VILNIUS UNIVERSITY
FACULTY OF MATHEMATICS AND INFORMATICS

Baigiamasis bakalauro darbas

Acceleration of the automated password guessing method

Automatizuoto slaptažodžių parinkimo metodo paspartinimas

Done by:
Iveta Lenčiauskaitė

Supervisor:
prof. dr. Igoris Belovas

Scientific advisor:
Andrius Chaževskas

Vilnius
2024

Contents

Abstract

Investigating protected information sources is necessary during criminal investigations in the modern era of technology. These inquiries are conducted in Lithuania by the Forensic Expertise Centre of Lithuania's (FSCL) Digital Information Expertise Department. Data encryption and password security present a significant challenge when attempting to access the information. Numerous artificial intelligence algorithms have been created to access that by employing various password-guessing strategies.

The time required to crack a password grows exponentially with its complexity. The present study aims to improve the efficiency of one of the newest algorithms. Various techniques will be examined, and the most efficient will be selected for practical implementation.

Keywords: password guessing, code optimization, GPU optimization, CPU optimization, Python performance enhancement.

Santrauka

Acceleration of the automated password guessing method

Šiuolaikinėje technologijų eroje atliekant kriminalinius tyrimus būtina tirti įvairius informacijos šaltinius, tarp jų ir įvairias skaitmeninės informacijos laikmenas. Tokius tyrimus Lietuvoje atlieka Lietuvos teismo ekspertizės centro (LTEC) Skaitmeninės informacijos ekspertizės skyrius. Duomenų šifravimas ir slaptažodžio sudėtingumas daro įtaką kokiu greičiu ir kokiais būdais informacija gali būti išgauta. Šiems atvejams egzistuoja dirbtinio intelekto algoritmai, kurių dėka yra atliekamas slaptažodžių spėjimas naudojant žodynus ir nutekintas slaptažodžių duomenų bazes.

Priklausomai nuo slaptažodžio sudėtingumo, eksponentiškai auga ir laikas bei resursai, reikalingi slaptažodžiui atspėti. Šio darbo tikslas – pagerinti vieno iš naujausių algoritmų efektyvumą. Šiame darbe bus išnagrinėtos įvairios kodo optimizavimo technikos. Viena iš jų bus įgyvendinta praktiškai ir aprašyta, kaip ji buvo implementuota.

Raktiniai žodžiai: slaptažodžio parinkimas, kodo optimizavimas, GPU optimizavimas, CPU optimizavimas, Python kodo spartinimas.

1 Introduction

The growth of information technologies is inextricably linked to the existence of modern society. These technologies have an impact on numerous activities and aspects of life; as a result, we observe improvement and modernization in many sectors. Regardless of how many benefits information technology development delivers, it can also be utilized for various illegal actions. As a result, investigations of various digital information carriers (such as mobile phones, memory cards, USB memory sticks, hard drives, and so on) are frequently done alongside criminal investigations.

In Lithuania, these investigations are carried out by the Digital Information Expertise Department of the Forensic Expertise Center of Lithuania (FSCL). FSCL is a state institution of public administration, the purpose of which is to carry out expert examinations in accordance with the tasks assigned by the courts and pre-trial investigation institutions. These studies have been conducted at FSCL since 1995. We can infer that the number of digital information examinations is always rising due to the quick development of new technologies based on data from the FSCL's Digital Information Examination Department (see Figure 1). As a result, over the past three years, more IT research has been finished, more data has been analyzed, and questions have been resolved by forensic IT examiners [4].

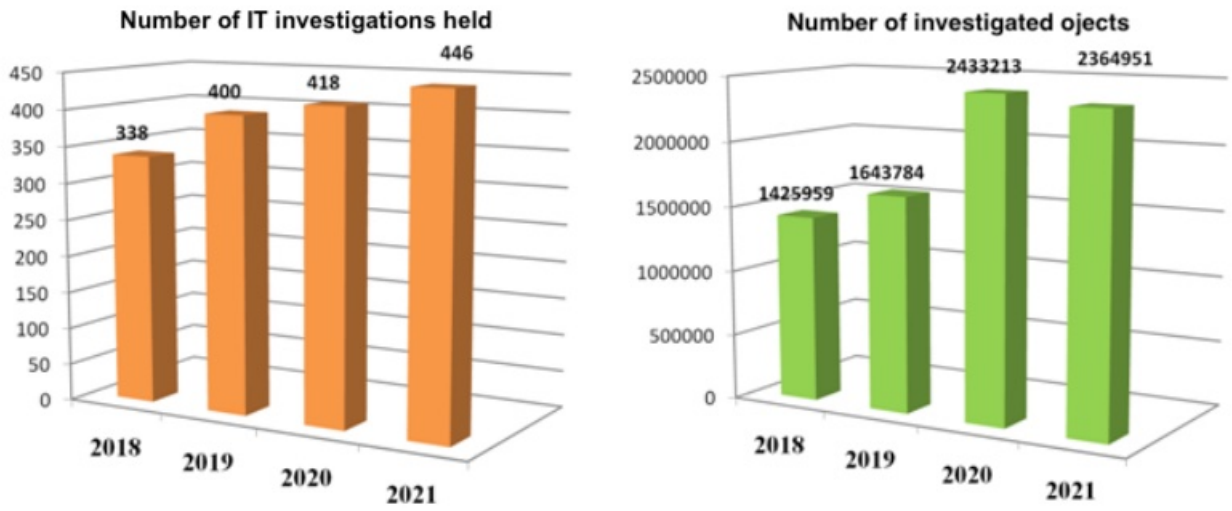


Figure 1. Digital Information Examination Department statistics [1].

The digital data retrieved from media and submitted for research is frequently password- and/or encryption-protected. Because of that, not every piece of digital evidence can be analyzed by forensic IT specialists. The issue of digitally encrypted data directly impacts the schedule and caliber of IT examinations. The digital information belonging to the suspected user cannot be decrypted without the correct password. Hence, no IT examination of that data will be carried out. In order to proceed with an investigation, forensic experts must use password-guessing attacks. A password-guessing attack is an attack in which possible passwords are tried in an attempt to guess the correct password and get access to the desired data.

Brute-force and dictionary attacks are the most widely used password-guessing methods. A brute force attack involves trying out every possible password using a predetermined collection of characters until the right one is found. The backbone of a dictionary attack is trying every word in

a prearranged dictionary. The size of a set of all potential password candidates, which increases exponentially with password length, is the most significant drawback of the brute-force attack.

Examiners use the following formula to assess the brute force attack's feasibility,

$$X = \frac{A^M}{N}.$$

Here A represents the size of the alphabet used, M - the length of the password, and N - the number of password guess attempts per second. From the equation above and the possible number of password combinations shown in the table below, it is evident that the speed of the password-guessing process largely depends on the hardware resources available.

Password length	Character set	Number of passwords
1-8	Numbers	111111110
1-8	Numbers, lowercase ASCII letters	2.90×10^{12}
1-8	Numbers, lowercase and uppercase ASCII letters	2.22×10^{14}
1-8	Numbers, lowercase and uppercase letters, special ASCII symbols	6.16×10^{15}
1-10	Numbers	11111111110
1-10	Numbers, lowercase ASCII letters	3.76×10^{15}
1-10	Numbers, lowercase and uppercase ASCII letters	8.53×10^{17}
1-10	Numbers, lowercase and uppercase letters, special ASCII symbols	1.75×10^{19}

Figure 2. Possible number of passwords depending on length and character set [4].

1.1 Statement of the problem

The code that we are working with is a script written in Python programming language. This script utilizes a long short-term memory (LSTM) network, which is a special type recurrent neural network (RNN), for character-level text generation. In our case, the script generates passwords. Although, there is an issue: the original password generation script works quite slowly, and generating passwords of more than ten characters takes weeks. We aim to improve Python's performance in this given script to be able to guess passwords in a reasonable time.

1.2 Aims and objectives

This work aims to compare and review existing techniques while investigating prospective approaches to improve and accelerate the most recent password-guessing algorithm, making password generation more effective. This is the list of the objectives:

1. To describe the password-guessing algorithm under investigation.
2. To review existing acceleration techniques relevant for Python programming language.
3. To determine which approach fits best.
4. To implement the most efficient method.

1.3 Structure of the work

This work is organized as follows: In section 2 we survey the related research and literature. In section 3 we present the algorithm outline, flowchart, and explain its functionality in detail. In section 4 we present the improvement concept, including thoughts on methods that were not chosen. Section 5 contains performed experiment data, such as different code runtime before and after the implemented improvement. We also include our results in section 5. In the last section of this paper we discuss future work that this paper has given a start.

2 State-of-art

In this section we will survey recent research literature in order to highlight available approaches for program acceleration.

2.1 Python accelerators

Python accelerator review is necessary and relevant here because the password-guessing algorithm under investigation is written in Python.

Stackoverflow's annual Developer survey indicates that Python is one of the most popular programming languages among developers worldwide [18]. At least since 2020, it has occupied a high position. Python has gained popularity in the scientific and engineering computing communities due to its open-source nature, simple syntax, and the abundance of scientific and mathematical packages that have grown to be available for it. Marowka indicates the following solutions to speed up Python's performance [8].

2.1.1 NumPy

The NumPy project [12] started in the mid-90s as a collaborative effort of an international team of volunteers aimed to develop a data structure for efficient array computation in Python. Today, NumPy is an open-source, one of the core extension modules for Python scientific computing.

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more. It offers quick and pre-compiled functions for mathematical and numerical operations.

NumPy provides a powerful multidimensional array object with advanced and efficient general-purpose array operations. It contains three sub-libraries with numerical routines providing basic linear algebra operations, basic Fourier transforms, and sophisticated capabilities for random number generation. It also provides facilities to support interoperability with C, C++, and Fortran. Besides its obvious scientific applications, NumPy can also be used as an efficient multidimensional container of generic data. New structured data types with fixed storage layouts can be defined by combining fundamental data types like integers and floats. This allows NumPy to seamlessly and speedily integrate with various database formats. NumPy speeds up calculations by having two strong features - broadcasting and vectorization. To be more precise, vectorization makes it possible to perform arithmetic operations on an array without writing any *for* loops.

2.1.2 SciPy

SciPy, standing for Scientific Python, is a free, open-source library for scientific and technical computing. SciPy contains modules for optimization, linear algebra, integration, interpolation, special functions, FFT, signal and image processing, ODE solvers, and other tasks common in science and engineering [15]. This library extends the capabilities provided by NumPy.

2.1.3 PyPy

PyPy is a Just-in-Time compiler and Python interpreter [14] that uses RPython, an alternative interpreter language, in place of the standard CPython interpreter, to provide a faster, more effective, and more compatible way to implement the Python language. Additionally, Python code can be dynamically compiled into machine code using PyPy's Just-in-Time compiler.

2.1.4 Numexpr

Numexpr is a module that aims to accelerate the evaluation of a numerical expression operation on NumPy arrays while using less memory footprint than pure Python. It makes use of multi-core architectures and an effective multithreading mechanism to accelerate computations. These capabilities are reached by avoiding memory allocation of temporary results that improve cache utilization, using an integrated virtual machine that parses expressions into its own op-codes, and splitting array operands into chunks that fit in cache memory. Below is a pseudo-code illustration of how speedup is achieved by using Numexpr [11].

```
1 In [1]: import numpy as np
2 In [2]: import numexpr as ne
3 In [3]: a = np.random.rand(1e6)
4 In [4]: b = np.random.rand(1e6)
5 In [5]: timeit 2*a + 3*b
6 10 loops, best of 3: 18.9 ms per loop
7 In [6]: timeit ne.evaluate("2*a + 3*b")
8 100 loops, best of 3: 5.83 ms per loop   # 3.2x: medium speed-up (simple
      expr)
9 In [7]: timeit 2*a + b**10
10 10 loops, best of 3: 158 ms per loop
11 In [8]: timeit ne.evaluate("2*a + b**10")
12 100 loops, best of 3: 7.59 ms per loop   # 20x: large speed-up due to
     optimised pow()
```

2.2 GPU optimization

In [19], Träff et al. present three implementations to speed up the code. Two of those methods optimize the GPU. It is shown that both GPU-accelerated codes can solve large-scale topology optimization problems with 65.5 million elements in approximately 2 hours using a single GPU, while the reference implementation takes about 3 hours and 10 minutes using 48 CPU cores. Authors highlight the advantage of GPU computing - modern GPUs offer an excellent computing power-to-price ratio compared to the more general-purpose central processing units (CPUs).

The authors' results prove that GPU programming can attain considerable performance improvements over the CPU. That is achieved by exploiting a large amount of parallelism in graphics processing - GPU can be thought of as a stream processor, which applies some function, a so-called kernel, to a large set of inputs. However, it is crucial that implementations for the GPU follow this structure. Most frameworks for general-purpose GPU programming, for example, CUDA or OpenCL, are based on the notion of kernels in order to ensure that the resulting programs are structured correctly. While these frameworks can produce very efficient code for the GPU, it is challenging to port CPU programs to GPUs, as the reformulation of the problem to kernels is left to the programmer, which is one of the biggest challenges in optimizing the GPU.

In [7], Ge et al. proposed an improved GPU-based brute force password recovery method for SHA512 by employing multiple optimization techniques. They used C to write GPU-running programs and OpenCL to make the most of the GPU to utilize it fully. The authors created a simple GPU system and improved it using several optimization techniques, including password concatenation, register reuse, faster instructions, and a meet-in-the-middle strategy. In the first step of their work, they include a basic password recovery scheme for SHA-512. In this scheme, the CPU generates passwords in batches, while GPUs are utilized to perform concurrent SHA-512 hash calculations in parallel. The basic scheme uses asynchronous parallel to improve CPU and GPU occupancies. In a sequential workflow, the CPU waits for results while the GPU processes the current batch. With asynchronous parallel, the CPU could generate the next batch of passwords while the GPU is calculating. In the second and third sections, they do some code optimization, and after all of their testing and research, the authors achieved a speed of 1055 M hashes per second on two AMD R9 290 GPUs. Their solution was 11% quicker when compared to *Hashcat*, the fastest password-cracking program.

2.3 Numba

Numba [10] is a just-in-time (JIT) compiler for a subset of the Python language, allowing numeric Python code to be compiled down to fast machine code. Using this handy Python package that supports CUDA programming of GPU, Dogaru and Dogaru [6] demonstrate that substantial speedups (100-250 times) with respect to the CPU implementation can be achieved by maximizing the load (number of threads on each computational GPU core) and by properly allocating the GPU global memory. In [5], Crist provides an example of code, which is then run by Numba, and suggest that it is possible to optimize existing Python code incrementally rather than requiring a full rewrite. He shows that this package, with its simple decorator API can compile Python functions to native code, often matching equivalent code written in C.

2.4 Parallel programming

Many different approaches have been developed for parallel programming with Python on shared and distributed memory environments. A comprehensive list of parallel Python libraries and software is available on Python's documentation site, parallel processing section [13]. We will discuss some of the approaches used in recent studies.

2.4.1 pPython

In [3], Byun et al. proposed pPython for Parallel Python Programming and achieved good speed up without sacrificing the ease of programming in Python by implementing partitioned global array semantics (PGAS) with porting pMatlab, MatlabMPI, and gridMatlab in Python. pPython is mentioned to achieve high performance because it adopts a coding style that uses some fragmented PGAS constructs - the style is less elegant but provides strict guarantees on performance. More specifically, distributed arrays are used as little as possible and only when inter-processor communication is required.

2.4.2 Multiprocessing package

Python's standard library comes equipped with several built-in packages for developers to use and take advantage of. One such package is the multiprocessing package, that supports spawning processes using an API similar to the threading module. In other words, applications can be broken into smaller processes that can run in parallel. The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads. Due to this, the multiprocessing module allows to fully leverage multiple processors on a given machine. [9]In [2], Abeykoon et al. manage to speed up the X-ray Photon Correlation Spectroscopy Software Tool by using the multiprocessing package. Below is an example of how multiprocessing is implemented into code.

```
1 from multiprocessing import Process
2 # Create a new process
3 process = Process()
4 # Create a new process with a specified function to execute.
5 def example_function():
6   pass
7 new_process = Process(target=example_function)
```

2.4.3 Multithreading

Threads are lightweight processes, which can also be defined as process instances. In multithreading, the GIL (Global Interpreter Lock) prevents the threads from running simultaneously. Multithreading can give an illusion of parallelism because of high-speed switching by the CPU. Actually, they implement concurrency - which means several tasks running at the same period. Therefore, using the Python threading module to parallelize programs is an ineffective strategy for increasing the performance of CPU-bound tasks. However, in [16], multithreading is listed as an excellent option for I/O operations. Multithreading is chosen as the initial technique to speed up the runtime of their code by Sodian et al. in [17]. The authors introduced a function to their code designed to store the outputs into a shared list. Within the code, individual threads were instantiated for each pair of coordinates and then started. The program awaited the completion of all threads before proceeding. In the result section, multithreading is mentioned to have achieved around 7x speedup in the program's runtime compared to the original code.

3 Automated password guessing: outline of the algorithm

In this section, we present the structure of the password generation algorithm and outline the origin of the main problem that we are trying to solve in this paper. We also present a flowchart to illustrate better how the script works.

3.1 Algorithm outline

The script starts by importing necessary libraries such as TensorFlow, NumPy, and Pandas. After that, it loads the vocabulary by reading the vocabulary file ('Vocab.txt' in our case), which contains a list of characters used for generating passwords. This file serves as the vocabulary for the characters that the model understands. With the vocabulary read, the script builds a neural network using TensorFlow's Keras API. The model consists of:

- An Embedding layer that converts characters into numerical vectors.
- Two LSTM layers responsible for learning patterns and generating sequences.
- A Dense layer that predicts the probability distribution of the next character.
- Pre-trained weights, which are loaded from a checkpoint file to initialize the model.

Next, the script loads pre-trained weights for the neural network model from a checkpoint file. After performing these initial steps, the script starts the password generation process. Password generation starts with an initial character ("<bos>" - beginning of sequence) and converts it into a numerical representation using the vocabulary. Using this initial character representation, the model predicts the probabilities for the next characters. The script calculates the probabilities of the predicted characters. It applies a threshold (THRESHOLD variable in our case, which is equal to 10^{-10} in the original code), which is the decision-making point to filter characters. Only the characters with probabilities higher than the threshold are kept and saved in the iteration results. The code then goes through a series of iterations - it is a for loop that goes as many times as the value of the PASSMAX variable. So, for example, if PASSMAX = 10, we can define until which length our passwords should be generated. Below is the basic shell of our *for* loop to generate passwords, showing that PASSMAX variable defines how many iterations our code will run for.

```
1 for i in range (PASSMAX):
2  {
3  #rest of password generation
4  }
```

The code generates new sequences based on the previously generated text - here, we have a recursive loop. With each iteration, we get results that are one symbol longer than the previous iteration generated characters. Each iteration results - generated characters and possibilities - are written into separate text files, 'Chars_i.txt' and 'Pred_i.txt', where *i* marks the iteration number. We also utilize 'Passwords.txt' to save all potential passwords, and generated characters are considered passwords if they have the <eos> (end of sequence) symbol.

In order to conduct tests on the amount of time a script runs, we included time tracking both inside the *for* loop and for each iteration to determine precisely where the script spends its most time.

The variables that can be altered to generate less or more results are THRESHOLD and PASSMAX. In our case, in the original code, the threshold is 10^{-10}. PASSMAX is the variable given to the for loop to determine how long the passwords should be. In the original code, PASSMAX is equal to 22.

However, in the experiments that we will write about later in this paper, we will be using wider thresholds, such as 10^{-6} or 10^{-7}. This way, we will have more time to perform experiments, but even if we choose to work with shorter passwords and, at the same time, smaller arrays of data, we will still notice an improvement if we manage to achieve any. The original code version with 10^{-10} threshold and the PASSMAX variable equal to 22 runs about 30 days. This execution time is unacceptable, and here originates the aim of our work - to make the script more efficient.

3.2 Script flowchart

The flowchart below presents the general idea of how the script works. It is also important to highlight that the main *for* loop is recursive.

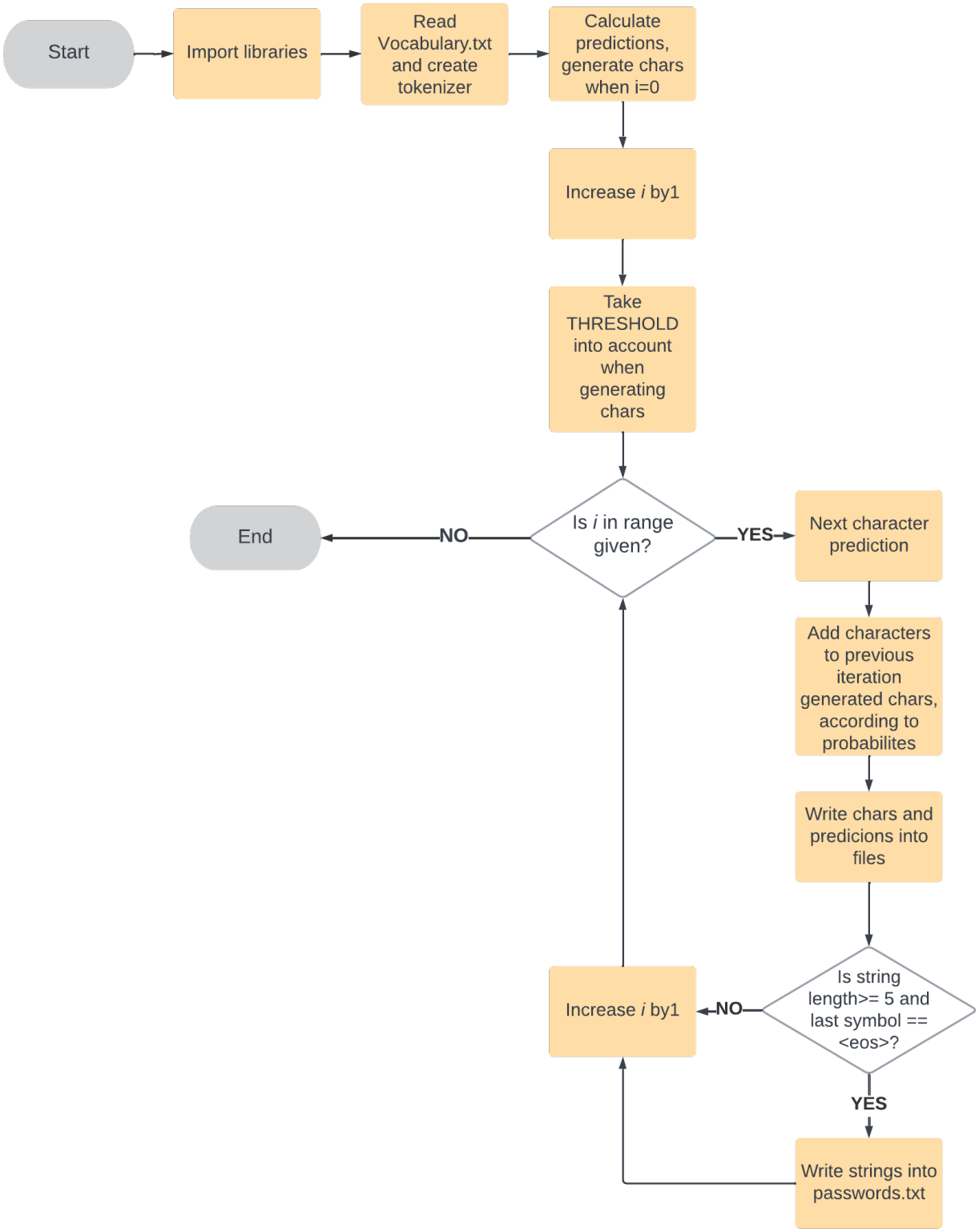


Figure 3. Flowchart of the script

4 Acceleration concept

In this section, we discuss how the implementation method was chosen. Also, we introduce an improvement of the code that was not part of our initial plan, however made a significant difference in script's performance.

4.1 Acceleration

Following the presented literature survey and practice-proven code acceleration examples, several options have been considered. Code parallelization was one of them. Following the examples, there was an idea and an attempt to separate the code into multiple processes and use the Python multiprocessing package to run multiple parts of the code simultaneously and speed up execution time.

However, after examining the script, we have determined that the bottleneck to implementing parallelism is the recursive loop, as seen in Figure 3. When the loop starts generating chars according to probabilities using i, it uses the previous iteration generated characters and adds another symbol to those already generated characters. Because of the nature of this type of loop, parallelizing the script becomes complicated. Most examples of parallelizing code suggest running certain loops/processes simultaneously - but that only works if they do not depend on each other's results. If there is a dependency of results, then we cannot separate loop iterations and make them run in parallel - they can only run sequentially. In our case, parallelization might still be an option for future work since there might be a way to separate parts of the loop, but this approach was not chosen to be implemented in this work.

As reviewed in Section 2.1, numerous libraries could be used to speed up calculations. NumPy is one of them, and it was already implemented in the code. In section 2.1.2 we reviewed SciPy, but in our case, the functionalities of this library were irrelevant, so it wouldn't have sped up the generation of passwords.

The first thing we implemented to start our experiments was adding lines to the code to track time. Doing this allowed us to collect the first data samples - execution times (without any code improvements). At first, only different iterations were timed, but later, we added time tracking inside the loop. With this implemented, we could track how long separate parts of the loop run. This was done because we discovered that it wouldn't be possible to parallelize the code by breaking it into processes since each iteration depends on results calculated in the previous iteration.

By implementing time tracking inside the loop, we could pinpoint one particular calculation that took much longer than the others. These are the lines of the code that caught our attention:

```
1   sample_tensor = tf.repeat(sample_tensor, BATCH_SIZE, axis=0)
2     pred = model(sample_tensor)
3
4     pred = pred[0].numpy()
```

In our script, BATCH_SIZE is equal to 64. In these particular code lines above, we ask the algorithm to prepare the pred variable, which has 64 elements (as many as the BATCH_SIZE variable is equal to). Still, later in the code, we only use the first element - pred[0], and all the other repeated 63 filler elements of the array pred are never used. That is a redundant initial calculation, and because of it, we lose a lot of time. After noticing this issue, it became apparent that rewriting this part should enhance the speed of our code execution time.

After noticing this, we worked on changing how the batch is handled in the code. We wanted to use the whole batch instead of filling the batch with repeated values. The way that the modified code works is that if we need to guess 1000 passwords, instead of creating 1000 batches of the same element, we create 15 batches of 64 elements and fill in the last batch with filler values just to keep it the same size as the previous batches. We do lose some data because, in the last batch, only the first 40 elements were used, but this is a very small loss compared to how many values were calculated and not used in the previous code version.

4.2 Bug fix

While implementing the idea described in 4.1, we noticed that the original code has a bug in how the vocabulary is interpreted. The script still generated passwords, so it was an unnoticeable bug because we did not get any errors in the console, and the results of the code looked like the results that we would expect.

The bug was found in how the loop interpreted start and end symbols. Our tokenizer was implemented correctly to interpret <pad>, <eos> and <bos> as one symbol; however, in the second loop, tokenizing of a string was written this way:

```
1 SecondSample_vector = [tokenizer[s] for s in SecondSample]
```

In this case, SecondSample is just a string that was written into a file and read as, for example, "aa" or "<pad>a". This tokenization method goes symbol by symbol, and string "aa" will be separated into ["a", "a"], and then "<pad>a" will be understood as ["<", "p", "a", "d", ">", "a"]. Because of this mistake, instead of 2 symbol inputs, we received 6, which interferes with batch creation because the batch has to have the input of the same dimension. With the example mentioned, if the code encounters the symbol <pad>, its dimension will be larger than the other strings.

To cope with this, we decided to alter the vocabulary, and instead of using <pad>, <bos>, and <eos> in our vocabulary, we changed them with the first letters of the Greek alphabet, α, β, and γ.

Because of this, we will have three different code versions to compare - the original code version with old vocabulary, the new code version with updated vocabulary, and the old code that takes in the updated vocabulary.

5 Experiments

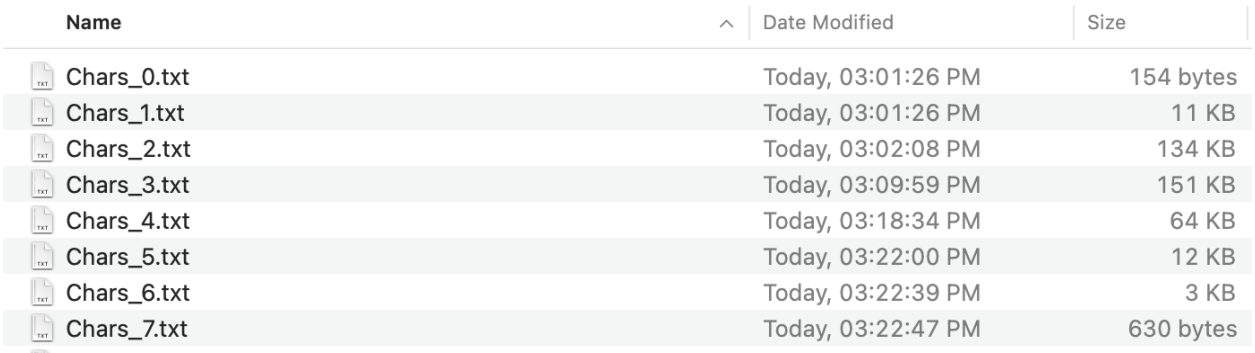
We have chosen the approach to use code execution time as the identifier of how efficiently our code works. We timed different versions of the script and saw how long it took to generate passwords with the old version of the code as well as with the modified version, where we implemented both the vocabulary and redundant calculation modifications.

We chose to work with wider thresholds, because they take less time to generate passwords compared to the original threshold. We will also use the ability to modify variable PASSMAX (introduced in Section 3.1), which is used in the main *for* loop and defines how many iterations our script will run for.

5.1 Experiment 1

The experiments described in this section are performed on an Apple M1 chip-equipped computer with 16GB RAM, running on macOS 13.6.3.

First, we timed the original script and used the original vocabulary file. The table below shows the results of using 10^{-6} threshold, and PASSMAX equal to 10. With these parameters, we get 16 file outputs, consisting of 8 text files with generated characters and eight text files with predictions of each iteration.

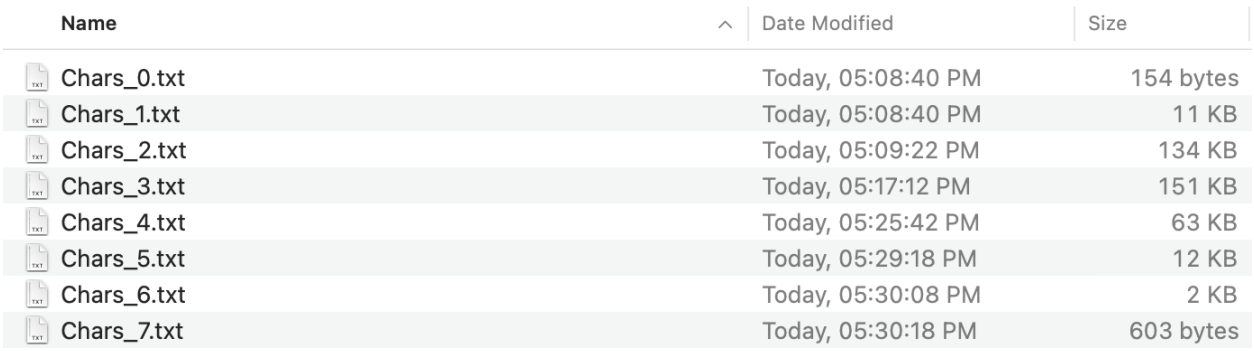


Name	Date Modified	Size
Chars_0.txt	Today, 03:01:26 PM	154 bytes
Chars_1.txt	Today, 03:01:26 PM	11 KB
Chars_2.txt	Today, 03:02:08 PM	134 KB
Chars_3.txt	Today, 03:09:59 PM	151 KB
Chars_4.txt	Today, 03:18:34 PM	64 KB
Chars_5.txt	Today, 03:22:00 PM	12 KB
Chars_6.txt	Today, 03:22:39 PM	3 KB
Chars_7.txt	Today, 03:22:47 PM	630 bytes

Figure 4. Results of original code, obtained with PASSMAX=10 and THRESHOLD=10^{-6}.

As we can see from Figure 4, the script ran for about 21 minutes, since after 21 minutes we generated the last character file.

Next, below we present results of running the same script, using the modified vocabulary for generation. Vocabulary modification reasons and implementation is discussed in section 4.2. The only changes that this version of the script had were the addition of encoding="utf-8" whenever reading or writing into a file, as well as replacing <bos>, <eos>, and <pad> characters in the code with the Greek letters that we used. No other adjustments were made to impact the code's functionality.

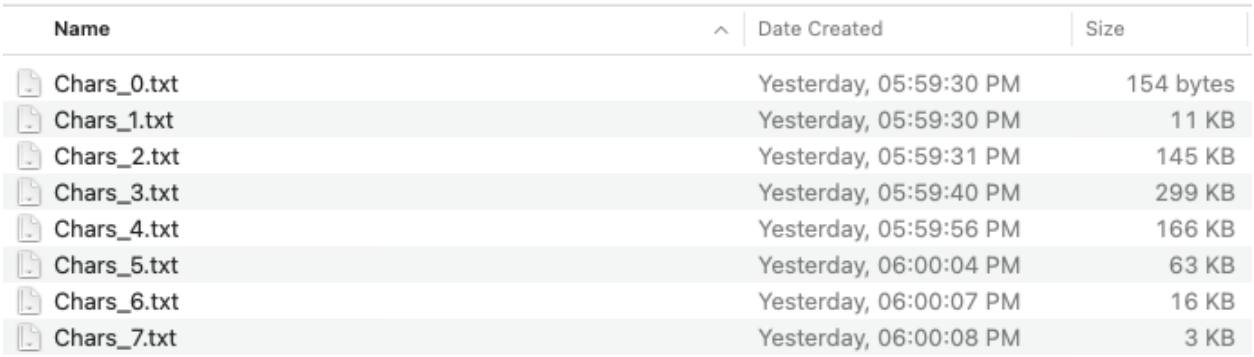


Name	Date Modified	Size
Chars_0.txt	Today, 05:08:40 PM	154 bytes
Chars_1.txt	Today, 05:08:40 PM	11 KB
Chars_2.txt	Today, 05:09:22 PM	134 KB
Chars_3.txt	Today, 05:17:12 PM	151 KB
Chars_4.txt	Today, 05:25:42 PM	63 KB
Chars_5.txt	Today, 05:29:18 PM	12 KB
Chars_6.txt	Today, 05:30:08 PM	2 KB
Chars_7.txt	Today, 05:30:18 PM	603 bytes

Figure 5. Results of original code with modified vocabulary, obtained using PASSMAX=10, THRESHOLD=10^{-6}.

As we can see from Figure 5, the script execution time is very similar, about 21 minutes. We do not see a huge difference in file sizes either. Comparing the output files 'Chars_7.txt' of the old code version with the version with an updated vocabulary, we see that the latter has generated three more passwords.

In Figure 6, we present results achieved using the modified code version with new vocabulary. As we can see, the last file was created in just 38 seconds after the code started running.



Name	Date Created	Size
Chars_0.txt	Yesterday, 05:59:30 PM	154 bytes
Chars_1.txt	Yesterday, 05:59:30 PM	11 KB
Chars_2.txt	Yesterday, 05:59:31 PM	145 KB
Chars_3.txt	Yesterday, 05:59:40 PM	299 KB
Chars_4.txt	Yesterday, 05:59:56 PM	166 KB
Chars_5.txt	Yesterday, 06:00:04 PM	63 KB
Chars_6.txt	Yesterday, 06:00:07 PM	16 KB
Chars_7.txt	Yesterday, 06:00:08 PM	3 KB

Figure 6. Results of modified code, obtained with PASSMAX=10 and THRESHOLD=10^{-6}.

With the data of each iteration runtime in seconds collected, we can put the numbers in one place to better see how execution times differ. Below, we present a table comparing the script execution time in the original version of the code as well as in the modified version of the code. By seeing execution times in seconds side by side, we can see how much faster the revised version of the code generates passwords.

File names (txt)	Original version	Modified version
Chars_0	<1	<1
Chars_1	<1	<1
Chars_2	42	1
Chars_3	471	9
Chars_4	515	16
Chars_5	206	8
Chars_6	39	3
Chars_7	12	1
Total runtime	1285	38

Table 1. Runtime of original code and modified code comparison by iteration (in seconds).

To measure the difference in execution time by percentage improvement, we will use the following formula,

$$\frac{t_{old} - t_{new}}{t_{old}} = 100\left(1 - \frac{t_{new}}{t_{old}}\right)\%,$$

here t_{old} stands for original code run time, and t_{new} is the execution time of the modified script. Using this formula, we can compare the original code execution time, 1285 seconds, with the modified code execution time, which is just 38 seconds. We see that the new version of the code executed 97% faster compared to the original version.

It is also worth mentioning that we noticed a difference in generated file sizes. After checking each of the files, it became clear that the modified code generated more unique passwords than the original version of the code.

For example, we compared the file sizes of file 'Chars_7.txt' generated using the original code and the 'Chars_7.txt' file generated by the modified code. In the experiment with the original code version, we see that this file weighs 630B. While looking at the results of the revised code output, we see that the generated file weighs 10KB. After checking the content of the files and comparing the generated passwords, we conclude that the new code version generated 349 unique passwords, while the old version generated only 70.

Considering the results, we conclude that the modified script version not only speeds up the execution time but also generates more output.

5.2 Experiment 2

The second experiment was performed on a machine equipped with Intel core i7-8750h processor, GPU GTX 1060 Mobile with 6GB VRAM. In this experiment, we decided to generate more passwords by using PASSMAX=22 and selected a less wide threshold with a value THRESHOLD=10^{-7}. This way, we expect increased execution time in the original code. By comparing it to the modified version of the code, we expect an even more drastic improvement in the runtime compared to the previous experiment.

Below, we present results generated by the old code version.

Name	Date Modified	Size
Chars_0.txt	Today, 11:09:50 AM	252 bytes
Chars_1.txt	Today, 11:09:52 AM	20 KB
Chars_2.txt	Today, 11:12:16 AM	463 KB
Chars_3.txt	Today, 12:01:48 PM	1,7 MB
Chars_4.txt	Today, 02:33:28 PM	1,3 MB
Chars_5.txt	Today, 04:14:10 PM	512 KB
Chars_6.txt	Today, 04:48:00 PM	115 KB
Chars_7.txt	Today, 04:54:52 PM	24 KB
Chars_8.txt	Today, 04:56:12 PM	6 KB
Chars_9.txt	Today, 04:56:30 PM	2 KB
Chars_10.txt	Today, 04:56:36 PM	1 KB
Chars_11.txt	Today, 04:56:40 PM	560 bytes
Chars_12.txt	Today, 04:56:42 PM	270 bytes
Chars_13.txt	Today, 04:56:42 PM	112 bytes
Chars_14.txt	Today, 04:56:42 PM	Zero bytes
Chars_15.txt	Today, 04:56:42 PM	Zero bytes

Figure 7. Results of original code, obtained with PASSMAX=22 and THRESHOLD=10^{-7}.

As we can see from Figure 7, the old version of the code ran for 5 hours and 46 minutes. After the disappointing results of the original code, we ran an experiment with the modified version of the code, using the same PASSMAX and THRESHOLD values. Below is a table with the generated files with timestamps.

Name	Date Modified	Size
Chars_0.txt	Today, 10:43:50 AM	252 bytes
Chars_1.txt	Today, 10:43:50 AM	21 KB
Chars_2.txt	Today, 10:43:54 AM	541 KB
Chars_3.txt	Today, 10:44:44 AM	2,6 MB
Chars_4.txt	Today, 10:48:36 AM	2,6 MB
Chars_5.txt	Today, 10:52:38 AM	1,3 MB
Chars_6.txt	Today, 10:54:02 AM	463 KB
Chars_7.txt	Today, 10:54:26 AM	118 KB
Chars_8.txt	Today, 10:54:32 AM	27 KB
Chars_9.txt	Today, 10:54:34 AM	6 KB
Chars_10.txt	Today, 10:54:34 AM	2 KB
Chars_11.txt	Today, 10:54:34 AM	882 bytes
Chars_12.txt	Today, 10:54:34 AM	330 bytes
Chars_13.txt	Today, 10:54:34 AM	192 bytes
Chars_14.txt	Today, 10:54:34 AM	Zero bytes
Chars_15.txt	Today, 10:54:34 AM	Zero bytes

Figure 8. Results of modified code obtained with PASSMAX=22 and THRESHOLD=10^{-7}.

The data provided shows that the new version of the code ran for around 11 minutes. Comparing this runtime with the original version's runtime, we see that the modified version ran 97% faster than the original. Below we present a table that puts the execution times side by side for a better comparison. We excluded 'Chars_14.txt' and 'Chars_15.txt' from the table because these files were empty.

File names (txt)	Original version	Modified version
Chars_0	<1	<1
Chars_1	2	<1
Chars_2	144	4
Chars_3	2976	50
Chars_4	9100	232
Chars_5	6042	242
Chars_6	2030	84
Chars_7	412	24
Chars_8	80	6
Chars_9	18	2
Chars_10	6	<1
Chars_11	4	<1
Chars_12	2	<1
Chars_13	<1	<1
Total runtime	20816	644

Table 2. Runtime of original code and modified code comparison by iteration (in seconds).

While comparing the execution times, we also notice that generated file sizes differ. It is apparent that starting from the 1st iteration, our modified version of the script generated larger files. That means that the modified code not only ran so much faster but also provided more passwords. Looking at 'Chars_7.txt' generated using the original code and 'Chars_7.txt' generated by the modified code, we notice that files weigh 24KB and 118KB, respectively. The old version of the script generated 2388 unique passwords, and the modified script generated 11779 passwords. That is almost five times more results generated.

Conclusions and discussion

The work aimed to accelerate the running time of the automated password-guessing algorithm, which would be the base of IT examinations carried out by forensic experts during criminal investigations. While solving this problem, we have completed the following objectives and came to the following conclusions:

1. We have analyzed the automated password guessing algorithm, described how it works, and presented a flowchart to illustrate the functionality and the key issue better - the main *for* loop being recursive. Studying the algorithm, we came to the **conclusion** that there is room for improvement and acceleration.

2. We have reviewed the most recent acceleration techniques (ranged from code modification to GPU and CPU optimization) relevant to Python programming language and came to **conclusion** that the list of prospective methods has to be narrowed down because of the nature of the code and the recursive loop it relies on. We have discovered this by analyzing the code more explicitly.

3. Analyzing the code, we found a bug preventing it from executing faster. The issue was related to how batches are created. This discovery led us to the idea that the fix in batch creation logic could be the most efficient way to accelerate the algorithm. We ran experiments on two different machines to see if the implemented fix leds us to the goal.

4. The outcomes of the experiments showed that we could speed up the code 32 times. The modified code performed 97% faster than the original version. Moreover, we have noticed that the modified version of the code generates more passwords than the original version of the script. Thus, the script was improved to run faster and generate more results (note that the original research plan did not envisage this development). Everything described above leads us to the **conclusion** that all of the goals were accomplished.

While investigating the possible methods of code enhancement, we discovered that it might be feasible to use parallelization methods by dividing the script into separate pieces and utilizing CPU or GPU parallelization. This research did not explore this much because another method to pursue was chosen due to its efficiency. However, this leaves us with a strong start for future study.

References

[1] 2021–2030 metų plėtros programos valdytojos Lietuvos Respublikos Teisingumo ministerijos teisingumo sistemos plėtros programos Pažangos priemonės nr. 13-001-08-01-04 „Didinti ekspertinių tyrimų atlikimo efektyvumą“. https://tm.lrv.lt/uploads/tm/documents/files/dokumentai/Pl%C4%97tros%20programos/PPP%20apraso%20pagrindimo%20forma_LTEC_projektas_2022%2007%2001%20viesinimas.pdf. Accessed: 2023-11-05.

[2] Sameera K. Abeykoon, Meifeng Lin, and Kerstin Kleese van Dam. Parallelizing X-ray Photon Correlation Spectroscopy Software Tools using Python Multiprocessing. In *2017 New York Scientific Data Summit (NYSDS)*. IEEE, 2017. https://doi.org/10.1109/NYSDS.2017.8085042.

[3] Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, Hayden Jananthan, Michael Jones, Kurt Keville, Anna Klein, Peter Michaleas, Lauren Milechin, Guillermo Morales, Julie Mullen, Andrew Prout, Albert Reuther, Antonio Rosa, Siddharth Samsi, Charles Yee, and Jeremy Kepner. pPython for Parallel Python Programming. In *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, September 2022. http://dx.doi.org/10.1109/HPEC55821.2022.9926365.

[4] Andrius Chaževskas, Igoris Belovas, and Virginijus Marcinkevičius. Forensic password examination in leaked user databases. *Criminalistics and forensic expertology: science, studies, practice. Bratislava, Slovak Republic September, 2021*, page 241–257, 2021. ISBN 9788080549053.

[5] James Crist. Dask & Numba: Simple libraries for optimizing scientific python code. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2342--2343, 2016. http://dx.doi.org/10.1109/BigData.2016.7840867.

[6] Radu Dogaru and Ioana Dogaru. Optimization of GPU and CPU acceleration for neural networks layers implemented in python. In *2017 5th International Symposium on Electrical and Electronics Engineering (ISEEE)*. IEEE, October 2017. http://dx.doi.org/10.1109/ISEEE.2017.8170680.

[7] Can Ge, Lingzhi Xu, Weidong Qiu, Zheng Huang, Jie Guo, Guozhen Liu, and Zheng Gong. Optimized Password Recovery for SHA-512 on GPU. In *22017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. IEEE, July 2017. http://dx.doi.org/10.1109/CSE-EUC.2017.226.

[8] Ami Marowka. Python accelerators for high-performance computing. *The Journal of Supercomputing*, 74(4):1449--1460, Apr 2018. https://doi.org/10.1007/s11227-017-2213-5.

[9] Multiprocessing — process-based parallelism. https://docs.python.org/3/library/multiprocessing.html. Accessed: 2023-12-05.

[10] Numba. https://numba.pydata.org/. Accessed: 2023-12-18.

[11] Numexpr. https://numexpr.readthedocs.io/en/latest/intro.html#expected-performance. Accessed: 2023-12-18.

[12] Numpy official website and documentation. https://numpy.org/. Accessed: 2023-12-18.

[13] Parallel Processing and Multiprocessing in Python. https://wiki.python.org/moin/ParallelProcessing. Accessed: 2023-12-03.

[14] Pypy. https://en.wikipedia.org/wiki/PyPy. Accessed: 2023-12-18.

[15] Scipy. https://en.wikipedia.org/wiki/SciPy. Accessed: 2023-12-18.

[16] Lazar Smiljković, Miloš Radonjić, Marko Mišić, and Milo Tomašević. Comparing python code parallelization techniques for spatial transcriptomics data. In *2023 31st Telecommunications Forum (TELFOR)*, pages 1--4, 2023. https://ieeexplore.ieee.org/document/10372599.

[17] Loigen Sodian, Jaden Peterson Wen, Leonard Davidson, and Pavel Loskot. Concurrency and parallelism in speeding up i/o and cpu-bound tasks in python 3.10. In *2022 2nd International Conference on Computer Science, Electronic Information Engineering and Intelligent Control Technology (CEI)*, pages 560--564, 2022. https://ieeexplore.ieee.org/document/9950068.

[18] Stackoverflow developer survey 2023. https://survey.stackoverflow.co/2023/#technology-most-popular-technologies. Accessed: 2023-12-03.

[19] Erik A. Träff, Anton Rydahl, Sven Karlsson, Ole Sigmund, and Niels Aage. Simple and efficient gpu accelerated topology optimisation: Codes and applications. *Computer Methods in Applied Mechanics and Engineering*, 410:116043, 2023. https://doi.org/10.1016/j.cma.2023.116043.