correct).
`maxPasswordLength`	Specifies what is the password length limit for the brute-force search. Algorithm will try all combinations of characters from length 1 up to `maxPasswordLength`.

`guessIterationCount`	If set to true, brute-force search will also iterate over iteration count (from 1 up to `maxIterationCount`).
`maxIterationCount`	Specifies what is the limit for iteration count (used only if `guessIterationCount` is set to true).

Table 3.1. Description of settings for brute-force search attack program

4 Algorithm for generating all possible combinations of a given character set

Although the assignment expected the program to use only digits while trying to break the password, I was interested in how different character sets affect the time needed to find the correct password. I also wanted to have flexibility in this approach, in order to be able to specify only certain characters (eg., only a few letters). To achieve that, I have prepared an algorithm that is able to generate all possible combinations of specified character set (given the combination length).

The idea is based upon an algorithm used to change representation of a number to a different positional system. Algorithm treats length of the character set as a base of the positional notation (ie., a radix). Then it changes representation from decimal to base-N system, where N is the length of the character set. For instance, if chosen character set consisted of letters `a` and `b`, then representation would be changed to base-2 (ie., binary). After that, the newly obtained number in base-N representation is translated in a way that only letters from the character set specified in settings (`characters` variable from Table 3.1) appear in the returned variable.

Example 4.1.

Let's assume that only letters `a` and `b` can appear in a password. Therefore:

`characters = {a, b}` and `length(characters) = 2`. Now we can take any decimal number, say 5, and change it into binary. After that, we can translate every number (ie., the computed binary representation) using the `characters` variable (which serves as a dictionary in this case). So in this simple example, all 0s and 1s would be translated to a's and b's, respectively.

So in the end, number $\mathbf{5_{10}=101_2}$, therefore letter representation would be **bab**. In this way, by iterating over decimal numbers and changing their representation, we can

obtain all combinations of a particular character set.

Pseudocode 4.1.

Pseudocode for algorithm generating all possible combinations is as follows.

`maxLength` - maximum length of the password

`n` - length of the character set (which serves as a radix)

```
for 1:maxLength loop
      while not found every possible combination
            increase i
            r = get representation of i in a base-n system
            c = translate from r using characters from dictionary
            // c now holds a unique combination
            // now we can use c for encryption to check if it is a password...
```

In code

In the attached source code, the discussed algorithm is located in `BruteForce.java` file, where:

`attack()` - is a method where all possible combinations are tried to break the password,

`repr(n, r)` - is a method used to change the representation of a number `n` given the radix `r`,

`getChars()` - is a method which serves as a translator of the value obtained from `repr()` method.

5 Experiments

All experiments described below were conducted on Linux (Ubuntu 14.04) with Java 1.7.0_65 installed. The computer on which the program has been run has Intel Core i5-4210U (dual-core) processor and 8 GBs of RAM.

5.1. Measuring time needed for encryption using all possible combinations of digits

Because algorithm discussed in section 4 is deterministic, it does not make much sense to measure time looking for a specific password. The reason for that is straightforward: combinations of digits are always generated in the same order. For instance, a combination "123456" would be generated before "987654", therefore in presented approach it would always take significantly more time to find the latter. That is why I have decided to measure the time needed to find *all* possible combinations (ie., using flag `stopAtSuccess` set to false, as discussed in section 3). This approach essentially tells us what the worst-case scenario is, given particular character set.

Experiments were conducted without printing information to the console (ie., with `printInformation` set to false). Printing all information tend to increase search time.

All charts in this section show results, where all possible combinations of digits are generated.

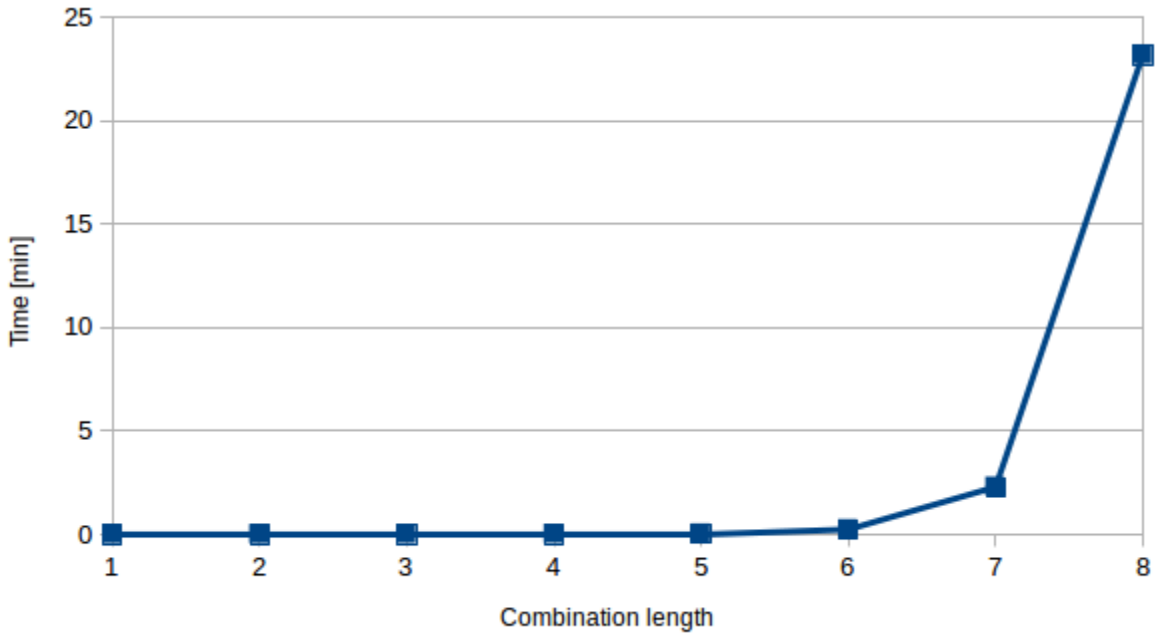

Figure 5.1. This chart presents how execution time grows for increasing combination lengths. In this example, iteration count is very small (24). As shown above, even for small iteration count, time grows rapidly from a certain point (significant increase

begins with length equal to 7 in this case). I would argue that for lengths bigger than 8 time needed for computing all combinations would surpass 1 hour.

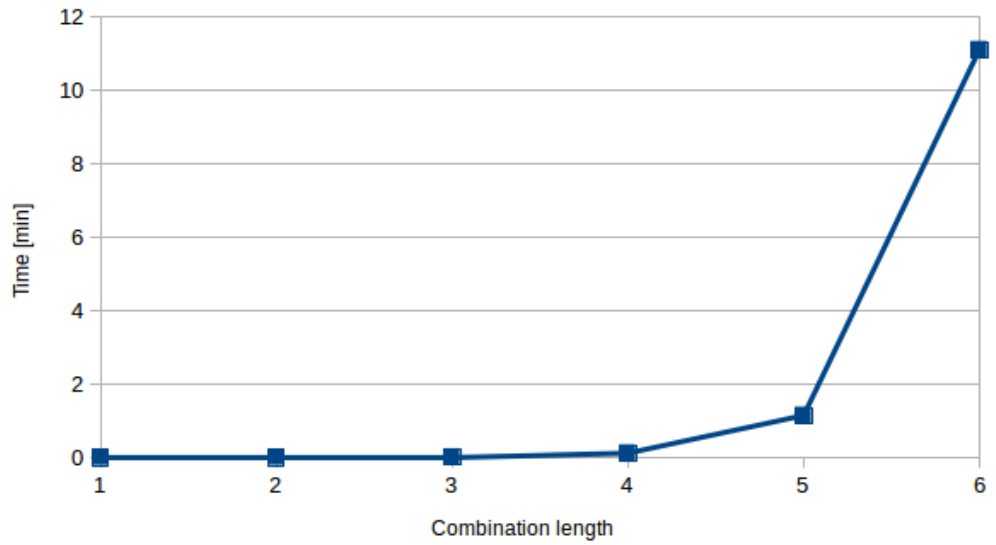

Figure 5.2. In this case, iteration count was increased to 2048. Now growth begins with length 5 and at 6 it becomes significant. Combinations of length 7 would most certainly be generated in more than 1 hour.

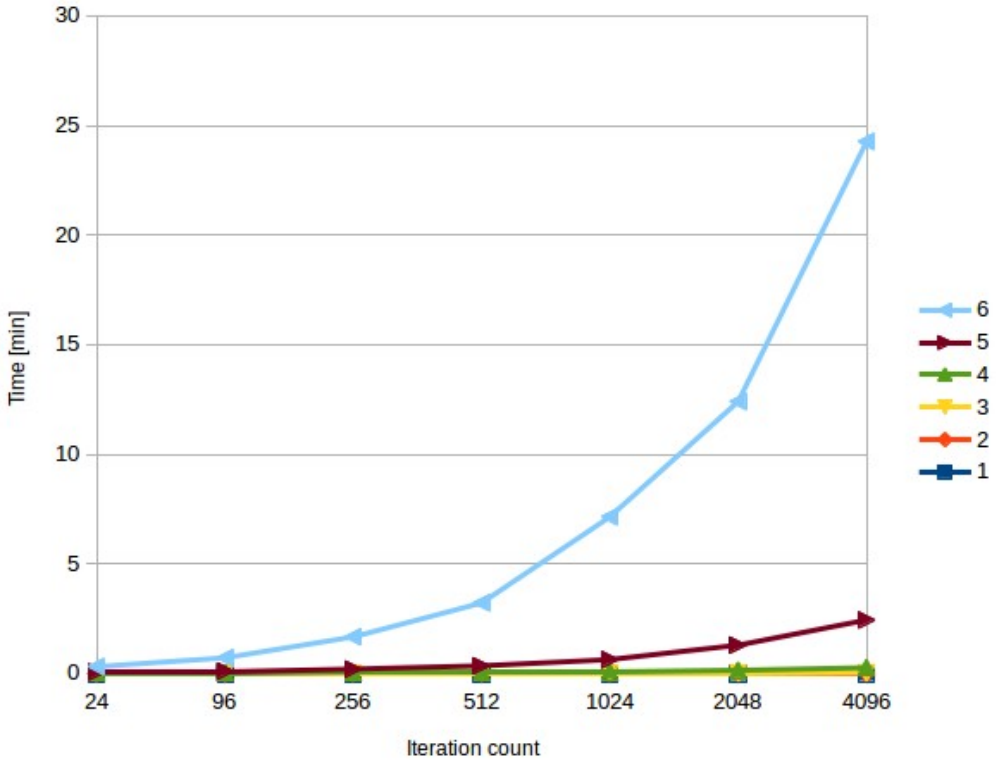

Figure 5.3. The chart shows how time needed for computation is dependent upon iteration count. Each line shows results for different lengths (as described in the

legend). Essentially, iteration count massively impacts time (as shown here for lengths 5 and 6).

5.2. Measuring time for smaller character sets

I found it interesting how length of the character set affects search time. In the previous example we've used digits (ie., 10 characters). In the following examples, different character sets are used. For example, length 4 means that only letters a, b, c and d could appear in passwords. In all examples in this section iteration count = 2048 was used.

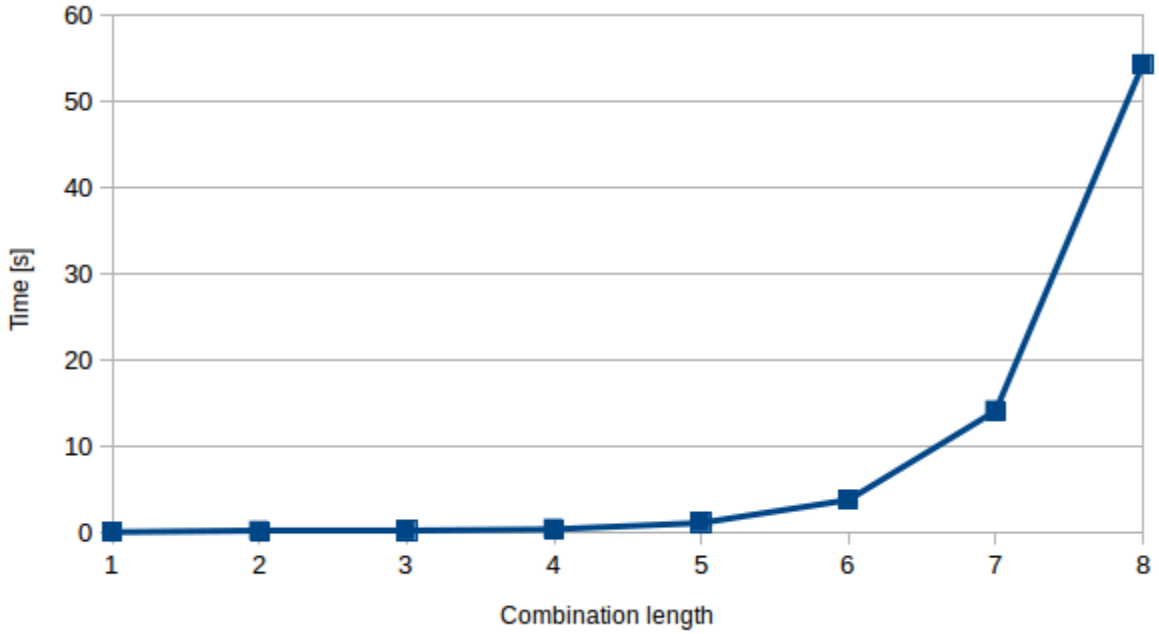

Figure 5.4. The chart shows how fast search time grows for character set consisting of four letters (a, b, c, d) when increasing maximum password length. Clearly, even for that short, impractical character set, search time grows rapidly with longer passwords.

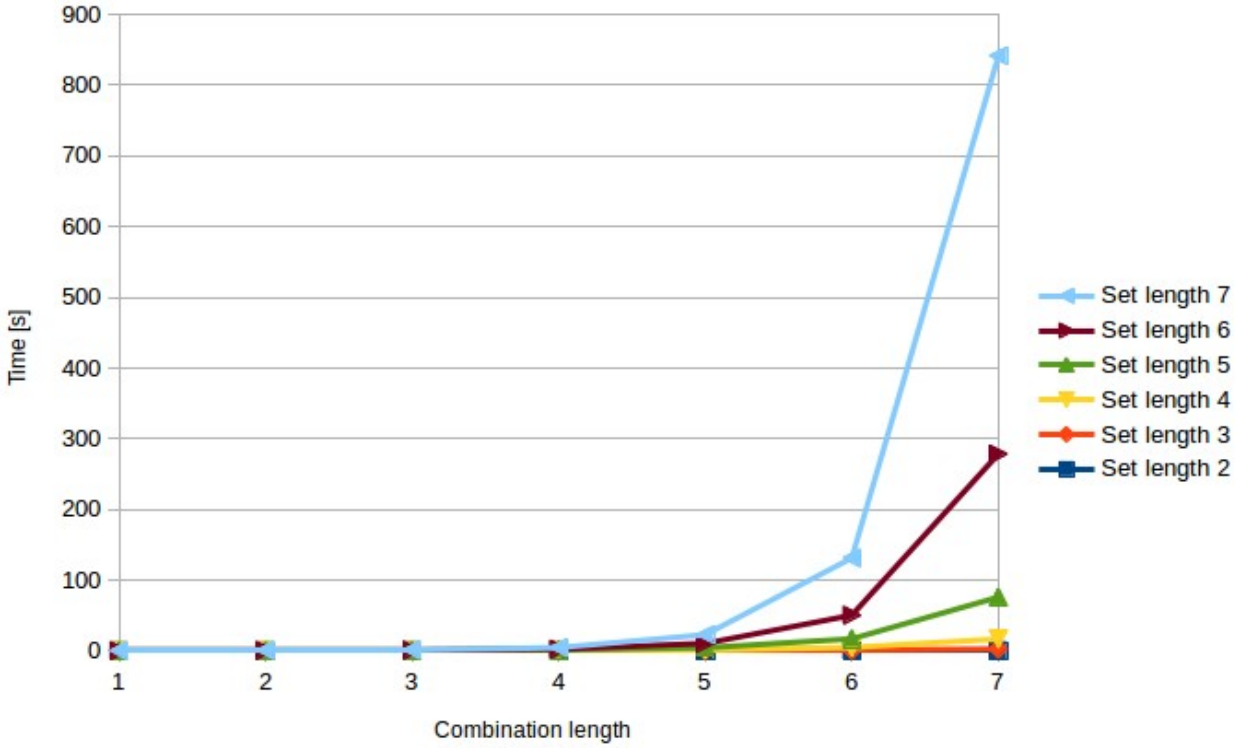

Figure 5.5. This chart presents how set and combination lengths affect search time. Having bigger character sets, along with longer passwords, clearly increase time needed to undermine security.

5.3. Measuring time in more practical example

Examples discussed in the previous sections used very small character sets. In this section I have used bigger ones. And although, for lack of computational power, I've only been able to check short passwords, it is evident that more real-life character sets greatly impact time needed for brute-force search.

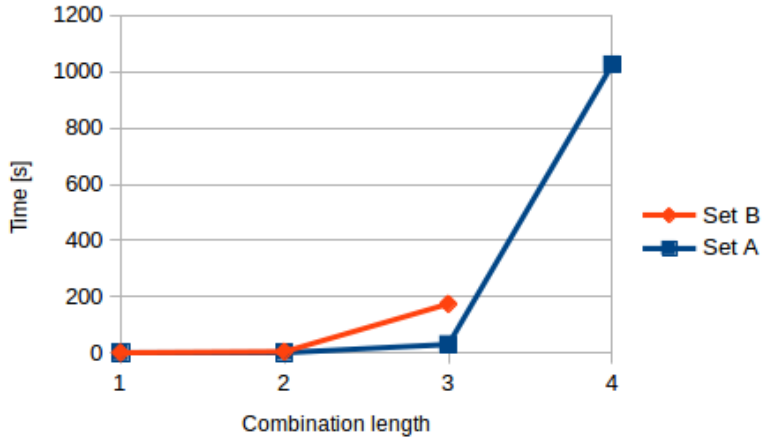

Figure 5.6. In this case, two character sets were used. First one (set A), consisted of all lowercase letters of the English alphabet as well as digits (total 36 characters). The second one (set B) consisted of both lower and upper case letters plus digits (total 62 characters). Even for very short passwords, search time grew rapidly. For lack of computational power I was not able to measure time needed to find 4-character password for set B. I argue that it would take several hours to finish.

5.4. Measuring search time in case where iteration count is unknown

In this case, program needs to additionaly loop over iteration counts, which obviously increases computation time. Character set used in this section consists of digits.

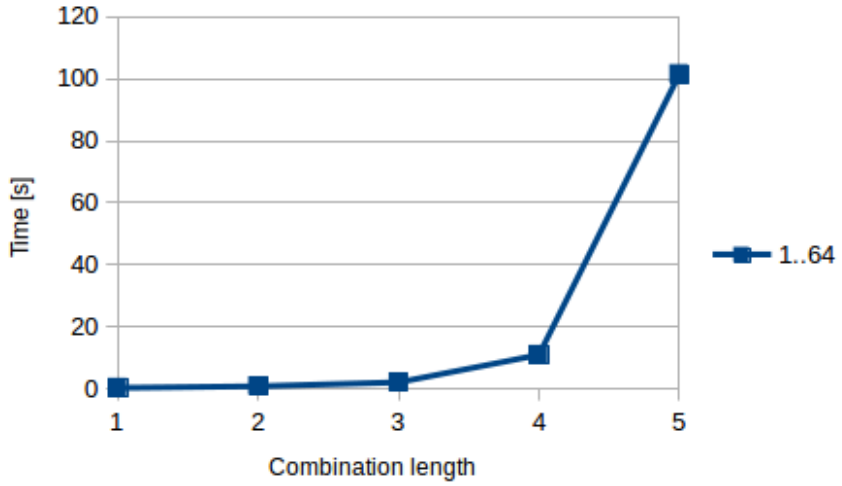

Figure 5.7. Search time rapidly increases even while searching in very small range of iteration counts (from 1 to 64).

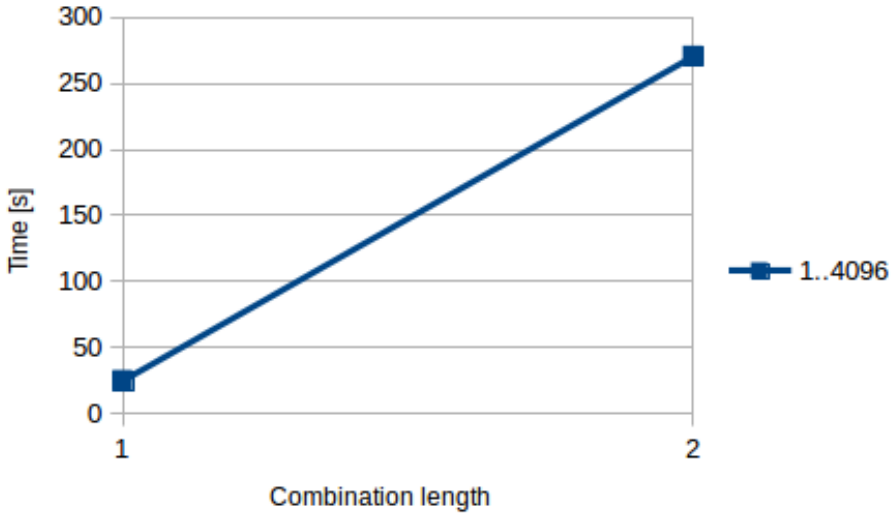

Figure 5.8.

When searching for iteration count in a broad range of values (1..4096), computation time increases dramatically. Looking at the chart, I would argue that it would take hours to recover passwords of length 3 and possibly dozens of hours to get passwords of length 4.

6 Conclusions

After conducting a number of experiments described above, I would argue that three factors dramatically increase computation time needed to recover passwords. First is obviously the length of the password, which as shown in charts 5.4 and 5.5, among others, makes recovering the password immensely more difficult. The other important factor is length of the list of characters that are used to compose the password. Those made up of lower and uppercase letters as well as numbers tend to extend search time greatly. The last significant way to increase time needed for successful brute-force attack is hiding the iteration count. As shown in charts 5.7 and 5.8, it lenghtens the whole process dramatically.