A Computational Model for Efficient Password Guessing

Calder White

August 2019

Abstract

The prodigious amount of combinations for characters within a fixed set and length presents a false sense of security in passwords. Due to human involvement in password choice a large portion of combinations are rarely ever seen in passwords. In this paper an algorithm is proposed and proven to dramatically improve the efficiency of a brute force strategy to password guessing by learning the patterns in human password choice and applying them to the order in which passwords are guessed.

1 Introduction

Password security is a multidimensional problem involving two core vectors of attack. The first is the ability to attempt a password. This could be attempting to break a hash or to attempt a sign in on various platforms. There are many ways to counteract these vectors of attack such as hash salting or sign in timeouts. However, the proposed model focuses on the second vector of attack, the base security of a password. The base security in the context of the proposed model is defined to be how many attempts are required to guess a password. In a practical sense this is what many services characterize as a "strong" password.

Passwords are often viewed as secure for their difficulty to be guessed. The purpose of a password is to prevent those without authorization from accessing a given service due to their inability to produce the password for entry. There is comfort in the idea that within n number of possible characters for a string of length l that there are n^l possible combinations. However, in practice this comfort is seen to be misplaced.

Humans are far from perfect machines and their compatibility with this type of cryptography is especially poor. In a perfect world each password chosen by any given person at any given time would be completely random. This would provide an unbiased distribution of passwords making it the hardest possible to guess any password. The reality is that humans must also remember the passwords they have chosen. Humans do not work like computers and have trouble remembering raw data without any patterns without deliberate memorization techniques. In fact, it has been well observed that there are practical limits to what the average human can remember which is around 7 consecutive characters [8]. In contrast, humans do have a talent for recalling and recognizing patterns. This is beneficial in many areas of life but also produces predictable behavior. A bias is introduced with an uneven distribution of passwords that follow certain patterns that humans can easily remember.

In addition to the patterns humans tend to put in their passwords, there is also a social aspect to password choice. Certain passwords may be viewed as "weak" by both password standards and people themselves. However, the presence of these "weak" passwords actually strengthen the passwords that are "strong". For example, if a password

contained 8 `a` characters in a row, most people would view it as insecure and not choose it. The same goes for many services that require a minimum password standard. Even so, a password that must have a number, a capital letter, and length of 8 could potentially be `aaaaaaA1`. This would be valid by the service's standards, but a human would still likely not choose it. Using this principle we can reduce the number of passwords to guess for a given length and password requirement greatly. This is because the amount of passwords that are viewed as "insecure" out of the total pool of possible passwords is immense.

Two assumptions are to be made if the proposed model is to be able to efficiently guess passwords.

1. Passwords must be remembered
2. The entity remembering a password is a human.

These are important, since there are password managers that violate the second condition. The issue with this is that there is still a required password to access the password manager. In this way the password manager has done nothing to increase the security of the user's passwords, but has rather placed a higher payoff on breaking the one password of a user's password. Additionally, a password is not a password if it is not remembered. Saving a password in plaintext anywhere opens a multitude of vectors for attack and because of this, the possibility of not remembering passwords has been excluded from this paper's observations.

2 Computational Model

Now that the phenomenon of patterns in passwords has been acknowledged, the following question is now how to collect said patterns. The chosen method was to use a trie-like tree of characters (Count Tree) to keep the count of each character after a given sequence of other characters. Figure 2.1 presents an example this trie structure. It has a block size of 4, as the block size is the maximum depth of the tree. The Count Tree aggregates the patterns found into a series of counts for each character. These patterns then emerge in the sorted order of the counts at each depth level of the tree and path of characters to get there.

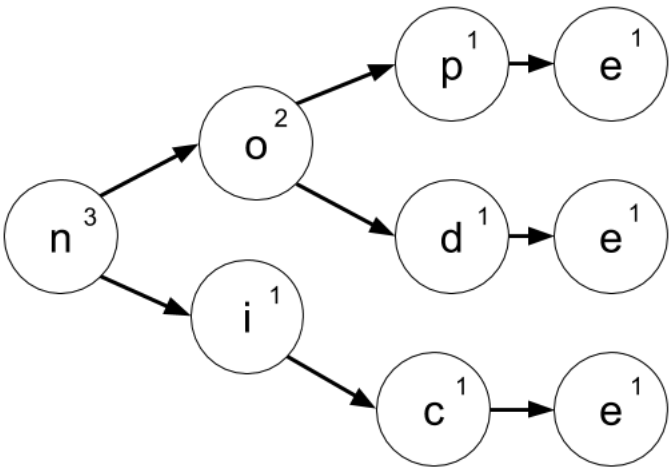

Figure 2.1: A CountTree after `node`, `nope` and `nice` are added to it. The counts of each character can be seen in the upper right corner of the circles.

Without regard to computational and memory limits the block size would be the length of the password that is to be guessed. However, there are practical limits to what can be done. In the model the indicies are dictated by the characters. So, if the first character in the character set is `a`, then all indicies will be calculated via `c - 'a'`. At each of these indicies there is a corresponding count for the character represented by a 32 bit integer. As the memory is a geometric series due to the tree structure, the amount of memory in bytes S required for any given character set size c and block size n can be calculated in closed form via the following equation.

$$S = 8 \cdot \left(\frac{c^{n+1} - 1}{c - 1} \right)$$

There are 95 type-able characters in ASCII so in order to include all possible passwords, a character set size of 95 is required. With this, the required amount of memory can be easily calculated. The resulting table in Figure 2.2 shows the practical limits of this approach to ingesting the patterns in human passwords. Due to the exponential growth of the memory requirements, the model jumps from being able to run on a laptop to requiring a server to requiring sharding among multiple servers all within 3 block sizes (4, 5, and 6 respectively).

Block Size	**RAM Required (MB)**
1	0.00077
2	0.07297
3	6.93197
4	658.53697
5	62,561.01197
6	5,943,296.137
7	564,613,133
8	53,638,247,636

Figure 2.2: The memory requirements for a character set size of 95.

It is clear that the block size cannot be the length of the password that is to be guessed. Even if it could be, this model would simply be a mediocre rainbow table of passwords with giant holes (counts of 0) where no passwords were added. This idea is a reminder that this model's advantage is making use of the patterns between a character's neighbours, not just memorizing common passwords. Alas a method for guessing passwords longer than the block size must be created. The same goes for training. Each password is on average going to be longer than the block size being used, so one password can in fact be used to create multiple blocks for training. This is done simply by moving the starting index over until the end of the string is reached, as demonstrated in Figure 2.3.

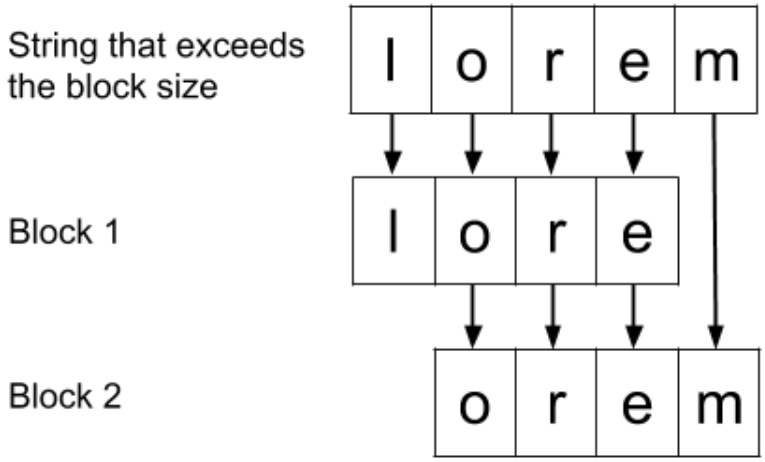

Figure 2.3: How a string longer than the block size is split up into blocks for training.

Once the patterns are collected in this manner, the next phase of execution is outputting the password attempts in order of most probable to least probable. This is done by sorting all of the characters by count at each depth level for all possible strings of characters that can come before them. This is because the model has a fixed character set so regardless if the count is 0 for a particular character, it is still in the Count Tree. The result is a tree of sorted character arrays with the counts now discarded. The lowest index has the highest count. This new sorted tree is called the Predictor.

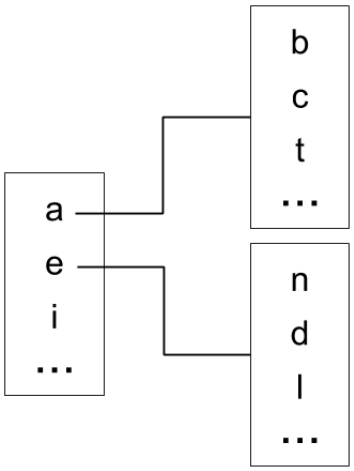

Figure 2.4: A truncated representation of the sorted character tree that makes up the Predictor.

The block sizes of the Predictor and the Count Tree are identical and this introduces the same problem for predicting a password longer than the block size. The solution is similar to that of training on passwords longer than the block size: rotate over the history. The beginning of the password can be generated without any extra effort simply by choosing the lowest indices at each depth. After the block size is reached, the next character is generated by inputting the last $blocksize - 1$ characters into the Predictor and then appending the outputted character to the generated password.

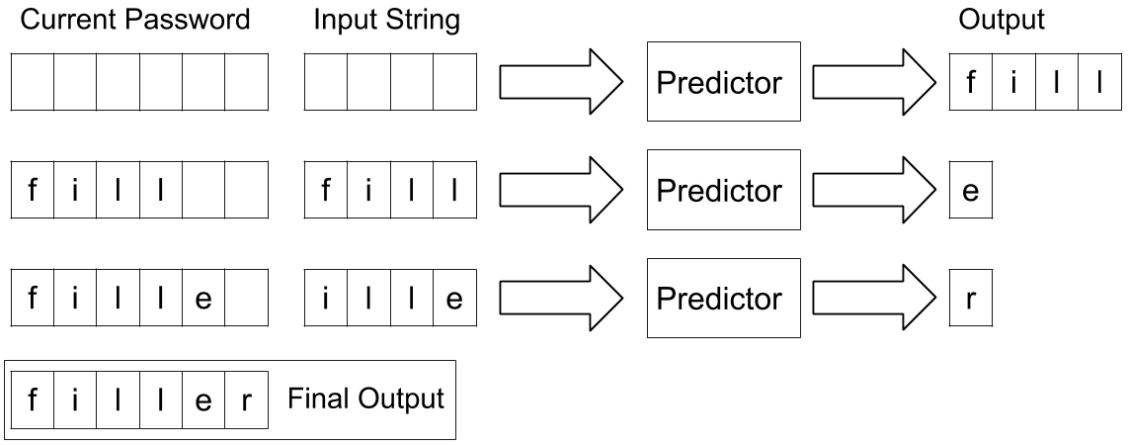

Figure 2.5: The steps involved in generating a password guess with a Predictor that has a smaller block size than the length of the password to be guessed.

With this passwords can now be efficiently guessed using a Predictor trained via the Count Tree and real data. An added benefit of this approach of indexing and brute forcing is that the time at which a password will be guessed can be deterministically calculated by simply finding the index of said password in the Predictor. This is done by finding the index of each character at each level and multiplying it by the size of the sub tree at the current depth level. This will prove extremely useful when attempting to analyze the effectiveness of the model.

3 Efficiency Analysis

The performance of the model detailed in this paper is pivotal on the data used to train it as well as the passwords it is predicting. Good data makes for good performance and unorthodox passwords make for bad efficiency ratings. This model captures the patterns that lie in the majority of passwords, and the data reflects that. The implementation for this model is in C++ for optimal speed and code organization. The data used to train the model is a plaintext password dump of roughly 60 Million human passwords from crackstation.net [5]. A character set size of 95 was used to encapsulate all readable characters in ASCII [3]. The block size was chosen to be four as it is the upper limit of block sizes which can be run on consumer grade hardware due to memory limits detailed in Section 2, Figure 2.2.

To analyze efficiency the trained model was used to calculate the overall index percentage of millions of passwords across various datasets. An index percentage is the overall index of a password divided by the total number of possible passwords for the length of the password. This is a representation of how good the model was at chosing when to guess a particular password. If the model works correctly, the index will be low for organic, human passwords and high for passwords that are unlikely. Note that the word index is used deliberately. Imagine all the possible passwords for a given length as a gargantuan array. The idea of this model is to move the more probable possibilities up in this array (decreasing the index) and move the lower probability combinations to the back.

A formula to calculate the percentage index of a password in a trained model can be expressed with an equation. With c as the character set size, n as the block size and a

representing the index at depth i for the current character in the password at index i.

$$\text{index percentage} = 100\% \cdot \left(\sum_{i=0}^{n} a_i \cdot c^{i+1} \right) \div c^n$$

The first test was done on the dataset the model was trained on. This should provide the most flattering results for the model, and as seen in Figure 3.1 the logarithmic shape of the line shows this to be true. A naive brute force approach is also graphed to provide a form of comparison. The naive approach is simply to loop through all possible characters in the order of ASCII [3]. To quickly note the potential shapes of these graphs, an optimal output is a logarithm and the worst is an exponential function. If there is a perfect distribution of passwords (maximum entropy across all passwords) then the naive approach should produce a linear function.

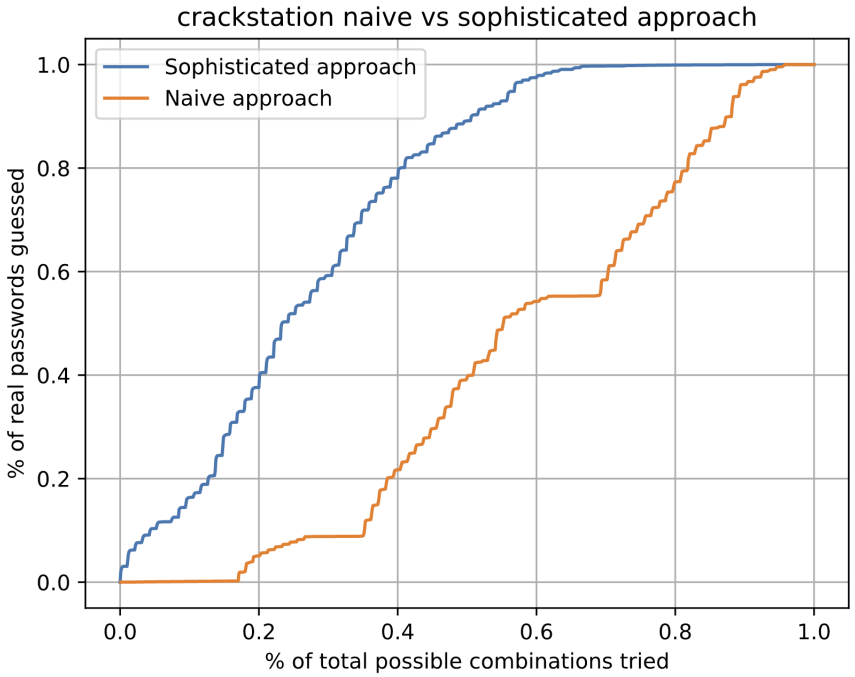

Figure 3.1: The amount of passwords guessed for total tried for crackstation's human only password dataset. Data source: *Crackstation's Password Cracking Dictionary* [5]

Notice that after 25% of the possible combinations are guessed, a whopping 50% of the passwords in the dataset are guessed correctly. Shortly after at 40% a whopping 80% of real passwords have been guessed. This is the exact behavior predicted in the introduction of senseless combinations such as `aaaaaaa` and such being guessed before practical combinations of characters. This reveals the enormous hole in human password choice due to their inherent security flaws. However, positive results are to be expected from the data set that the model was trained on. Next is a dataset that the model has never seen before.

The next dataset is a popular plaintext dump often referred to as "rockyou", after the company rockyou, which was hacked in December of 2019 [1]. The dataset consists of roughly 10 Million plaintext passwords. Shown by the naive approach, the dataset does present a bias in starting character of the passwords. This can be deduced from the plateau in the middle of the graph followed by high growth after it. This means that the dataset lacks passwords that start with a large amount of characters that are inside the 95 ASCII

readable characters. This contrasts the healthy distribution of the crackstation dataset. However, even through this bias the model still preforms to the same degree of efficiency as with the crackstation dataset it was trained on, as seen in Figure 3.2. This is a key strength of this algorithm. Despite the bias presented by a particular set of passwords, the model performance of the model shown by the shape of the line is very similar. At 30% of the total, 50% of real passwords are guessed and at 40% of the total, 80% of passwords are guessed.

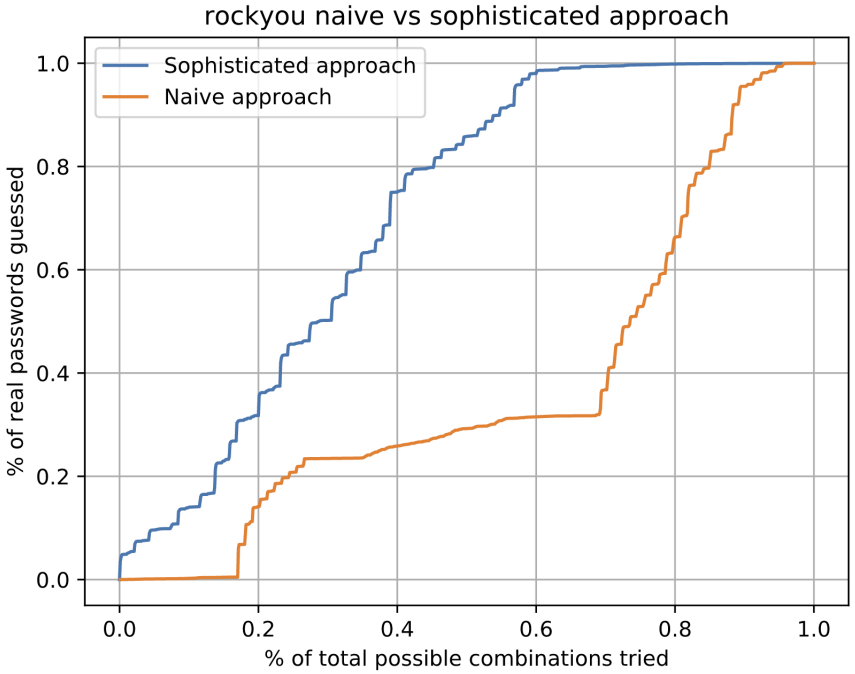

Figure 3.2: The performance of the model trained on crackstation on the rockyou dataset. Data source: *Rockyou Wordlist* [2]

4 Practical Applications

The efficiencies of this model allow for a particular application to brute forcing passwords. A normal brute forcing approach consists of attempting all possible passwords in an arbitrary order for a given length. With this comes an exponential curve of password possibilities as length increases, and the same goes for character set. This model enables a coefficient attached to the function of total passwords tried to allow the extension of feasible character sets and lengths of passwords to be guessed with this method. This is with the caveat that some small percentage of passwords will not be guessed within this time, as shown by Figure 3.1 and Figure 3.2.

Humans severely underestimate the power of computers, especially when put to work in parallel. "Experts" often talk about just how long it would take to crack a password of a large character set and length, but this is no longer the case! Gargantuan companies already possess incomprehensible amounts of compute resources beyond human imagination. Even more, companies like Google, Amazon, and Microsoft have opened up their compute to the public to be used in exchange for money. The question is no longer how long it will take to crack a password, but how much it will cost.

Now, in order to find the cost of a password by brute force, first the amount of attempts

per second must be calculated. This was accomplished via a hashing benchmark written in C++ calculating all the hashed for a fixed length string and checking if it was equal to an arbitrary string. The length was 4 and the character set size was 95. Then, the number of hashes per second was calculated via the following equation:

$$\text{hashes/second} = \frac{c^n}{\text{seconds elapsed}}$$

GCE was chosen due to the free initial $300 USD promotion, and it was found that on their compute intensive c2 instance 3.149×10^6 hashes per second are possible. A preemptible c2 instance costs $0.00822 per vCPU hour [7]. This is without any 1 month or 1 year commitment discounts, which are also worth considering. Regardless, with this the dollar per hash can be calculated, and it comes out to $7.25097914 \times 10^{-13}$ $/hash. Now, the price per password can be calculated by password length. The following calculations are under the assumption that the character set is as large as possible with all 95 readable ASCII characters.

Password Length	**Cost (USD)**
5	0.0056
6	0.5330
7	50.6363
8	4810.4477
9	456,992.5323
10	43,414,290.5700
11	4,124,357,604
12	391,813,972,373
13	37,222,327,375,438

Figure 4.1: The prices of cracking a password by brute force using GCE c2 instances for a character set size of 95. Decimal values are rounded to 4 places.

However, using this model these numbers can be greatly reduced due to the increase in probability of an early guess. Based on the data examined in Section 3, 80% of passwords will required 40% of the total cost shown in this table. What was once a $4,810 password will become a $1,924, 80% of the time. Also note that with particular discounts, the price of a c2 instance can be decrease significantly. For example, the 30 day sustained use discount of 30% of base fees [6]. With this in mind a new table can be created with these two multipliers attached to the price.

Password Length	**Cost (USD)**
5	0.00067
6	0.06396
7	6.07635
8	577.25373
9	54,839.10387
10	5,209,714.868
11	494,922,912.5
12	47,017,676,685
13	4,466,679,285,053

Figure 4.2: The Prices in Figure 4.1 with the 30% sustained use discount and %40 model advantage factored in.

It is important to note that these costs are only for public users. The question of private interests is even more relevant. Providers like Google are clearly making a profit on these instances [9], so for tech giant's interests, these costs are even lower. This transitions into the value of a password. What is the value of a password? Given a high enough value person, it can be quite exorbitant. In 2019 Jeff Bezos, the Founder and CEO of Amazon, was black mailed with sensitive information leading to his eventual divorce [4]. The agreement resulted in his ex wife taking $38 Billion in assets from their divorce [10]. This sum makes the price of a discounted and efficiently guessed 11 character password look like a rounding error. Additionally, there is value which is hard to quantify. Value such as gaining a competitive advantage in a tight business or controlling a public opinion with inside knowledge. To large corporations password cracking via this method is completely within question for malicious purposes.

5 Defenses

There are certain methods that can be used when choosing a password to avoid vulnerability to efficiency of this model. As the model takes advantage of the patterns between strings of characters the best defence is not using comprehensible strings of characters as your password. Although this may seem counter intuitive at first, it can be easier than it may seem. A password such as `hockey123` has a healthy length but lacks security in its abundant patterns. The word is English, opening up thousands of patterns that the model knows all about and the end of the password is a number – something common in many lazy passwords. Now, although this password is even more vulnerable to a dictionary attack, is serves as an example of a password abundant with predictable patterns. Even if the letters were to be reversed, it would still be vulnerable to this model due to the fact that the inter-character relationships are maintained. However, there are strategies that can be used to choose passwords that are safe to this vector of attack while still abiding by the requirements outlined in Section 1.

A strong method to generate a password that is strong against this vector of attack is the acronym method. Take any song, phrase or string of words and take a particular letter from each word. For ease of remembering, the first is usually easiest. This way the characters are not so heavily related to eachother in the password. A phrase as simple as "One small step for man, one giant leap for mankind" can be used to generate a strong password, which in this case is `ossfmoglfm`. Case can be varied and substitutions can be made to increase the character set, `Os$fmOglfM`. This method is powerful since it keeps length reasonable while maintaining a high degree of security to human based attacks. This is important, since it easy to create a strong password but hard to create a strong password that the average person is willing to type in thousands of times a year and remember without effort.

References

[1] N. Cubrilovic. "RockYou Hack: From Bad To Worse". In: (2009). URL: https://techcrunch.com/2009/12/14/rockyou-hack-security-myspace-facebook-passwords/ (visited on 09/15/2019).

[2] B. Dorsey. *Rockyou Wordlist*. 2019. URL: https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt (visited on 09/15/2019).

[3] Saul Gorn, Robert W Bemer, and Julien Green. "American standard code for information interchange". In: *Communications of the ACM* 6.8 (1963), pp. 422–426.

[4] A. Hartmans. "Jeff Bezos and MacKenzie Bezos have finalized their divorce agreement..." In: (2019). URL: https://www.businessinsider.com/jeff-bezos-mackenzie-bezos-divorce-finalized-2019-4 (visited on 09/15/2019).

[5] T. Hornby. *CrackStation's Password Cracking Dictionary*. 2019. URL: https://crackstation.net/crackstation-wordlist-password-cracking-dictionary.htm (visited on 09/15/2019).

[6] Google LLC. *Sustained Use Discounts*. 2019. URL: https://cloud.google.com/compute/docs/sustained-use-discounts (visited on 09/15/2019).

[7] Google LLC. *VM Instances Pricing*. 2019. URL: https://cloud.google.com/compute/vm-instance-pricing#c2_machine_types.

[8] George A Miller. "The magical number seven, plus or minus two: Some limits on our capacity for processing information." In: *Psychological review* 63.2 (1956), p. 81.

[9] R. Miller. "Google's Diane Greene says billion-dollar cloud revenue already puts them in elite company". In: (2018). URL: https://techcrunch.com/2018/02/01/googles-diane-greene-says-billion-dollar-cloud-revenue-already-puts-them-in-elite-company/ (visited on 09/15/2019).

[10] R. Neate. "Amazon's Jeff Bezos pays out $38bn in divorce settlement". In: (2019). URL: https://www.theguardian.com/technology/2019/jun/30/amazon-jeff-bezos-ex-wife-mackenzie-handed-38bn-in-divorce-settlement (visited on 09/15/2019).