Rainbow Tables

Yukai Zang
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
zangx040@morris.umn.edu

ABSTRACT

This paper introduces rainbow tables and shows how they are used to break a form of password protection known as a hash. Two types of rainbow tables are introduced, homogeneous and heterogeneous. Homogeneous rainbow tables were the first to be used, but heterogeneous rainbow table can be more efficient. Some details of their differences are explored. The choice of the structure of a rainbow table produces a time-memory trade-off that is also described. To make all the necessary concepts clear, a demo of an actual attack is outlined. Finally, some tests and results detailing the time-memory trade-off are explained.

Keywords

rainbow tables; chain; hash; reduction

1. BACKGROUND

Security concerns are important especially when they relate to sensitive information. It is crucial to protect this sensitive information. Normally, after a user has signed up on a website, his username and password will be sent to a server and stored in a database. Every time he tries to log in, the server will match his username and password in the database to identify him to be the account owner. However, if usernames and passwords are stored in their original form (called **plain-text**), then anybody who gains access to that file can now log in as the compromised user. Since most people tend to reuse the same pair of usernames and passwords on a variety of different websites, once such information has been hacked there is a high risk that other account being compromised. So, this type of information will not be stored as its original plain-text, instead it is more secure to store a string that can act as proof that the password for an account was correctly entered. A common approach is transform the password by applying a type of function known as a cryptographic hash function. Each cryptographic hash function maps data of arbitrary size onto data of fixed size. The string produced from a cryptographic hash function is called **hashed value**, which usually is in hexadecimal using [0-9] and [a-f]. The following are five features that a cryptographic hash function must have but a normal hash function might only have some of them:

- it is deterministic so the same original plain-text always results in the same hash;
- it is quick to compute the hash value for any given plain-text;
- it is infeasible to generate the original plain-text from its hash value except by hashing all possible plain-texts;
- a small change to the original plain-text should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value;
- it's infeasible to find two different plain-texts with the same hash value [2].

So, an ideal cryptographic hash function allows the hashing process for the password to be infeasible to invert. But, due to the first property of cryptographic hash functions, the same plain-text will correspond to the same hashed value so that an effective and commonly used way to break the hashing is making lookup tables for every possible plain-text based on the requirements. However, the size of lookup tables is dependent on the number of possible plain-texts. For example, on many websites passwords are 8 characters long where a character can be any one of the 62 alphanumeric characters (the digits [0-9], lower-case English letters [a-z], or upper-case English letters [A-Z]). This makes for $62^8 \approx 2.18 * 10^{14}$ (or about 218 trillion). In order to store these many combinations, about 2 quadrillion bytes are needed, and even working on a fast computer with an i7 processor, it still takes about 32 days to generate and write all of them [1]. Therefore, the idea of crypt-analytic time-memory trade-off (TM-TO) was introduced in 1980 by Martin Hellman [3]. The concept was improved by Philippe Oechslin in 2003, and then being known as "Rainbow tables" [1].

2. INTRODUCTION

In this paper, I will introduce some key parts of rainbow tables (Section 3), which includes offline and online stage, hash function, reduction function, chain, and collision. This section will explain each part of this technique. After that, Section 4 will discuss an improved version (called **heterogeneous rainbow tables**) compared with the original version (called **homogeneous rainbow tables**) of rainbow tables in order to perform faster in search time. In Section 5, an actual demo will be provided. A simple illustration of the attack step-by-step will be showed in this section. At the end, Section 6 will mention and explain the results of three different tests: relationship between search time and number of tables used, relationship between chain length and search time, and relationship between chain length and space needed.

3. RAINBOW TABLES

Rainbow tables are pre-computed tables for password search from a given hash value. According to the author in [5], rainbow tables with size of 1.4 GB can have 99.9% success rate in breaking

UMM CSci Senior Seminar Conference, December 2019 Morris, MN.

Windows password consisting of alphanumeric symbols. Using rainbow tables to break the hash is divided into two stages: the offline stages, and the online stage.

The offline stage (Section 3.1) is where the table(s) are computed and stored in memory. The tables must be computed prior to use and the computation will not depend upon any passwords that you want to find. Using parallel processing, multiple, non-redundant tables can be produced. The number of tables chosen will depend upon the number of processors available for the online stage. There is an extra cost associated to detecting and removing redundant information which will be discussed in Section 4.

The online stage (Section 3.2) is when the Rainbow Tables are being used to recover the original plain-text password from its hashed value. The speed of the lookup can be increased by using multiple tables and multiple processors–one for each table.

3.1 Offline stage

At this stage, rainbow tables are generated and stored in memory. In order to compute rainbow tables, there are three key components: hash functions, reduction functions, and chains. A common problem that arises when producing chains is called a collision (Section 3.1.1).

3.1.1 Collision

If a function maps two different values in one set onto the same value in another set, it is called a **collision**. It happens more frequent when the size of these two sets are significant different. The mathematical principle known as the Pigeonhole Principle [4] ensures that anytime the range of function is smaller than the domain a collision is inevitable. Therefor, when we want to map data from a larger set into a smaller set, collision will occur.

3.1.2 Hash functions

The hash function is dependent on the hash algorithm chosen: MD5, SHA-1, SHA-2, SHA-3, and BLAKE2 are some algorithms referenced relatively often. Each uses a different technique to hash the plain-text to get the hash value. Due to the different restrictions in different hash algorithms, there will be slight differences in time to generate a hash value since it takes different time to run different hash functions [5]. The hash algorithm used in building the rainbow tables must the same one that was used to hash the password that was stored on the server.

3.1.3 Reduction functions

The reduction function is a mathematical function that maps from the set of hash values back to the set of plain-texts. Different hash algorithms result in hashed values of different lengths, therefore, the reduction function used to construct the rainbow tables will also depend upon the hash algorithm used to encrypt the passwords. The set of legal passwords is much smaller than the set of hash values, for example, the size of a set of plain-texts which allows 62 alphanumeric characters for 8 digits is about $2.18 * 10^{14}$. Meanwhile, the size of a set of hashed values based on hash function MD5 is about $16^{32} \approx 3.40 * 10^{38}$, which is significantly larger than the set of plain-texts. Since a reduction function takes a hash value and reduces it to a shorter plain-text collisions are inevitable. One way to reduce the number of collisions is to use multiple reduction functions (See Section 3.1.4 for an example of how this works) [6].

3.1.4 Chains

Each entry in a rainbow table represents one chain. A chain is an alternating sequence of strings produced by applying the hash function and the reduction function one after another (See Equation 1). The length of a chain is the number of hashed values in the sequence (which is equal to the number of plain-text values in the sequence as well). This is called the **chain length**. The original version of rainbow tables used the same chain length in all the tables, and this became known as **homogeneous rainbow tables**. Author G. Avoine and X. Carpent [1] explored the consequences of using chains of differing lengths among different tables. This is known as **heterogeneous rainbow tables** (Section 4). With the pre-set chain length n, a single chain starts at t_0 (which is called starting point S_0), an arbitrary plain-text, and we apply hash function h and a reduction function r_0 in order to get t_1. Repeat this step n times, and hash the final answer to get h_n (which is called ending point E_0). Only S_0 and E_0 are stored in the rainbow table.

$$S_0 = t_0 \xrightarrow{h} h_0 \xrightarrow{r_0} t_1 \xrightarrow{h} h_1 \xrightarrow{r_1} \dots \xrightarrow{h} h_n = E_0 \tag{1}$$

Although only the starting point and ending point of a chain (one plain-text value and one hashed value) are stored, the other values in the chain can still be considered to exist as virtual **columns**. Column 0 contains the initial plain-text (such as t_0 in the example above), Column 1 would contain the hashed values of Column 0, Column 2 would be the **reduced values** of Column 1, etc.

3.1.5 Table generation

In order to perform a faster generation and computation, tables' generation are done in parallel among processors. Each processor generates its own table. So there will be l tables generated in total when l processors are used. During the generation, the same reduction function will be applied to the same column for all chains in the same table (See Table 1). For example, in one table, the reduction function used between Column 1 and Column 2 would be the same (it might be different for different tables). However, in order to lower the frequency of collisions, different reduction functions are used among different columns. For example, the reduction function used between Column 3 and Column 4 is different from the one used between Column 1 and Column 2.

It is possible for two entries in the same column to end up with the same value. This kind of **collision** will usually occur in a plain-text column. After such a collision, the remainder of the respective chains will match perfectly and the ending points of those two chains will be the same. In this circumstance, one of the two chains with the duplicated endpoint will be removed from the table. Note that the longer the chain, the more opportunities for collisions there are–so the more entries will likely need to be removed. There might be duplicated chains among different tables, which is acceptable.

The generation usually stops when the number of different ending points m is deemed satisfactory [1]. After that the table will be called a **clean table** [1]. On the other hand, we are removing information from the table and it is now possible for a password to no longer be able to be successfully looked up. This is where using multiple tables with different reduction functions is important. On any given table it is a near certainty that collision will occur and information will be removed from the table, but if the reduction functions are different between the tables then the chances are less likely that the same password information will be removed from all tables. As the number of tables goes up so too does the probability of successfully finding a password on at least one of the tables. If a table contains all or almost all possible ending points, which happens when adding any new chain with any possible starting point would have a high probability to cause a collision, it is called **maximal size table** [1]. The following is the relationship between the probability of success and the number of

Table 1: Structure of a rainbow table [1]

$S_0 = t_{0,0}$	$\xrightarrow{h}$	$h_{0,0}$	$\xrightarrow{r_0 \circ h}$	$h_{0,1}$	$\xrightarrow{r_1 \circ h}$	$\dots$	$\xrightarrow{r_{n-2} \circ h}$	$h_{0,n-1}$	$\xrightarrow{r_{n-1} \circ h}$	$h_{0,n} = E_0$
$S_1 = t_{1,0}$	$\xrightarrow{h}$	$h_{1,0}$	$\xrightarrow{r_0 \circ h}$	$h_{1,1}$	$\xrightarrow{r_1 \circ h}$	$\dots$	$\xrightarrow{r_{n-2} \circ h}$	$h_{1,n-1}$	$\xrightarrow{r_{n-1} \circ h}$	$h_{1,n} = E_1$
$\vdots$		$\vdots$		$\vdots$		$\vdots$		$\vdots$		$\vdots$
$S_j = t_{j,0}$	$\xrightarrow{h}$	$h_{j,0}$	$\xrightarrow{r_0 \circ h}$	$h_{j,1}$	$\xrightarrow{r_1 \circ h}$	$\dots$	$\xrightarrow{r_{n-2} \circ h}$	$h_{j,n-1}$	$\xrightarrow{r_{n-1} \circ h}$	$h_{j,n} = E_j$
$\vdots$		$\vdots$		$\vdots$		$\vdots$		$\vdots$		$\vdots$
$S_m = t_{m,0}$	$\xrightarrow{h}$	$h_{m,0}$	$\xrightarrow{r_0 \circ h}$	$h_{m,1}$	$\xrightarrow{r_1 \circ h}$	$\dots$	$\xrightarrow{r_{n-2} \circ h}$	$h_{m,n-1}$	$\xrightarrow{r_{n-1} \circ h}$	$h_{m,n} = E_m$

clean rainbow tables of maximal size [1]:

$$P^* \approx 1 - e^{-2l}.$$

The author, based on this result, indicates that a value of $l = 4$ tables will obtain a total success rate of 99.97% [1].

3.2 Online stage

At this stage, Rainbow tables are loaded in memory and searched to find a possible match for the given hash value. In Table 1 [1], S_j is the j^{th} chain in table, $t_{j,x}$ is the j^{th} chain in the table, and x^{th} plain-text in a single chain. $h_{j,n}$ is the ending point of j^{th} chain, which is also represented as E_j. The steps provided by G. Avoine and X. Carpent [1] are as follows: according to the given hash value $y = h(x)$, search through the column of ending points in all tables; if such a j that $E_j = y$ is not found, compute $h(r_{n-1}(y))$ and search through the column of ending points to find a match j satisfying $E_j = h(r_{n-1}(y))$; if still no match, compute $h(r_{n-1}(h(r_{n-2}(y))))$ and so on until a match j is found or all the columns in all tables are searched; if we find a match, we rebuild a chain based on the corresponding staring point S_j; we stop when we get $t_{j,x}$ which has the property $h(t_{j,x}) = y$. An example of this process is in Section 5.2.

4. HETEROGENEOUS RAINBOW TABLES

Author G. Avoine and X. Carpent [1] suggested an optimal structure of rainbow tables. Compared with the original version, it allows different chain length in order to improve the performance, especially on search time. This setup is called **heterogeneous rainbow table**. Authors also indicated these tables should be clean with maximal size since it is the most memory-efficient to tables satisfying these two criteria (See detail in [1]). Similar to the procedure for homogeneous rainbow tables, the technique for using heterogeneous rainbow tables is still divided into two stages that have similar roles: offline stage and online stage. The important differences that will be discussed in this section are for: the chain length in the offline stage, the order of visit in the online stage, and the parameter set-up.

4.1 Table generation

In the offline stage, table generation is the main part of this stage. While, the same chain length among all different tables in homogeneous rainbow tables, G. Avoine and X. Carpent [1] showed that chain length n of each table should be individually computed. Meanwhile, since the tables are clean and of maximal size, the number of chains m in a specific table, size of inputs N, and chain length n in that table are related as following [1]:

$$m = \frac{2N}{n+1}$$

4.2 Order of visit

In heterogeneous rainbow tables, the chains and tables visiting order is quite different from in homogeneous rainbow tables. Different chain lengths among different tables is the main issue. In homogeneous rainbow tables, the visiting order is paralleled among each table. Since the chain length and number of chains are identical to each other, we could promise to search through tables entirely at the same time without worrying there is no entry for any table. However, tables of different chain length would have this problem. If we do the same parallel searching, we might end up with only one search involving a few tables with longer chain length but do nothing to tables with short chain length. Also, we might only search a few tables with more chains but do nothing to tables with fewer chains. Either of these two situations are a huge waste of time. G. Avoine and X. Carpent suggested that we should start a search from the table with shortest and most chains [1]. After each search step, a decision should be made based on a metric for each tables, which is computed from the ratio of the probability to find a solution over the average amount of work. The following is the metric for the i^{th} steps in k^{th} table:

$$\eta(i,k) = \frac{P(\text{x found at the } i^{th} \text{ step in table } k)}{E(\text{work for the } i^{th} \text{ step in table } k)},$$

with x to be the desired answer.

The decision is to choose the metric with the highest value in order to decide which table the next search will be done. Author had proved when we use this technique, the search time could be minimized (See detail in [1]).

4.3 Parameter set-up

In heterogeneous rainbow tables, there are five parameters:

- number of tables l,
- size of inputs N,
- number of chains $\{[m]_1, \dots, [m]_l\}$,
- chain lengths $\{[n]_1, \dots, [n]_l\}$,
- memory needed to store each table $\{[M]_1, \dots, [M]_l\}$

For these parameters, the number of tables l is decided in Section 3.1.5. The size of inputs is fixed by the requirements on plaintext. The number of chains $\{[m]_1, \dots, [m]_l\}$ is defined in Section 4.1. Memory of each tables should sum to a fixed value M, in Section 6, in order to compare the search time between homogeneous and heterogeneous rainbow tables, author use the size needed for homogeneous rainbow tables to be the total size M in heterogeneous rainbow tables set-up. Besides, author used an optimization function to compute the chain length $\{[n]_1, \dots, [n]_l\}$ for different tables according to the fixed total size M (See detail in [1]).

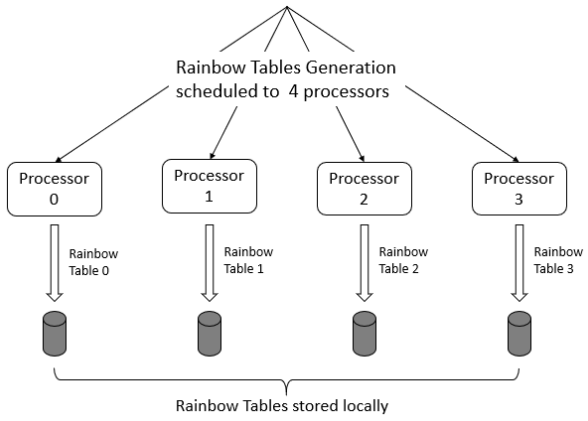

Figure 1: Structure of parallel computation [6]

5. ATTACK DEMO

In this section, an attack demo based on the normal rainbow tables technique (homogeneous rainbow tables)will be present [6]. For easy illustration in this demo, the set of plain-text allows 4 characters with [a-z] texts; there are $l = 4$ processors used; $m = 1000$ chains would be produced in each table; the chain length is $n = 2$. Each reduction function r_i is distinct from the rest. Table 1 defines the symbolism we'll use to refer to the contents of one of the l rainbow tables. Table 2 is a fake rainbow table.

5.1 Offline stage

At this stage, the set-up and generation of the tables would be done, which would have the following parts: computation set-up, chain set-up, and chain storage.

Computation set-up: With $l = 4$ processors, generate four rainbow tables, one per processor. Each table will be stored in processor locally (See Figure 1).

Chain set-up: In order to know how to produce a chain two types of functions need to be determined in advance, the *hash function* and the *reduction functions*. In our demo we will use MD5 as the hash function. A family of reduction functions can be generated using a method known as **index shifting** [5].

The hash value is a sequence of hexadecimal values represented in ASCII. Each reduction function takes 4 bytes, which are represented by 8 hexadecimal digits. Each byte is reduced modulus 26. The resulting value is converted to ASCII using $0 \mapsto$ a, $1 \mapsto$ b, $\ldots$, $25 \mapsto$ z. Each reduction function uses a different set of four characters. Imagine a sliding-window that is advanced by 1 character each iteration. This is the *index shift*. We keep shifting the starting index as we generate the chain (See the example below).

Assume we start with the following hash:

`74b87337454200d4d33f80c4663dc5e5`

In order to produce the next mapped plain-text, we compute:

$$\texttt{|74b87337|}454200d4d33f80c4663dc5e5 \xrightarrow{r_0} mcld$$

Notice the sequence `74b87337` at the beginning of the MD5 hash. Here is how that is broken into bytes and reduced to a 4 character plain-text sequence:

$$\begin{aligned}
(74)_{16} &= (116)_{10} \xrightarrow{\text{mod } 26} 12 \rightarrow m \\
(b8)_{16} &= (184)_{10} \xrightarrow{\text{mod } 26} 2 \rightarrow c \\
(73)_{16} &= (115)_{10} \xrightarrow{\text{mod } 26} 11 \rightarrow l \\
(37)_{16} &= (55)_{10} \xrightarrow{\text{mod } 26} 3 \rightarrow d
\end{aligned}$$

This is how to get the output $h \xrightarrow{r_0} t$.

Notice that the same hash-value produces a different plain-text value when we use a different reduction function:

$$\texttt{7|4b873374|}54200d4d33f80c4663dc5e5 \xrightarrow{r_1} xfzm$$

$$\begin{aligned}
(4b)_{16} &= (75)_{10} \xrightarrow{\text{mod } 26} 23 \rightarrow x \\
(87)_{16} &= (135)_{10} \xrightarrow{\text{mod } 26} 5 \rightarrow f \\
(33)_{16} &= (51)_{10} \xrightarrow{\text{mod } 26} 25 \rightarrow z \\
(74)_{16} &= (116)_{10} \xrightarrow{\text{mod } 26} 12 \rightarrow m
\end{aligned}$$

This is how to get the output $h \xrightarrow{r_1} t$.

As we can see, even with the same hashed value, as long as we are at different part of the chain, which means we are using different reduction functions (i.e. r_0 and r_1), we could produce different plain-texts so that collisions can be reduced. However, this method is still very weak in eliminating collisions since only a few parts of the string has been used. We could even create our own reduction functions to be used in generation. But the rule of using the same reduction function at the same column of all chains in the same table has to be followed, otherwise, there is no way we could provide the search steps outlined in Section 3.2.

Chain storage: After we follow the steps in section 3.1.5, pairs of S_j and E_j will be sorted with the respect to the ending points in each tables in order to have a minimal search time with $O(\log_2(n))$, which is binary search.

5.2 Online stage

At this stage, we would based on the given hashed value to find its corresponding plain-text via rainbow tables generated and stored in each processor in the offline stage. Computation set-up would still be explained, and a simple search process with the fake rainbow table (See Table 2) will be illustrated.

Computation set-up: The search will be executed in parallel among the l processors, like how the tables were generated in the offline stage (similar to the structure showed in Figure 1). For easy illustration, a simple single rainbow table is provided as Table 2.

Text	Hash
aaaa	`9056bcf1112722e02e4379d1a0287c4`
aaab	`4c189b020ceb022e0ecc42482802e2b8`
aaac	`3963a2ba65ac8eb1c6e2140460031925`
aaad	`aa836f154f3bf01eed8df286afbb388`
⋮	⋮

Table 2: A fake rainbow table

Search: With chain length $n = 2$, the structure of a single chain will be:

$$\text{Chain}: t_0 \xrightarrow{h} h_0 \xrightarrow{r_0} t_1 \xrightarrow{h} h_1 \xrightarrow{r_1} t_2 \xrightarrow{h} h_2$$

A search will be performed based on the following hash (which is not presented in the table):

`74b87337454200d4d33f80c4663dc5e5`

Assume this hash is mapped from x via MD5. In our demo, there will be three different cases for x to be located on: t_0, t_1, or t_2.

1. Search through the ending points list, and there is no match (in this case, we assume that $x = t_2$)

2. Compute $h(r_1(h(x)))$=2b740a74b4de122dad1317e874ee9711 and search through the ending points list. There is still no match (in this case, we assume that $x = t_1$)

3. Compute $h(r_1(h(r_0(h(x)))))$=9056bcf11127242e02e4379d1a0287c4 and search through the ending points list. A match is found.

4. Since a match is found, a chain is built from the corresponding starting point *aaaa*. The generation stops when our desired plain-text x is produced:

$$\begin{aligned} \textbf{\textit{aaaa}} &\xrightarrow{h} \texttt{74b87337454200d4d33f80c4663dc5e5} \xrightarrow{r_0} \\ \textit{mcld} &\xrightarrow{h} \texttt{297c9f80e9ad49f832e029840b39534a} \xrightarrow{r_1} \\ \textit{vloo} &\xrightarrow{h} \texttt{9056bcf11127242e02e4379d1a0287c4} \end{aligned}$$

5. Our desired plain-text is "*aaaa*"

In real life attacks, it is possibility that we could not find any match by searching through all the tables we have since we are not using the maximal-size tables, which means there is a chance that the given value has not been covered by our rainbow tables. In this case, the only way to solve it is try to make all tables with maximal-size, which would cost more time to generate but has almost 100% coverage on all possible plain-texts.

6. TESTS AND RESULTS

Author G. Avoine and X. Carpent calculated the average number of computation needed for both homogeneous and heterogeneous rainbow tables in order to indicate the average search time needed. They used $N = 2^{40}$ and the same way to set-up all parameters mentioned in Section 4.3. They compared the average improvement in search time with different number of tables l are used [1] (See Figure 2).

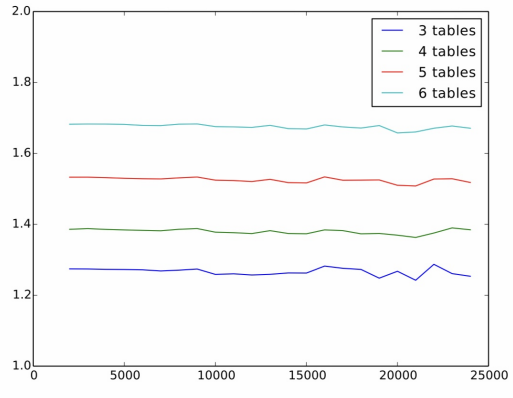

Figure 2: Average search time improvement using heterogeneous table over using homogeneous tables [1]

In Figure 2, the lower the line is, the few tables are used. From the results, we could clearly observe the difference among different number of tables are used. The more tables we generated, the more improvement we get from using heterogeneous rainbow tables compared with homogeneous ones, which indicates that heterogeneous rainbow tables do perform a faster crypt-analyses, like suggested in [1].

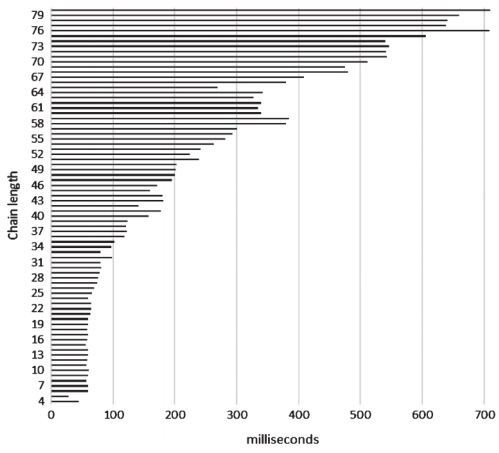

Figure 3: Dependence of chain length with average search time for a password [5]

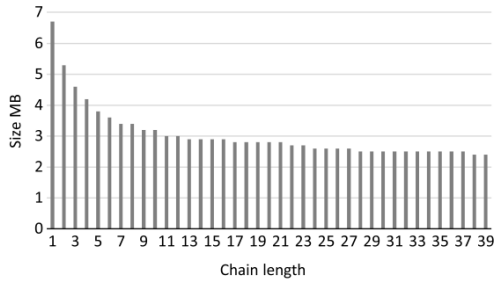

Figure 4: Dependence of chain size with the length of rainbow tables [5]

On the other hand, some authors did researchers on changing the value of m would affect the search time and the storage space on homogeneous configuration with fixed number tables [5]. They performed two specific tests. The first one is the dependence of chain length with average search time, see Figure 3. From this result, it is clear to observe an increasing in search time in milliseconds as the chain length increases. For search process in Section 3.2, finding a match from all ending points in all tables has efficiency $O(\log_2(n))$ since we sort all tables with the respect to hash values in Section 5.1, and there are n columns to search through, which has a time complexity of $O(n \log_2(n))$. The result from first test proves this complexity clearly.

The second test is focused on the relationship between chain length and the size of rainbow tables, see Figure 4. In Figure 4, chains are generated form the whole set of plain-texts with different chain length. From this figure, we are confident to conclude that as the chain length grows, the size of rainbow tables shrinks. Based on Section 3.1.5, the collisions will be handled, and a clean table will be produced, and the longer your chain are, the more likely you will have collisions. This indicates as the chain length goes up, the total size of homogeneous rainbow tables will decrease.

7. CONCLUSIONS

From the results in Section 6, first of all, we conclude that heterogeneous rainbow tables have a better performance on search time compared with homogeneous rainbow tables with the same memory used. Then, under homogeneous configuration, shorter chain length leads to less search time. However, the shorter chains need more memory to store. On the other hand, longer chain length requires less space to store but a longer searching time to search through longer chains. This is a typical time-memory trade-off model. There are several ways to optimize the model. One could use heterogeneous tables instead of homogeneous tables or one could just use more tables. When space is limited, a longer chain might be a great choice. Otherwise, if it takes a long time to perform the attack, a shorter chain length that produces tables requiring more storage space is probably better.

Acknowledgments

Extremely thanks to Professor Peter Dolan, Professor K.K. Lamberty, and Professor Elena Machkasova! Also thanks for everyone's help!

8. REFERENCES

[1] G. Avoine and X. Carpent. Heterogeneous rainbow table widths provide faster cryptanalyses. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, pages 815–822, New York, NY, USA, 2017. ACM.

[2] Cryptographic hash function. Cryptographic hash function — Wikipedia, the free encyclopedia, 2019. [Online; accessed 15-March-2019].

[3] M. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transactions on Information Theory*, 26(4):401–406, July 1980.

[4] I. N. Herstein. *Topics in algebra*. Blaisdell Publishing Company, 1964.

[5] J. Horáleka, F. Holík, O. Horák, L. Petr, and V. Sobeslav. Analysis of the use of rainbow tables to break hash. *Journal of Intelligent & Fuzzy Systems*, 32(2):1523 – 1534, 2017.

[6] D. D. Mishra, C. S. R. C. Murthy, K. Bhatt, A. K. Bhattacharjee, and R. S. Mundada. Development and performance analysis of hpc based framework for cryptanalytic attacks. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 789–794, New York, NY, USA, 2012. ACM.