New Approach in the Rainbow Tables Method for Human-Like Passwords

Mark A. Alpatskiy, Georgii I. Borzunov, Anna V. Epishkina, Konstantin G. Kogos
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
mark.alpatskiy@gmail.com, borzunov_g@mail.ru, avepishkina@mephi.ru, kgkogos@mephi.ru

***Abstract*—This paper represents a new approach to rainbow tables, a method of password recovery that was originally developed by Martin E. Hellman and then improved by P. Oechslin, so most of its implementations use Oechslin's modification. An improvement represented in this work mostly lies in the reduction function, which uses character statistics to generate more "human-like" passwords. Though it generates passwords 5 to 10 times slower than reduction function, which uses direct dependency between hash bytes and the inserted characters, it significantly increases common efficiency in memory (8 to 30 times less memory needed to store these tables) and successful "human-like" passwords recovery probability, while these tables are generated by the same time as tables with the use of "random" reduction function.**

Keywords—rainbow tables; reduction function; user passwords; password generation models; SHA-256

I. Introduction

Nowadays the demands on information security increase as fast as the possibilities of its revelation. In this case, even encrypted data leaks can lead to big problems for companies or individuals. Eventually, improved methods of information protection were developed: stronger hash functions, "salt" encrypting, etc. However, these methods are not always used in modern information processing systems. Moreover, even the use of salt does not guarantee the absence of the possibility of searching passwords through the ready-made tables, especially if the attacker knows how the salt is formed. Therefore, the use of rainbow tables as a password recovery method is in demand today.

Since this method is usually considered in the context of password recovery, the fact that far from all combinations of characters will be real (user) passwords is not taken into account. When creating secret words or phrases, a person tries to make the combination of characters easy to remember. This can be a name, phone number, date of birth, or, for example, a combination of keys on a keyboard that follows a certain rule: several keys in a row in one character row or column, alternating keys on adjacent lines, etc. At the same time, the length of the password increases, and much more memory is required to recover it. Therefore, in this paper, various models of the forming of such passwords are also reviewed and examined for the effectiveness of their use in rainbow tables.

Now there are several software solutions [1][2] that implement the generation of rainbow tables and password recovery. They use reduction function, which has a "random" behavior of the character insertion in the plaintext string (it uses direct dependency between a hash byte and an inserted character). In this paper, we decided to conduct a study of the available solutions [3-5] for generating user passwords in order to identify the effectiveness of their use in the rainbow tables method.

II. Rainbow Tables Method for Password Recovery

Rainbow tables—a variant of the search tables proposed by P. Oechslin in 2003 as a variant of the TMTO (Time-Memory Trade-Off) method proposed by Martin E. Hellman in 1980 [6]. The original method is to calculate all possible pairs of the plaintexts and corresponding encrypted texts by sequentially applying a one-way function (hash function) to the plaintext and the reverse transform function—reduction function—to the plaintext encrypted at the previous step. To reduce memory costs, these pairs are put into several chains of a fixed length. In this case, the chain is formed in a certain way and has known start and end points, which are stored in memory. Accordingly, to restore the entire chain, it is enough to know its start, the end and the reduction function used [7].

However, this method was improved, since the application of the same reduction function within the same chain generally gave many collisions—ending plaintext matches,—which could subsequently affect the amount of memory used. Based on this, Oechslin proposed using a sequence of different fixed reduction functions within a single table to generate chains. This led to a decrease in the possibility of collisions, and, consequently, to a decrease in the possibility of chain mergers in comparison with the original Hellman table. In turn, this reduced the total number of tables needed to achieve a certain coverage of plaintexts. This modification of the original method is used to this day.

A. Existing Solutions

One of the most well-known solutions for rainbow tables available on the Internet is the RainbowCrack software package [1]. This solution is supported by 64-bit Windows OS and Linux OS, however, the possibility of graphic acceleration is available only in the implementation on Windows. Among the supported graphics accelerators (for Windows OS) there are video cards

 978-1-7281-5761-0/20/$31.00

from NVIDIA and AMD, and among the parallel computing technologies, it supports OpenCL and CUDA.

According to the project's official website, perfect tables for an alphabet size of 62 characters, SHA-1 hash function and plaintext length of 1 to 8 characters take 127 GB on the hard drive and have a 99.9% chance of a successful search. Moreover, each table has a chain length of 90,000, and the number of different tables (tables that have a reduction function different from others) is 8. In total, the number of chains is 17062460182. In this case, in order to generate perfect tables, it is necessary to create non-perfect tables, from which chains with repeated end points are deleted, all but one. Moreover, initially, the tables are generated in such a way that a fixed number of bytes is required to write a chain (its start and end points)—16. Therefore, these tables will take 448 GB of memory, according to calculations (1) from the official website of the project:

$$V = (m \times s) / 1024^3 \quad (1)$$

where m is the number of chains in the table and s is the amount of memory required to store one chain in bytes (start and end points).

Then you need to apply the conversion of the type of files, which the tables are written in, into a format with a variable size of the memory occupied by the chain. The transformation can be applied to parts of the same table, but the time spent on generating the table is significantly increased because when you delete duplicate chains, large data arrays are checked and the person manually enters this command.

Another example of an implementation of the rainbow tables method is Cryptohaze [2]. This product is supported by 64-bit Windows OS, Linux OS and macOS (formerly OS X) along with the graphics acceleration support on all these platforms. It supports video cards from all vendors described earlier and the same parallel computing technologies as RainbowCrack.

According to the project's official website, tables for a full set of American printable characters (95 characters), SHA-256 hash function and plaintext length of 1 to 7 characters take 41 GB on the hard drive. At the same time, it is not indicated what is the probability of a successful search in these tables, but the length of each chain is stated as 100,000. As long as we do not have any documentation on this product, we can only suppose that successful password search probability is high. For that we can use both (1) and (2) from [8]:

$$n = \ln (1 - P_y) / \ln (1 - (1 / S)),\ S = \sum_{i=n_1}^{n_2} s^i \quad (2)$$

where n is the number of hash values in the table, P_y is defined successful password recovery probability, S is the amount of different plaintext on a set of characters with the size s and minimum and maximum length of it n_1 and n_2, accordingly.

B. Main problems of Tables Generation

Though these solutions' tables are supposed to contain more than 99% of pre-defined characters combinations, it also contains many "robot-like" combinations. Therefore, it is contrary to the purpose of our work, which can be stated as get rid of such combinations to make the rainbow tables method useful for "human-like" passwords recovery. In other words, make it faster, lighter and still having a high probability of password recovery.

Another problem is memory optimization. For example, RainbowCrack takes more than 3.5 times more memory to temporarily store these tables and then transform them into perfect tables. In this work, we avoid this kind of optimization by simply reduction of every block of chains generated simultaneously. As the experiments show, it has even increased the successful search probability in comparison to non-perfect tables generated the same way and by the same time, but with no reduction to perfect tables. In addition, we tried to move all the possible data to compute to the shared memory of a graphic card. It is made also for the sake of making the computations faster.

Solutions to all these problems can increase average passwords length to recover. For example, perfect rainbow tables with passwords with a length of 1 to 8, that use all 95 ASCII printable characters, take 460 GB on the hard drive and have a successful password search rate of 96,8%, according to RainbowCrack project website [1]. Meanwhile, the minimal required password length and complexity on different Web applications is increasing. Thus, we need a solution, which can efficiently reduce the password space, or make the computation algorithms more efficient, or both. In this paper, we provide such a solution.

III. Statistical Approach to Rainbow Tables

In order to implement any model, which could give us a possibility to generate "human-like" passwords, first, we should choose one. There are not only models, which based on character statistics, but there are models, e.g., based on a simple dictionary or mask generating, or even neural networks.

However, when we talk about implementing these models to the rainbow tables method, we acknowledge that the chosen model should be easy to use, fast to implement and generate tables, take the optimal amount of memory—both for generating and storing, and still have good quality of generated passwords, good "humanity" attributes. These are the main aspects of our choice.

A. Dictionary-Based Model

The first model, which comes to our minds, is the dictionary-based model. It uses pre-defined lists of words or passwords to generate a new one. It can use an existing password or combine several passwords by different masks. The products, that represent this model, are HashCat [3] and John the Ripper [4].

Both these products use a dictionary attack to recover a password along with the statistical methods, which we discuss later in this section. HashCat has different hash functions to recover passwords from, such as SHA functions, MD5, Keccak, phpass, CRC32 and more. John the Ripper has much fewer implementations of hash functions, and has some restrictions on different platforms (e.g., you cannot execute SHA-256 and SHA-512 password recovery on any platform but Linux OS).

In general, the dictionary attack does not allow us to recover any complex passwords, which can contain character replacements based on outward similarity (e.g., "p@55w0rd"). It is suitable for simple password recovery, which contains one or two words with no character change or addition. Thus, it is not our case.

B. Neural Network Model

The next model that we examine in this paper is the neural network model. This model allows us to generate most "human-like" passwords that computer can generate using neural networks. For example, PassGAN [5] uses a generative adversarial network (GAN) to improve the quality of generated passwords. First, it learns how the real passwords look like and then it tries to "cog" itself trying to generate similar passwords. Then the process becomes the mutual learning when the first neural network tries to generate more and more similar to real passwords, and the second tries to distinguish the real passwords from those generated by the generative neural network.

This model gives us an opportunity to use a pre-trained neural network result as the reduction function. But even using the pre-trained model is complex, if we talk about parallel computations and memory efficiency. Thus, it is not our case, too.

C. Markov Model

The next model, which HashCat and John the Ripper also implement, is a Markov model and N-grams. Mostly it uses different leaked password lists, e.g., from hashes.org [9], to gather the statistics of characters. This method is much better than the dictionary attack because it has more freedom of choosing characters to be inserted. Thus, it can help us recover more passwords, and even those passwords, which are not in the set that is used to gather the character statistics. According to experimental results on the comparison between the statistical method in John the Ripper and HashCat [10], John the Ripper significantly outperforms HashCat in recovering passwords for samples that are not in the set used to gather character statistics. However, on statistics collected from some of the recovered passwords, HashCat was able to recover some new passwords from the current selection, and John the Ripper did not recover any. This means that because of the different implementation of Markov model John the Ripper is limited in recovering passwords not from the selection used to gather statistics, but HashCat is not.

This model is simple, has good memory efficiency, is not hard to implement and use, and still can generate complex passwords, which we define as "human-like". Moreover, if we try to make an algorithm, which could give us as much freedom in generating passwords as we can request (e.g., mix "human-like" and random passwords), then this is the perfect model for us. Thus, we use exactly this model.

IV. Implementation

A. Reason for Using SHA-256

The rainbow table method is widely used in cryptanalysis tasks, especially in attacks on password-protected systems. In particular, this method can be used to perform attacks on password systems that are encrypted with some hash function. In 2017, the SHA-1 hash function was stated as unsafe to use in systems that support data encryption, since a significant collision of two files with different content was found [11]. Based on this, many companies were recommended to switch to stronger encryption, for example, SHA-256 or SHA-3, for security and data safety purposes. Therefore, creating tables for the SHA-1 hash function is considered impractical. For this reason, in this paper, we decided to explore the possibilities of optimizing the generation of rainbow tables for more stable hash functions, in particular, for SHA-256.

B. Markov Model Implementation

The basis of our model is character statistics taken from leaked passwords. We implement the Markov model, which is using 3-grams to store the dependencies between characters. Practice shows that using 4-grams for the Markov model in our case is inefficient because of storing and using such large arrays of data (each step in N enlarges the memory used to store these statistics almost 100 times), and using 2-grams is less consistent. In addition, there is a character position parameter, which helps us make complex and consistent passwords.

The statistics are taken from leaked passwords from [9]. All passwords there are depersonalized, so there is no use of them except this research. We took leaked passwords from dropbox.com, linkedin.com, yahoo.com, and haveibeenpwned.com. The last source has two large packages of passwords with many issues like containing some email addresses, or hexadecimal strings of hashes of passwords, which were not recovered. That is why we also implemented a "filter" function, which could get rid of some of these issued strings (e.g., we prohibited using strings with ".net" or ".co" in them).

As we propose a solution, which could help recover longer passwords, we also filtered strings of potential passwords by their lengths. We limited them from 6 to 20 characters long.

The main statistics arrays are the ones with numbers that count every entry of a single character after two previous characters on exact position (*stats*) and the similar one, which contains the characters themselves (*chars*). This is necessary because we need to sort the subarrays in the first array in order to form lists of characters from most common to the least ones. This is made to effectively store these statistics in memory, so we do not need to sort them each time we use them.

Because we need the dependency between a hash value and the inserted characters, and we want to use the memory of a video card as little as possible, we suggested two auxiliary arrays (*rating* and *nums*), which can be produced from the first one. They use almost 8 times less memory, because we store only one byte instead of full unsigned 64-bit integer and we need two bytes containing auxiliary information about the subarray for each subarray, and works fast.

The *rating* array contains bytes, which correspondingly represent the first main array. The value of each cell of this array is calculated like (3)

$$value = (stats[i][j][k][m] \times 255) / localSum \quad (3)$$

where *stats* is the array defined earlier, *localSum* is the sum of *n* first values in stats array, $i \in [0;length]$, where *length* is the

current length of plaintext, j, $k \in [0;94]$, $m \in [0;n-1]$. The indexes i, j, k and m of the *stats* array represent the position of a character in a password, the index of second previous character before this one accordingly to the used alphabet, the index of previous character before this one accordingly to the used alphabet, and the index of a cell, which contains the number of entries of a character, correspondingly. The last level of this four-dimensional array is sorted from the biggest number to the least one as the characters array.

After we calculated the byte value with (3), we subtract it from the previous state of the *bound* variable, which is initially set to 255. We do so until we reach the zero value. The *localSum* value is calculated in such a way that the last number we add to this variable from *stats* array is more or equal to 1/255 if we divide it by the current *localSum* value. This means that not all statistically possible characters get their byte value, which we describe further, how to avoid the reduction of possible characters set. The difference between two adjacent values in the *rating* array represents the reduced probability of a character in a password, which this approach is implemented for.

Two auxiliary values for each subarray of *rating* array are stored in *nums* array. The first value represents the number of *rating* array non-zero values plus one zero value. The second value represents the number of characters that have no entries in any passwords after two specific characters in the current position. This is done to simplify the calculations on the GPU when we need to use one or another part of the *rating* and *chars* array. Each value is byte size, so we store them in one unsigned 16-bit integer one after another.

C. Reduction Function Algorithm

After we calculated *rating* and *nums* arrays, we use them to get the characters of a new plaintext from hash bytes. First, we define the maximum and minimum passwords length (*max_l* and *min_l*, correspondingly) and the parameter we call *humanity* of a password—how "human-like" it is—for the whole table we generate. This is important not only to recover more passwords by mixing pure "human-like" password patterns with some randomization but also to avoid many collisions, which happen because of the statistical approach. Also, we apply XOR to every 32-bit hash part with a value of the position of the plaintext in the chain plus the chain length multiplied by the index of the rainbow table part (as long as we divide the whole rainbow table into small parts in order to be efficient).

We define the *length* of a new plain text as in (4):

$$length = hashByte[0] \bmod (max\_l - min\_l + 1) + min\_l \quad (4)$$

where *hashByte* is the array of bytes from the current hash digest.

Then each next byte of *hashByte* array defines a character of a plaintext. The first character index in *chars* array is defined simply by calculating the second hash byte value by modulo 95—which represents the number of ASCII printable characters with codes from 32 to 127. The first character is one with the calculated index in sorted *chars* array.

The next characters are defined by the *humanity* variable, which is initially set by a user, and the *randomness* variable, which is calculated by applying XOR to the previous hash byte and the current one. If *randomness* value is more or equal to the *humanity* value, we use the characters, which do not have a non-zero value in *rating* array except one first "zero-valued" character, as we described earlier. Then we calculate, whether we use complete zero statistics characters or not, by calculating current hash byte value by modulo 2. If the calculated value is 0, we use them; else—we do not use these characters. Then we calculate the index of an inserted character in *chars* array by adding to the number of characters, which have a non-zero value in *rating* array, plus one the result of the current hash byte value modulo the number of used characters, which we decided to use.

If *randomness* value is less than *humanity* value, then we search for a character, the value of which in *rating* array is less or equal to the current hash byte value.

If on any position we do not have any statistics for a character, then we apply the *shift* variable, which represents a shift in indexes of *rating*, *nums* and *chars* arrays, from which we can get the character. The next character will be calculated as the first one, but with the use of the current hash byte value, and the *shift* value will be set to the current position of the inserted character in plaintext.

Besides, we implement the "random" reduction function, which uses direct dependency between hash bytes and the inserted characters. We prepare the hash values and calculate the plaintext length as we described earlier. Then each character is defined as the character, which has the code in ASCII according to the value of hash byte value modulo 95 (the size of the printable ASCII set) plus 32 (a shift for a "space" character).

We implement our own "random" reduction function because there are some technical issues on RainbowCrack and Cryptohaze solutions. As of January 2020, you cannot reach the RainbowCrack website or download existing rainbow tables for Cryptohaze from its website; therefore, you cannot compare it in the same environment. Still, we can compare overall results like the memory used and the successful search probability.

V. Experimental Results

In order to compare fairly the new approach with the existing one, we used the same environment with the same conditions on both approaches. We used NVIDIA GeForce GTX 660 video card for generating these tables using CUDA implementations of our algorithms and the SHA-256 algorithm [12], which was also optimized by using the shared memory of a video card. The part with getting statistics from passwords, reading and writing tables, and all the preparation for the main experiment were written on C++. We used Microsoft Visual Studio Community 2017 environment with version 15.9.17.

As the hash list we try to recover, we used the hash values of the passwords from the list of top 50 worst passwords of 2018 [13]. Also, we added 30 complex variants of the word "password", which contain character substitutions like 's' to '$' or '5', or 'a' to '@', and case change, as described in [14], in order to check, if this approach works not only on simple "human-like" passwords.

A. Comparison Experiment

We generated two perfect rainbow tables: one with the "random" approach, one with the statistical one. Each table was

set to be generated by 5 hours. The minimum length of passwords was set to 6, the maximum — to 20. The chain length was set to 1000 in both tables. The results of the comparison of these approaches are represented in Table I.

TABLE I. COMPARISON EXPERIMENT RESULTS

Approach	Random	Statistical
Chain amount	640582356	18394754
Memory volume, GB	13.7	0.4
Parts amount	53	37
Efficient generating timea, min	111	267
Successful Password Search Probabilities for SplashData list, %	4.0	98.0
Successful Password Search Probabilities for our list, %	0.0	16.7

[a.] Time spent on generating operations only, without writing to file

B. Results

The experiment shows that the rainbow table with the statistical approach recovered 49 out of 50 hashes from the SplashData hash list and only 5 out of 30 from our list. It recovered the password "passw0rd" as well as the words "Pa$$w0rd" and "P@$$word". It proves that our approach works not only on simple passwords but on the complex ones, too. It is possible that others were not recovered because they were not in the set of passwords, which were taken to gather character statistics. It is also possible that one of these recovered passwords was not in the set used to gather character statistics, which means that it is possible to recover such passwords.

The rainbow table with the "random" reduction function recovered only 2 out of 50 SplashData passwords and 0 out of 30 from our list. These passwords are "666666" and "123123". They are both 6 characters long when the maximum recovered password length in the first rainbow table is 9 (which is maximum for this list of passwords). As we described in Sect. 1.2, you need 460 GB of storage memory to recover passwords 8 characters long for full printable ASCII set using RainbowCrack solution. In our case, we need less than half of 1 GB. To be consistent, in our rainbow tables you cannot recover all 8 characters long passwords, not even a half. The probability of successful recovery with the help of our rainbow tables is not stable, you cannot define it with any formula, because it is complex to count every single possible "human-like" password, and it depends on what we define as "human-like".

VI. CONCLUSIONS

As we can see, the statistical approach is significantly more efficient in memory, thus, in the average password search time, and the average password recovery probability. Nevertheless, it can still be more efficient, if we try to optimize the chain calculations and the reduction function. For example, we noticed that some chains have collisions when a certain set of characters on a certain position due to character statistics is much more intended to become a certain password with almost no choice of either the randomization or changing the characters. So, one of the future goals could be modifying statistics in a way, which will not let the password be inclined to one. For example, we can use different distributions to modify the character statistics, but still save its common behavior (in other words, the same character order in the rating).

Also, we encountered a problem with the successful password recovery probability of our rainbow tables. Though we cannot define it with any formula, maybe we can set the bounds of what we define as "human-like" and then define some inequalities, which can allow us to estimate the real successful password recovery probability. This problem can be solved by linguists and mathematicians with the use of existing works on this topic.

While working on this project, we also tried to compare the common statistics approach to the language-categorized approach. This means that different passwords have different language words in them, depends on the password user, which country is he from, or which language he is used to using. Also, there are some neutral to languages passwords (e.g., "qwerty12345"), which also have some special statistics. The goal was to find any differences in recovering passwords if we split the common statistics to the language-dependent. However, there was found none, so this approach definitely does not improve the efficiency of the rainbow tables method.

REFERENCES

[1] RainbowCrack project, http://project-rainbowcrack.com/.

[2] Cryptohaze project, http://www.cryptohaze.com/.

[3] HashCat project, https://hashcat.net/hashcat/.

[4] John the Ripper project, https://www.openwall.com/john/.

[5] B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz, "PassGAN: A Deep Learning Approach for Password Guessing," arXiv:1709.00440v3 [cs.CR], February 2019.

[6] M.E. Hellman, "A cryptanalytic time-memory trade-off," IEEE Transactions On Information Theory 26(4), pp. 401-406, 1980.

[7] P. Oechslin, "Making a faster cryptanalytic time-memory trade-off," D. Boneh (ed.), CRYPTO 2003. LNCS, vol. 2729, pp.617-630. Springer, Heidelberg, 2003.

[8] L.K. Babenko, E.A. Ishchukova, I.D. Sidorov, "Parallel algorithms for the solution of cryptanalysis problem" ["Parallel'nye algoritmy dlja reshenija zadach zashhity informacii"]. M.: Gorjachaja linija Telekom, 2014. 304 p.

[9] Hashes.org, Shared Community Password Recovery, https://hashes.org/.

[10] John the Ripper and HashCat—Markov mode comparison, http://www.adeptus-mechanicus.com/codex/jtrhcmkv/jtrhcmkv.php.

[11] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1," Annual International Cryptology Conference, 2017, pp. 570-596.

[12] CudaMiner project, https://github.com/cbuchner1/CudaMiner.

[13] The Top 50 Worst Passwords of 2018, https://www.teamsid.com/100-worst-passwords-top-50/

[14] L. Zhang, C. Tan, and F. Yu, “An improved rainbow table attack for long passwords,” Procedia Computer Science, vol. 107, pp. 47–52. 2017.