Rainbow Table Optimization for Password Recovery

Vrizlynn L. L. Thing, Hwei-Ming Ying

Cryptography & Security Department
Institute for Infocomm Research, Singapore
{vriz,hmying}@i2r.a-star.edu.sg

***Abstract*—As users become increasingly aware of the need to adopt strong password, it also brings challenges to digital forensics investigators due to the password protection of potential evidence data. In this paper, we discuss existing password recovery methods and present a design of a time-memory trade-off pre-computed table coupled with a new sorting algorithm. We also propose 2 new storage methods and evaluated their performance based on storage conservation and success rate improvement. Considering both alpha-numeric passwords and passwords consisting of any printable ASCII character, we show that we are able to optimize the rainbow table performance through an improvement of up to 26.13% in terms of password recovery success rate, and an improvement of up to 28.57% in terms of storage conservation, compared to the original rainbow tables.**

Keywords - Digital forensics, password recovery, rainbow table optimization, time-memory trade-off, cryptanalysis.

I. Introduction

In computer and information security, the use of passwords is essential for users to protect their data and to ensure a secured access to their systems/machines. However, in digital forensics, the use of password protection presents a challenge for investigators while conducting examinations. As mentioned in [3], compelling a suspect to surrender his password would force him to produce evidence that could be used to incriminate him, thereby violating his Fifth Amendment right against self-incrimination. Therefore, this presents a need for the authorities to have the capability to access a suspect's data without expecting his assistance. While there exist methods to decode hashes to reveal passwords used to protect potential evidence, lengthier passwords with larger characters sets have been encouraged to thwart password recovery. Awareness of the need to use stronger passwords and active adoption have rendered many existing password recovery tools inefficient or even ineffective.

The more common methods of password recovery techniques are guessing, dictionary, brute force and more recently, using rainbow tables. The guessing method is attempting to crack passwords by trying "easy-to-remember", common passwords or passwords based on a user's personal information (or a fuzzy index of words on the user's storage media). A statistical analysis of 28,000 passwords recently stolen from a popular U.S. website revealed that 16% of the users took a first name as a password and 14% relied on "easy-to-remember" keyboard combinations [4]. Therefore, the guessing method can be quite effective in some cases where users are willing to compromise security for the sake of convenience.

The dictionary attack method composes of loading a file of dictionary words into a password cracking tool to search for a match of their hash values with the stored one. Examples of password cracking tools include Cain and Abel [5], John the Ripper [6] and LCP [7].

In the brute force cryptanalysis attack, every possible combination of the password characters is attempted to perform a match comparison. It is an extremely time consuming process but the password will be recovered eventually if a long enough time is given. Cain and Abel, John the Ripper as well as LCP are able to conduct brute force attacks.

In [8-11], the authors studied on the recovery of passwords or encryption keys based on the collision of hashes in specific hashing algorithms. These methods are mainly used to research on the weakness of hashing algorithms. They are too high in complexity and time consuming to be used for performing password recovery during forensics investigations. The methods are also applicable to specific hashing algorithms only.

In [12], Hellman introduced a method, which involves a trade-off between the computation time and storage space needed to recover the plaintext from its hash value. It can be applied to retrieve Windows login passwords encrypted into LM or NTLM hashes [13], as well as passwords in applications using these hashing algorithms. Passwords encrypted with hashing algorithms such as MD5 [14], SHA-2 [15] and RIPEMD-160 [16] are also susceptible to this recovery method. In addition, this method is applicable to many searching tasks including the knapsack and discrete logarithm problems.

In [17], Oechslin proposed a faster cryptanalytical time-memory trade-off method, which is an improvement over Hellman's method. Since then, this method has been widely used and implemented in many popular password recovery tools. The pre-computed tables that are generated in this method are known as the rainbow tables.

In [18], Narayanan and Shmatikov proposed using standard Markov modeling techniques from natural language processing to reduce the password space to be searched, combined with the application of the time-memory trade-off method to analyse the vulnerability of human-memorable passwords. It was shown that 67.6% of the passwords can be successfully recovered using a 2x10^9 search space. However, the limitation of this method is that the passwords were assumed to be human-memorable character-sequence passwords.

In this paper, we present a new design of an enhanced rainbow table [1,2] by proposing a novel time-memory trade-off pre-computed table structure and a rainbow table sorting algorithm. Maintaining the core functionality of the rainbow tables, we optimized the storage space requirement while achieving the same success rate and search speed.

The rest of the paper is organized as follow. In Section 2, we present a discussion on the existing time-memory trade-off password recovery methods. We then give an overview of our enhanced rainbow table design in Section 3. We describe the design of our sorting algorithm in Section 4. Analysis and evaluation are presented in Section 5. The description of the 2 new proposed storage methods is provided in Section 5 due to the importance of their considerations during evaluations. Conclusions follow in Section 6.

II. Analysis of Existing Work

The idea of a general time-memory tradeoff was first proposed by Hellman in 1980 [12]. In the context of password recovery, we describe the Hellman algorithm as follows.

We let X be the plaintext password and Y be the corresponding stored hash value of X. Given Y, we need to find X, which satisfies h(X) = Y, where h is a known hash function. However, finding X = h^{-1}(Y) is feasibly impossible since hashes are computed using one-way functions, where the reversal function, h^{-1}, is unknown. Hellman suggested taking the plaintext values and applying alternate hashing and reducing, to generate a pre-computed table.

For example, the corresponding 128-bit hash value for a 7-character password (composed from a character set of English alphabets), is obtained by performing the password hashing function on the password. With a reduction function such as $H \bmod 26^7$, where H is the hash value converted to its decimal form, the resulting values are distributed in a best-effort uniform manner. For example, if we start with the initial plaintext value of "abcdefg" and upon hashing, we get a binary output of 0000000....000010000000....01, which is 64 '0's and a '1' followed by 62 '0's and a '1'. $H = 2^{63} + 1$ = 9223372036854775809. The reduction function will then convert this value to "3665127553", which corresponds to a plaintext representation "lwmkgij", computed from $(11(26^6) + 22(26^5) + 12(26^4) + 10(25^3) + 6(26^2) + 8(26^1) + 9(26^0)$. After a pre-defined number of rounds of hashing and reducing (making up a chain), only the initial and final plaintext values are stored. Therefore, only the "head" and "tail" of a chain are stored in the table. Using different initial plaintexts, the hashing and reducing operations are repeated, to generate a larger table (of increasing rows or chains). A larger table will theoretically contain more pre-computed values (i.e., disregarding hash collisions), thereby increasing the success rate of password recovery, while taking up more storage space. The pre-defined number of rounds of hashing and reducing will also increase the success rate by increasing the length of the "virtual" chain, while bringing about a higher computational overhead.

To recover a plaintext from a given hash, a reduction operation is performed on the hash and a search for a match of the computed plaintext with the final value in the table is conducted. If a match is not found, the hashing and reducing operations are performed on the computed plaintext to arrive at a new plaintext so that another round of search to be made. The maximum number of rounds of hashing, reducing and searching operations is determined by the chain length. If the hash value is found in a particular chain, the values in the chain are then worked out by performing the hashing and reducing functions to arrive at the plaintext giving the specific hash value. Unfortunately, there is a likelihood that chains with different initial values may merge due to collisions. These merges will reduce the number of distinct hash values in the chains and therefore, diminish the rate of successful recovery. The success rate can be increased by using multiple tables with each table using a different reduction function. If we let P(t) be the success rate of using t tables, then $P(t) = 1 - (1 - P(1))^t$, which is an increasing function of t since P(1) is between 0 and 1. Hence, introducing more tables increase the success rate but also cause an increase in both the computational complexity and storage space.

In [19], Rivest suggested a method of using distinguished points as end points for chains. Distinguished points are keys, which satisfy a given criteria, e.g., the first or last q bits are all 0. In this method, the chains are not generated with a fixed length but they terminate upon reaching pre-defined distinguished points. This method decreases the number of memory lookups compared to Hellman's method and is capable of loop detection. If a distinguished point is not obtained after a large finite number of operations, the chain is suspected to contain a loop and is discarded. Therefore, the generated chains are free of loops. One limitation is that the chains will merge if there is a collision within the same table. The variable lengths of the chains will also result in an increase in the number of false alarms. Additional computations are also required to determine if a false alarm has occurred.

In 2003, Oechslin proposed a new table structure [17] to reduce the probability of merging occurrences. These rainbow chains use multiple reduction functions such that there will only be merges if the collisions occur at the same positions in both chains. An experiment was carried out and presented in Oechslin's paper. It showed that given a set of parameters, which is constant in both scenarios, the measured coverage in a single rainbow table is 78.8% compared to the 75.8% from the classical tables of Hellman with distinguished points. In addition, the number of calculations needed to perform the search is reduced as well.

In the following sections, we present our enhanced rainbow table [2] with a novel sorting algorithm [1], and propose 2 new storage methods, so that password lookup in the stored tables can be optimized.

III. Enhanced Rainbow Table

In this section, we present a new design of a time-memory trade-off precomputed table structure.

In this design, the same reduction functions as in the rainbow table method are used. The novelty lies in the chains generation technique. Instead of taking a large set of plaintexts as our initial values, we systematically choose a much smaller unique set. We choose a plaintext and compute its corresponding hash value by applying the password hash algorithm. We let the resulting hash value written in decimal digits be *H*. Following that, we compute *(H+1) mod* 2^j, *(H+2) mod* 2^j*,......, (H+k) mod* 2^j for a variable *k*, where *j* is the number of bits of the hash output value. For example, in MD5 hash, $j = 128$. These hash values are then noted as the branches of the above chosen initial plaintext. We then proceed by applying alternate hashing and reducing operations to all these branches. We call this resulting extended chain, a block. The final values of the plaintexts are then stored with this 1 initial plaintext value. We perform the same operations for the other plaintexts. These sets of initial and final values make up the new pre-computed table.

To recover a password given a hash, we apply reducing and hashing operations alternatively until we obtain a plaintext that corresponds to one of the stored final values, as in the rainbow table method. After which, we generate the corresponding branch (e.g., if *k = 99* and *chain id = 212*, the initial value is the initial plaintext in the third block and the branch id is 12), till the value of the password hash is reached.

A. Differences and Similarities in the designs

We identify and list the differences and similarities between the design of our new method and the rainbow table method as follow:

- Both use *n* reduction functions.
- Instead of storing the initial and final values as a pair as in the rainbow table, the initial value is stored with multiple output plaintexts after a series of hashing and reducing operations. This results in a large amount of storage conservation in the new method.
- The hashes *H, (H+1) mod* 2^j, *(H+2) mod* 2^j*,...... , (H+k) mod* 2^j are calculated in order to generate subsequent hashes, resulting in the uniqueness of the values in the 1st column of hashes in the new method. The uniqueness of the hash values is guaranteed unless the total number of hashes is greater than 2^j. This situation is not likely to happen as it assumes an extremely large table, which fully stores all the possible pre-computed values.
- In this new design, the recovery of some passwords in the 1st column is not possible as they are not stored in the first place. However, we have shown in our analysis and evaluation [2] that the effect is neglible.

IV. Sorting Algorithm

The main drawback of the proposed enhanced rainbow table is that each password search will incur a significant amount of time complexity. The reason is that the passwords cannot be sorted in the usual alphabetical order now, since in doing so, the information of its corresponding initial hash value will be lost. The lookup will then have to rely on checking every single stored password in the table. Therefore, we propose a sorting algorithm so that password lookup in the stored tables can be optimized.

We require a sorting of the “tail” passwords to achieve a fast lookup. Therefore, we introduce special characters that cannot be found on the keyboard (i.e., non-printable ASCII characters). There are altogether 161 such characters and we assume that these non-printable ASCII characters do not form any of the character set of the passwords. We insert a number of these special characters into the passwords that we store. The manner in which these special characters are inserted will provide the information on the original position of the passwords in the rainbow table after the table has been re-arranged in alphabetical order. The consequence is that this will incur more storage space but we will illustrate later that the increase in storage space is minimal and is also lesser than the original rainbow table’s storage requirement. The advantage of this sorting algorithm is that the passwords can now be sorted and thus a password lookup can be optimized.

A. Algorithm Design

Definition of notations:
Y = total number of special characters available
w = number of special characters in password
m = length of password
x = special character in password (labelled x_1, x_2,.....), $1 \leq x_i \leq Y$
p = location of a special character within password (labelled p_0, p_1,.....), $0 \leq p_i \leq m$, where x_i is placed at location p_{i-1} within a password

$\binom{n}{r} = \frac{n!}{(n-r)!r!}$

From here on, position refers to the original position of a password in the rainbow table, while placement or location of a special character refers to its location in a password.

Password Position Computation

As an example, let 0000000 denote a 7-character password. The 161 non-printable ASCII characters are used as special characters x_1, x_2,........, x_{161}, and are represented by numeric values from 1 to 161, respectively. The 8 possible locations of the special characters in a password are represented as underlines in _0_0_0_0_0_0_0_. Each location can hold more than one special character.

For example, 0000000 does not carry any special character and is at position 0. In 0000000x_1, x_1 is the first (scanning from rightmost) special character in the password, as denoted by its subscript value of 1. The position of this password depends on the numeric value represented by the special character, x_1. Therefore, the position is from 1 to 161 depending on which of the 161 special characters is used (i.e., 0000000x_1 is at position 1 if x_1 = 1, and at position 161 if x_1 = 161). Continuing in this manner, 000000$x_1$0 is at position 162 if x_1 = 1, and at position 322 if x_1 = 161. Therefore, $x_1$0000000 is at position 1128 if x_1 = 1, and at position 1288 if x_1 = 161.

Ater covering all the locations for special characters in the password by using only 1 character but without completing the allocation of special characters for all the passwords in the password space, we can increase the number of allocated special characters in the password, one at a time. For the insertion of 2 characters, 0000000x_2x_1 is at position 1289 if x_1=x_2=1. 0000000x_2x_1 is at position 1290 if x_2 = 1 and x_1 = 2. 0000000x_2x_1 is at position 1291 if x_2 = 1 and x_1 = 3. Continuing in this manner, 0000000x_2x_1 is at position 27209 if x_2 = x_1 = 161, 000000$x_2$0x_1 is at position 27210 if x_2 = x_1 = 1, and so on.

We derive the following formulas for the computation of the original position of a password in the rainbow table.

w=1: Original position of password
$\mathrm{Y}p_0 + x_1$

w=2: Original position of password
$\mathrm{Y}x_2 + x_1 + \mathrm{Y}^2\binom{p_1+1}{2} + \mathrm{Y}^2\mathrm{p}_0 + \mathrm{Ym}$

w=3: Original position of password
$\mathrm{Y}^2\mathrm{x}_3 + \mathrm{Y}x_2 + x_1 + \mathrm{Y}^3\binom{p_2+2}{3} + \mathrm{Y}^3\binom{p_1+1}{2} + \mathrm{Y}^3\mathrm{p}_0 + \mathrm{Y}^2\binom{m+1}{2} + \mathrm{Y}^2\mathrm{m} + \mathrm{Ym}$

w=4: Original position of password
$\mathrm{Y}^3\mathrm{x}_4 + \mathrm{Y}^2\mathrm{x}_3 + \mathrm{Yx}_2 + \mathrm{x}_1 + \mathrm{Y}^4\binom{p_3+3}{4} + \mathrm{Y}^4\binom{p_2+2}{3} + \mathrm{Y}^4\binom{p_1+1}{2} + \mathrm{Y}^4\mathrm{p}_0 + \mathrm{Y}^3\binom{m+2}{3} + \mathrm{Y}^3\binom{m+1}{2} + \mathrm{Y}^3\mathrm{m} + \mathrm{Y}^2\binom{m+1}{2} + \mathrm{Y}^2\mathrm{m} + \mathrm{Ym}$

For a general w, the original position is given by
$\sum_{i=0}^{w-1}(Y^i x_{i+1} + \mathrm{Y}^w\binom{p_i+i}{i+1}) + \sum_{i=0}^{w-2}\sum_{j=0}^{i}\mathrm{Y}^{i+1}\binom{m+j}{j+1}$

V. Analysis

In this section, we present an analysis of the proposed enhanced rainbow table and its sorting algorithm. First, we analyse the maximum number of special characters required to sort tables of different sizes and password lengths, as well as demonstrate the storage conservation achieved. Next, we analyse the improvement in success rate of password recovery in the event of storage limitation. The 2 new storage methods are proposed and the impact on the storage conservation and success rate are demonstrated in this section.

A. Storage Conservation Analysis

Number of positions that can be assigned without using any special character
= 1

Number of positions that can be assigned using 1 special character
= Y(m+1)

Number of positions that can be assigned using 2 special characters
= $\mathrm{Y}^2[\mathrm{m}+1+ \binom{m+1}{2}]$

Number of positions that can be assigned using 3 special characters
= $\mathrm{Y}^3[\mathrm{m}+1+ 2\binom{m+1}{2} + \binom{m+1}{3}]$

Number of positions that can be assigned using 4 special characters
= $\mathrm{Y}^4[\mathrm{m}+1+ 3\binom{m+1}{2} + 3\binom{m+1}{3} + \binom{m+1}{4}]$

For w≥1, the number of positions that can be assigned using exactly w special characters
= $\mathrm{Y}^w\sum_{i=0}^{w-1}\binom{w-1}{i}\binom{m+1}{i+1}$

Total number of positions that can be identified using at most w special characters (inclusive of positions that can be identified for number of special characters smaller than w)
= $\sum_{i=0}^{w}\sum_{j=0}^{i}\mathrm{Y}^i\binom{m+j-1}{j}$

Table I shows the number of positions that can be assigned given a pre-defined Y, m and w.

TABLE I: Total Number of Positions in Enhanced Rainbow Table

Y	m	w	Total Number of Positions
161	7	1	1,289
161	7	2	934,445
161	7	3	501,728,165
161	7	4	222,228,147,695
161	7	5	85,897,316,654,087
161	7	6	29,972,224,023,967,164
161	8	1	1,450
161	8	2	1,167,895
161	8	3	689,759,260
161	8	4	333,279,388,555
161	8	5	139,555,298,211,442
161	8	6	52,440,627,036,009,328
161	9	1	1,611
161	9	2	1,427,266
161	9	3	919,549,086
161	9	4	481,326,791,401
161	9	5	217,048,911,627,003
161	9	6	87,385,501,807,956,800
161	10	1	1,772
161	10	2	1,712,558
161	10	3	1,195,270,924
161	10	4	673,765,410,165
161	10	5	325,525,142,663,568
161	10	6	139,795,049,776,791,264

Let s be the total number of passwords to be stored in a rainbow table. Therefore, the total storage space required by the original rainbow table is (2 * m * s) bytes. In the enhanced rainbow table method, for s < $\sum_{i=0}^{w}\sum_{j=0}^{i}\mathrm{Y}^i\binom{m+j-1}{j}$, at most w special characters are needed to be inserted to each password to identify its position. However, due to the use of special characters, the passwords to be stored are no longer of constant length. We propose two new methods to the storage of the passwords with their inserted special characters. In method 1, the passwords are still stored side by side as in the original rainbow table method. Retrieval of passwords for checking is performed at a specific fixed length. The w used in method 1 must be a fixed length too. The total number of positions that can be identified is reduced as they do not include positions that can be identified for number of special characters smaller than w. The total number of positions in storage method 1 is shown in Table II.

TABLE II: Total Number of Positions in Storage Method 1 (side by side)

Y	m	w	Total Number of Positions
161	7	1	1,288
161	7	2	933,156
161	7	3	500,793,720
161	7	4	221,726,419,530
161	7	5	85,675,088,506,392
161	7	6	29,886,326,707,313,076
161	8	1	1,449
161	8	2	1,166,445
161	8	3	688,591,365
161	8	4	332,589,629,295
161	8	5	139,222,018,822,887
161	8	6	52,301,071,737,797,880
161	9	1	1,610
161	9	2	1,425,655
161	9	3	918,121,820
161	9	4	480,407,242,315
161	9	5	216,567,584,835,602
161	9	6	87,168,452,896,329,808
161	10	1	1,771
161	10	2	1,710,786
161	10	3	1,193,558,366
161	10	4	672,570,139,241
161	10	5	324,851,377,253,403
161	10	6	139,469,524,634,127,680

In storage method 2, we propose storing the passwords line by line. Therefore, a special character has to be used for delimitation purpose. A good choice would be the line feed character. Y is then reduced to 160. The w used in method 2 can be of a variable length. The total number of positions that can be identified still includes positions that can be identified for number of special characters smaller than w. The total number of positions in storage method 2 is shown in Table III.

In storage method 1, the total storage space needed = (m+w)(s)

In storage method 2, the total storage space needed $=\sum_{i=1}^{w-1}\sum_{j=0}^{i-1}\mathrm{Y}^i(\mathrm{m+i+1})\binom{i-1}{j}\binom{m+1}{j+1}$ + (m+w+1)[s + 1 - $\sum_{i=0}^{w-1}\sum_{j=0}^{i}\mathrm{Y}^i\binom{m+j-1}{j}$]

We consider two main scenarios in the performance evaluation based on storage space. In the first scenario, any alpha-numeric characters can be used in the passwords. There would be 62 characters in total. In the second scenario, we increase the password character set to include all printable ASCII characters, which will consist of 95 characters.

TABLE III: Total Number of Positions in Storage Method 2 (line by line)

Y	m	w	Total Number of Positions
160	7	1	1,281
160	7	2	922,881
160	7	3	492,442,881
160	7	4	216,761,242,881
160	7	5	83,263,980,442,881
160	7	6	28,872,966,636,442,880
160	8	1	1,441
160	8	2	1,153,441
160	8	3	676,993,441
160	8	4	325,080,193,441
160	8	5	135,276,811,393,441
160	8	6	50,517,256,459,393,440
160	9	1	1,601
160	9	2	1,409,601
160	9	3	902,529,601
160	9	4	469,484,929,601
160	9	5	210,394,400,129,601
160	9	6	84,180,360,480,129,600
160	10	1	1,761
160	10	2	1,691,361
160	10	3	1,173,147,361
160	10	4	657,188,507,361
160	10	5	315,544,561,307,361
160	10	6	134,667,490,289,307,360

Scenario 1: Alpha-numeric character set in passwords

We consider the cases where the passwords are 7, 8, 9 and 10 characters in length. Table IV shows the required number of special characters, w, to store all the passwords when the number of hashing and reduction functions (i.e., virtual columns) in the rainbow table are 30,000.

TABLE IV: Required w for Alpha-Numberic Passwords in Both Storage Methods when virtual columns are 30,000

Password Length	Total Password Space (s)	w in Method 1	w in Method 2
7	117,387,154	3	3
8	7,278,003,520	4	4
9	451,236,218,209	4	4
10	27,976,645,528,945	5	5

Table V shows the required number of special characters, w, to store all the passwords when the virtual columns in the rainbow table are 100,000.

Table VI shows the storage requirement to store all the passwords when the virtual columns in the rainbow table are 30,000, while Table VII shows the storage requirement when the virtual columns are 100,000. In the case when the number of virtual columns in the rainbow table is set to 30,000, the improvement in terms of storage conservation of method 1 over the original rainbow table was 28.57%,

TABLE V: Required w for Alpha-Numberic Passwords in Both Storage Methods when virtual columns are 100,000

Password Length	Total Password Space (s)	w in Method 1	w in Method 2
7	35,216,147	3	3
8	2,183,401,056	4	4
9	135,370,865,463	4	4
10	8,392,993,658,684	5	5

25%, 27.78%, and 25% for password length of 7, 8, 9 and 10, respectively. Comparing method 1 over method 2, the improvement in terms of storage conservation was 9.03%, 7.03%, 7.13%, and 6.11% for password length of 7, 8, 9 and 10, respectively.

In the case when the number of virtual columns in the rainbow table is set to 100,000, the improvement in terms of storage conservation of method 1 over the original rainbow table was 28.57%, 25%, 27.78%, and 25% for password length of 7, 8, 9 and 10, respectively. Comparing method 1 over method 2, the improvement in terms of storage conservation was 8.87%, 5.43%, 7.10%, and 5.79% for password length of 7, 8, 9 and 10, respectively.

TABLE VI: Storage Requirement for Alpha-Numberic Passwords when virtual columns is 30,000

Password Length	Original Rainbow Table	Method 1	Method 2
7	1,643,420,156B 1.53GB 0.0015TB	1,173,871,540B 1.09GB 0.0011TB	1,290,334,534B 1.20GB 0.0012TB
8	116,448, 056,320B 108.45GB 0.1059TB	87,336, 042,240B 81.34GB 0.0794TB	93,935, 897,440B 87.48GB 0.0854TB
9	8,122,251, 927,762B 7,564.44GB 7.39TB	5,866, 070,836,717B 5,463.20GB 5.34TB	6,316,403, 114,126B 5,882.61GB 5.74TB
10	559,532,910, 578,900B 521,105.63GB 508.89TB	419,649,682, 934,175B 390,829.22GB 381.67TB	446,967,965, 115,280B 416,271.36GB 406.51TB

Scenario 2: All printable ASCII character set in passwords

We consider the cases where the passwords are 7, 8, and 9 characters in length. Table VIII shows the required number of special characters, w, to store all the passwords when the number of hashing and reduction functions (i.e., virtual columns) in the rainbow table are 30,000.

Table IX shows the required number of special characters, w, to store all the passwords when the virtual columns in the rainbow table are 100,000.

TABLE VII: Storage Requirement for Alpha-Numberic Passwords when virtual columns is 100,000

Password Length	Original Rainbow Table	Method 1	Method 2
7	493,026,058B 0.46GB 0.00045TB	352,161,470B 0.33GB 0.00032TB	386,453,457B 0.36GB 0.00035TB
8	34,934, 416,896B 32.54GB 0.0318TB	26,200, 812,672B 24.40GB 0.0238TB	27,706, 065,408B 25.80GB 0.0252TB
9	2,436,675, 578,334B 2,269.33GB 2.22TB	1,759,821, 251,019B 1,638.96GB 1.60TB	1,894,288, 175,682B 1764.19GB 1.72TB
10	167,859,873, 173,680B 156,331.69GB 152.67TB	125,894,904, 880,260B 117,248.77GB 114.50TB	133,629,535, 191,104B 124,452.20GB 121.54TB

TABLE VIII: Required w for All Printable ASCII Character Passwords in Both Storage Methods when virtual columns are 30,000

Password Length	Total Password Space (s)	w in Method 1	w in Method 2
7	2,327,790,987	4	4
8	221,140,143,764	4	4
9	21,008,313,657,487	5	5

TABLE IX: Required w for All Printable ASCII Character Passwords in Both Storage Methods when virtual columns are 100,000

Password Length	Total Password Space (s)	w in Method 1	w in Method 2
7	698,337,297	4	4
8	66,342,043,129	4	4
9	6,302,494,097,247	5	5

Table X shows the storage requirement to store all the passwords when the virtual columns in the rainbow table are 30,000, while Table XI shows the storage requirement when the virtual columns are 100,000. In the case when the number of virtual columns in the rainbow table is set to 30,000, the improvement in terms of storage conservation of method 1 over the original rainbow table was 21.43%, 25%, and 22.22% for password length of 7, 8 and 9, respectively. Comparing method 1 over method 2, the improvement in terms of storage conservation was 6.69%, 7.67%, and 6.53% for password length of 7, 8 and 9, respectively.

In the case when the number of virtual columns in the rainbow table is set to 100,000, the improvement in terms of storage conservation of method 1 over the original rainbow table was 21.43%, 25%, and 22.22% for password length of 7, 8 and 9, respectively. Comparing method 1 over method

2, the improvement in terms of storage conservation was 2.60%, 7.62%, and 6.20% for password length of 7, 8 and 9, respectively.

TABLE X: Storage Requirement for All Printable ASCII Character Passwords when virtual columns is 30,000

Password Length	Original Rainbow Table	Method 1	Method 2
7	32,589, 073,818B 30.35GB 0.0296TB	25,605, 700,857B 23.85GB 0.0233TB	27,440, 124,804B 25.56GB 0.0250TB
8	3,538,242, 300,224B 3,295.24GB 3.22TB	2,653,681, 725,168B 2,471.43GB 2.41TB	2,874,143, 720,612B 2,676.75GB 2.61TB
9	378,149,645, 834,766B 352,179.30GB 343.93TB	294,116,391, 204,818B 273,917.23GB 267.50TB	314,654,315, 991,905B 293,044.67GB 286.18TB

TABLE XI: Storage Requirement for All Printable ASCII Character Passwords when virtual columns is 100,000

Password Length	Original Rainbow Table	Method 1	Method 2
7	9,776, 722,158B 9.11GB 0.0089TB	7,681, 710,267B 7.15GB 0.0070TB	7,886, 680,524B 7.35GB 0.0072TB
8	1,061,472, 690,064B 988.57GB 0.9654TB	796,104, 517,548B 741.43GB 0.7241TB	861,768, 412,357B 802.58GB 0.7838TB
9	113,444,893, 750,446B 105,653.79GB 103.18TB	88,234,917, 361,458B 82,175.17GB 80.25TB	94,067,022, 588,305B 87,606.74GB 85.55TB

B. Success Rate Improvement Analysis

Here, we analyse the improvement in terms of success rate of password recovery. To do so, we set the storage requirement to be a fixed value and compute the achieveable success rate. Table XII to Table XV shows the success rate when the storage is capped at a certain value and the virtual columns are 30,000 for different password lengths, while Table XVI to Table XIX shows the evaluation when the virtual columns are 100,000 instead. The password character set consists of the alpha-numeric characters.

TABLE XII: Success Rate for 7-Character Alpha-Numeric Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
0.5GB	$\frac{38347922}{117387154}$ =32.67%	$\frac{53687091}{117387154}$ =45.74%	$\frac{48890461}{117387154}$ =41.65%
1GB	$\frac{76695844}{117387154}$ =65.34%	$\frac{107374182}{117387154}$ =91.47%	$\frac{97696907}{117387154}$ =83.23%
1.5GB	$\frac{115043766}{117387154}$ =98%	$\frac{117387154}{117387154}$ =100%	$\frac{117387154}{117387154}$ =100%

TABLE XIII: Success Rate for 8-Character Alpha-Numeric Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
50GB	$\frac{3355443200}{7278003520}$ =46.10%	$\frac{4473924266}{7278003520}$ =61.47%	$\frac{4181941501}{7278003520}$ =57.46%
75GB	$\frac{5033164800}{7278003520}$ =69.16%	$\frac{6710886400}{7278003520}$ =92.21%	$\frac{6246829624}{7278003520}$ =85.83%
100GB	$\frac{6710886400}{7278003520}$ =92.21%	$\frac{7278003520}{7278003520}$ =100%	$\frac{7278003520}{7278003520}$ =100%

We observe from the evaluations that in the event of storage limitation, method 1 performs significantly better in

TABLE XIV: Success Rate for 9-Character Alpha-Numeric Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
1TB	$\frac{61083979320}{451236218209}$ =13.54%	$\frac{84577817521}{451236218209}$ =18.74%	$\frac{78601112041}{451236218209}$ =17.42%
3TB	$\frac{183251937962}{451236218209}$ =40.61%	$\frac{253733452563}{451236218209}$ =56.23%	$\frac{235674201723}{451236218209}$ =52.23%
5TB	$\frac{305419896604}{451236218209}$ =67.69%	$\frac{422889087606}{451236218209}$ =93.72%	$\frac{392747291405}{451236218209}$ =87.04%

TABLE XV: Success Rate for 10-Character Alpha-Numeric Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
100TB	$\frac{5497558138880}{27976645528945}$ =19.65%	$\frac{7330077518506}{27976645528945}$ =26.20%	$\frac{6913095382840}{27976645528945}$ =24.71%
300TB	$\frac{16492674416640}{27976645528945}$ =58.95%	$\frac{21990232555520}{27976645528945}$ =78.60%	$\frac{20656990730040}{27976645528945}$ =73.84%
500TB	$\frac{27487790694400}{27976645528945}$ =98.25%	$\frac{27976645528945}{27976645528945}$ =100%	$\frac{27976645528945}{27976645528945}$ =100%

TABLE XVI: Success Rate for 7-Character Alpha-Numeric Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
0.1GB	$\frac{7669584}{35216147}$ =21.78%	$\frac{10737418}{35216147}$ =30.49%	$\frac{9845303}{35216147}$ =27.96%
0.2GB	$\frac{15339168}{35216147}$ =43.56%	$\frac{21474836}{35216147}$ =60.98%	$\frac{19606593}{35216147}$ =55.68%
0.3GB	$\frac{23008753}{35216147}$ =65.34%	$\frac{32212254}{35216147}$ =91.47%	$\frac{29367882}{35216147}$ =83.39%

terms of password recovery success rate compared to both the original rainbow table and method 2, consistently. Based on the results above, the improvement in success rate of method 1 over the original rainbow table can reach up to 26.13%.

Table XX to Table XXII shows the success rate when the storage is capped at a certain value and the virtual columns are 30,000 for different password lengths, while Table XXIII to Table XXV shows the evaluation when the virtual columns are 100,000 instead. The password character set consists of all the printable ASCII characters.

TABLE XVII: Success Rate for 8-Character Alpha-Numeric Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
10GB	$\frac{671088640}{2183401056}$ =30.74%	$\frac{894784853}{2183401056}$ =40.98%	$\frac{878120504}{2183401056}$ =40.22%
20GB	$\frac{1342177280}{2183401056}$ =61.47%	$\frac{1789569706}{2183401056}$ =81.96%	$\frac{1704075753}{2183401056}$ =78.05%
30GB	$\frac{2013265920}{2183401056}$ =92.21%	$\frac{2183401056}{2183401056}$ =100%	$\frac{2183401056}{2183401056}$ =100%

TABLE XVIII: Success Rate for 9-Character Alpha-Numeric Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
1TB	$\frac{61083979320}{135370865463}$ =45.12%	$\frac{84577817521}{135370865463}$ =62.48%	$\frac{78601112041}{135370865463}$ =58.06%
1.5TB	$\frac{91625968981}{135370865463}$ =67.69%	$\frac{126866726281}{135370865463}$ =93.72%	$\frac{117869384461}{135370865463}$ =87.07%
2TB	$\frac{122167958641}{135370865463}$ =90.25%	$\frac{135370865463}{135370865463}$ =100%	$\frac{135370865463}{135370865463}$ =100%

TABLE XIX: Success Rate for 10-Character Alpha-Numeric Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
50TB	$\frac{2748779069440}{8392993658684}$ =32.75%	$\frac{3665038759253}{8392993658684}$ =43.67%	$\frac{3477121546040}{8392993658684}$ =41.43%
100TB	$\frac{5497558138880}{8392993658684}$ =65.50%	$\frac{7330077518506}{8392993658684}$ =87.34%	$\frac{6913095382840}{8392993658684}$ =82.37%
150TB	$\frac{8246337208320}{8392993658684}$ =98.25%	$\frac{8392993658684}{8392993658684}$ =100%	$\frac{8392993658684}{8392993658684}$ =100%

TABLE XX: Success Rate for 7-Character Printable ASCII Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
10GB	$\frac{766958445}{2327790987}$ =32.95%	$\frac{976128930}{2327790987}$ =41.93%	$\frac{935898773}{2327790987}$ =40.21%
20GB	$\frac{1533916891}{2327790987}$ =65.90%	$\frac{1952257861}{2327790987}$ =83.87%	$\frac{1830683626}{2327790987}$ =78.64%
30GB	$\frac{2300875337}{2327790987}$ =98.84%	$\frac{2327790987}{2327790987}$ =100%	$\frac{2327790987}{2327790987}$ =100%

TABLE XXI: Success Rate for 8-Character Printable ASCII Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
1TB	$\frac{68719476736}{221140143764}$ =31.08%	$\frac{91625968981}{221140143764}$ =41.43%	$\frac{84629982776}{221140143764}$ =38.27%
2TB	$\frac{137438953472}{221140143764}$ =62.15%	$\frac{183251937962}{221140143764}$ =82.87%	$\frac{169207800297}{221140143764}$ =76.52%
3TB	$\frac{206158430208}{221140143764}$ =93.23%	$\frac{221140143764}{221140143764}$ =100%	$\frac{221140143764}{221140143764}$ =100%

TABLE XXII: Success Rate for 9-Character Printable ASCII Passwords when virtual columns are 30,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
100TB	$\frac{6108397932088}{21008313657487}$ =29.08%	$\frac{7853654484114}{21008313657487}$ =37.38%	$\frac{7361436776533}{21008313657487}$ =35.04%
200TB	$\frac{12216795864177}{21008313657487}$ =58.15%	$\frac{15707308968228}{21008313657487}$ =74.77%	$\frac{14691514295040}{21008313657487}$ =69.93%
300TB	$\frac{18325193796266}{21008313657487}$ =87.23%	$\frac{21008313657487}{21008313657487}$ =100%	$\frac{21008313657487}{21008313657487}$ =100%

Similarly, in the case of using all printable ASCII characters as the password character set, we observe from the evaluations that in the event of storage limitation, method 1 performs significantly better in terms of password recovery success rate compared to both the original rainbow table and method 2, consistently. Based on the results above, the improvement in success rate of method 1 over the original rainbow table can reach up to 23.60%.

TABLE XXIII: Success Rate for 7-Character Printable ASCII Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
3GB	$\frac{230087533}{698337297}$ =32.95%	$\frac{292838679}{698337297}$ =41.93%	$\frac{309549376}{698337297}$ =27.96%
5GB	$\frac{383479222}{698337297}$ =54.91%	$\frac{488064465}{698337297}$ =69.89%	$\frac{488506346}{698337297}$ =69.95%
7GB	$\frac{536870912}{698337297}$ =76.88%	$\frac{683290251}{698337297}$ =97.85%	$\frac{667463317}{698337297}$ =95.58%

TABLE XXIV: Success Rate for 8-Character Printable ASCII Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
500GB	$\frac{33554432000}{66342043129}$ =50.58%	$\frac{44739242666}{66342043129}$ =67.44%	$\frac{41349927716}{66342043129}$ =62.33%
700GB	$\frac{46976204800}{66342043129}$ =70.81%	$\frac{62634939733}{66342043129}$ =94.41%	$\frac{57869032701}{66342043129}$ =87.23%
900GB	$\frac{60397977600}{66342043129}$ =91.04%	$\frac{66342043129}{66342043129}$ =100%	$\frac{66342043129}{66342043129}$ =100%

TABLE XXV: Success Rate for 9-Character Printable ASCII Passwords when virtual columns are 100,000

Storage Limit	Original Rainbow Table	Method 1	Method 2
50TB	$\frac{3054198966044}{6302494097247}$ =48.46%	$\frac{3926827242057}{6302494097247}$ =62.31%	$\frac{3696398017280}{6302494097247}$ =58.65%
75TB	$\frac{4581298449066}{6302494097247}$ =72.69%	$\frac{5890240863085}{6302494097247}$ =93.46%	$\frac{5528917396906}{6302494097247}$ =87.73%
100TB	$\frac{6108397932088}{6302494097247}$ =96.92%	$\frac{6302494097247}{6302494097247}$ =100%	$\frac{6302494097247}{6302494097247}$ =100%

VI. Conclusions

This paper briefly describes our previous work on an enhanced rainbow table design [2] coupled with a sorting algorithm [1], which when applied, has a significant improvement over the orginal rainbow tables. Special characters are added to the storage to allow the sorting of the enhanced rainbow tables so that the password lookup time can be optimized. We further proposed 2 new storage methods to be applied with the enhanced rainbow table and showed that even with this insertion of characters to the passwords, the improvement in storage space required to store the same number of passwords reaches 28.57% lesser than what is required in the original tables in the case of alpha-numeric character set. The improvement, when the password character set consists of all the printable ASCII characters, reaches 25%. This is achieved while maintaining the same success rate.

By considering storage space limitations, we also evaluated the achieveable success rate of password recovery in different scenarios. Our analysis shows that an improvement of up to 26.13% and 23.60% can be achieved in terms of success rate, when compared to the original rainbow tables, for the alpha-numeric passwords and passwords containing any of the printable ASCII characters.

References

[1] H. M. Ying and V. L. L. Thing, “A novel rainbow table sorting method”, International Conference on Technical and Legal Aspects of the e-Society (CYBERLAWS), February 2011

[2] V. L. L. Thing and H. M. Ying, “A novel time-memory trade-off method for password recovery”, Digital Investigation, International Journal of Digital Forensics and Incident Response, Elsevier, Vol. 6, Supplement, pp. S114-S120, September 2009

[3] S. M. Smyth, “Searches of computers and computer data at the United States border: The need for a new framework following United States V. Arnold”, Journal of Law, Technology and Policy, Vol. 2009, No. 1, pp. 69-105, February 2009.

[4] Google News, “Favorite passwords: ‘1234’ and ‘password’”, http://www.google.com/hostednews/afp/article/ALeqM5jeUc6 Bblnd0M19WVQWvjS6D2puvw, [retrieved, January 2012].

[5] Cain and Abel, “Password recovery tool”, http://www.oxid.it, [retrieved, January 2012].

[6] John The Ripper, “Password cracker”, http://www.openwall.com, [retrieved, January 2012].

[7] LCPSoft, “Lcpsoft programs”, http://www.lcpsoft.com, [retrieved, January 2012].

[8] S. Contini and Y. L. Yin, “Forgery and partial key-recovery attacks on HMAC and NMAC using hash collisions”, Annual International Conference on the Theory and Application of Cryptology and Information Security (AsiaCrypt), Lecture Notes in Computer Science, Vol. 4284, pp. 37-53, 2006.

[9] P. A. Fouque, G. Leurent, and P. Q. Nguyen, “Full key-recovery attacks on HMAC/NMAC-MD4 and NMAC-MD5”, Advances in Cryptology, Lecture Notes in Computer Science, Vol. 4622, pp. 13-30, Springer, 2007.

[10] Y. Sasaki, G. Yamamoto, and K. Aoki, “Practical password recovery on an MD5 challenge and response”, Cryptology ePrint Archive, Report 2007/101, April 2008.

[11] Y. Sasaki, L. Wang, K. Ohta, and N. Kunihiro, “Security of MD5 challenge and response: Extension of APOP password recovery attack”, The Cryptographers’ Track at the RSA Conference on Topics in Cryptology, Vol. 4964, pp. 1-18, April 2008.

[12] M. E. Hellman, “A cryptanalytic time-memory trade-off”, IEEE Transactions on Information Theory, Vol. IT-26, No. 4, pp. 401-406, July 1980.

[13] D. Todorov, “Mechanics of user identification and authentication: Fundamentals of identity management”, Auerbach Publications, Taylor and Francis Group, June 2007.

[14] R. Rivest, “The MD5 message-digest algorithm”, IETF RFC 1321, April 1992.

[15] National Institute of Standards and Technology (NIST), “Secure hash standard”, Federal Information Processing Standards Publication 180-2, August 2002.

[16] H. Dobbertin, A. Bosselaers, and B. Preneel, “Ripemd-160: A strengthened version of RIPEMD”, International Workshop on Fast Software Encryption, Lecture Notes in Computer Science, Vol. 1039, pp. 71-82, Springer, April 1996.

[17] P. Oechslin, “Making a faster cryptanalytic time-memory trade-off”, Annual International Cryptology Conference (CRYPTO), Advances in Cryptography, Lecture Notes in Computer Science, Vol. 279, pp. 617-630, October 2003.

[18] A. Narayanan and V. Shmatikov, “Fast dictionary attacks on passwords using time-space tradeoff”, ACM Conference on Computer and Communications Security, pp. 364-372, 2005.

[19] D. E. R. Denning, “Cryptography and data security”, Addison-Wesley Publication, 1982.