**Matrices Applied in Cryptography**

**Team 13**

1. **Abstract**

Many people use the same passwords for numerous accounts, which is very dangerous. Thus, we created an encoding program to solve this problem, and used some quantitative methods to evaluate our results.

1. **Introduction**

Nowadays, most people have multiple accounts, yet with all passwords the same, because creating a new password is quite annoying and will be easily forgotten. However, using the same password across multiple accounts can lead to credential stuffing, which is unsafe. As a result, we came up with an idea to write a code that can convert our commonly used passwords into different passwords with higher security corresponding to the account’s name.

1. **Methods**

The main steps are as below:

1. Enter web name and general password users usually use
2. Encryption using matrix multiplication
3. Then we can get a new password!

(Detailed steps can be found in [Appendix A](#_tn1z5q2xdbg6))

1. **Results**

We collected some passwords (see [Appendix B](#_623zkk9fbqvf)) and used the UIC Password Strength Test to test if our encryption is effective. We divided the strength into four classes: very strong, strong, good, weak, and very weak. Our most successful batch of encrypted passwords showed that for the original passwords, 81.8% were very weak and 18.2% were weak. But after encryption, only 3% were very weak and 36.4% became very strong. In addition, we calculated the increment of password complexity. For instance, if a very weak password becomes a very strong one, it increases four classes, denoted as +4. We learned that the average increment of password complexity is +2.5149. (There are more detailed graphs, charts, and other two examples in [Appendix D](#_qc7doznlw5u8).) By the results we obtained from the strength test, we can prove that our encryption is really useful.

**Discussion**

The program that we used to convert the original passwords into more complex, thus stronger ones was successful due to the program using a randomly generated matrix to encrypt the passwords. The new generated passwords are random as well, which leads to lower possibilities of characters in predictable positions, common keyboard patterns, common words, and common phrases. Therefore, these new passwords are more secure than before. (The reason that the more complex a password is the more secure it is can be answered in [Appendix F](#_m384pj26u0fo).)

**Conclusion**

The majority of people tend to overuse passwords that are simple and easily guessed. With the implementation of our program which uses matrices to encrypt said passwords, we are able to make the passwords more complex and harder to guess.

**References**

1. Ur, B., Alfieri, F., Aung, M., Bauer, L., Christin, N., Colnago, J., ... & Melicher, W. (2017, May). Design and evaluation of a data-driven password meter. In Proceedings of the 2017 chi conference on human factors in computing systems (pp. 3775-3786).
2. Password Meter - A visual assessment of password strengths and weaknesses. (n.d.). https://www.uic.edu/apps/strong-password/
3. Dell'Amico, M., Michiardi, P., & Roudier, Y. (2010, March). Password strength: An empirical analysis. In 2010 Proceedings IEEE INFOCOM (pp. 1-9). IEEE.
4. Wheeler, D. L. (2016). zxcvbn:{Low-Budget} Password Strength Estimation. In 25th USENIX Security Symposium (USENIX Security 16) (pp. 157-173).
5. Kelley, P. G., Komanduri, S., Mazurek, M. L., Shay, R., Vidas, T., Bauer, L., ... & Lopez, J. (2012, May). Guess again (and again and again): Measuring password strength by simulating password-cracking algorithms. In 2012 IEEE symposium on security and privacy (pp. 523-537). IEEE.
6. Yan, J., Blackwell, A., Anderson, R., & Grant, A. (2004). Password memorability and security: Empirical results. IEEE Security & privacy, 2(5), 25-31.

**Table of Contents for Appendices**

1. [Methods](#_tn1z5q2xdbg6)
   1. Step 1 to Step 8
   2. Special Advantage
2. [Survey](#_623zkk9fbqvf)
   1. How has the survey been done?
   2. Rules of password generation
3. [The UIC Password Strength Test](#_fpow708r303k)
   1. About the Developer
   2. How the test works
4. [Detailed Results from Strength Test](#_qc7doznlw5u8)
   1. Social Account
   2. E-Mail
   3. Phone
5. [Validation](#_q59pjfdbqmo5)
   1. Background of the research paper
   2. What we used in the research paper for validation?
6. [Other Attempted Evaluation Methods](#_m384pj26u0fo)
   1. Cain and Abel
      1. Brute-force attack
      2. Dictionary attack
      3. Rainbow Table attack
7. [Source Code](#_hq2036h7pdb)
8. [Author Contributions](#_87kwytfn8h4o)

### Methods

Here below are the methods used to encrypt the passwords input into the program.

**Step 1:**

We created a special sentence by hand to build the web key. The web key is used to generate the matrix A, which is going to be used to encode the general password later.

**Step 2:**

The user’s general web account username and password is to be entered.

**Step 3:**

The username will correspond to a group of numbers in the web key.

**Step 4:**

We use that group of numbers as a seed of the random function, and generate a random number *w*. Then, our number of rows of matrix A = (*w* % length of password) + length of password (while number of columns of matrix A = length of password). This can ensure that the matrix A is not invertible, because the number of rows doesn’t equal the number of columns.

**Step 5:**

Generate encryption matrix and use the web key to fill it.

(according to step 4, the matrix is non-square so it is not invertible at the same time)

**Step 6:**

Convert the general password from type of *char* to type of *int* by using ASCII code, and transpose it in order to do the matrix multiplication in the next step.

**Step 7:**

Do matrix multiplication of the encrypted matrix generated in Step 5 and the number string generated in Step 6.

**Step 8:**

Remove characters that cannot be used as a password in the string generated in Step 7 to get the final new password.

**Special Advantage:**

The random number *w* will be different on each computer. But the number will be consistent on the same computer. Thus, we can ensure that we are able to get the same encoded password and this encoded password will not be known by others even if they have our general password.

### Survey

We conducted a survey (Figure 1.) to collect some manually generated passwords.

1. How has the survey been done?

We asked each respondent to provide three passwords. One of them is for social media accounts, another is for phone unlocking, the other is for email.

1. Rules of password generation
   1. Password length is preset to popular regulations. (Facebook for social media accounts, Android for phone unlocking, and Gmail for emails)
   2. Arabic numerals, upper/lower case letters, and special characters (e.g., #, %, $, @) are acceptable.

After collecting passwords from the total 34 respondents (one was eliminated due to rule violation), we generated new passwords using our proposed method and evaluated the security of passwords both before and after encryption. Figure 2. shows the original passwords and the encrypted ones.

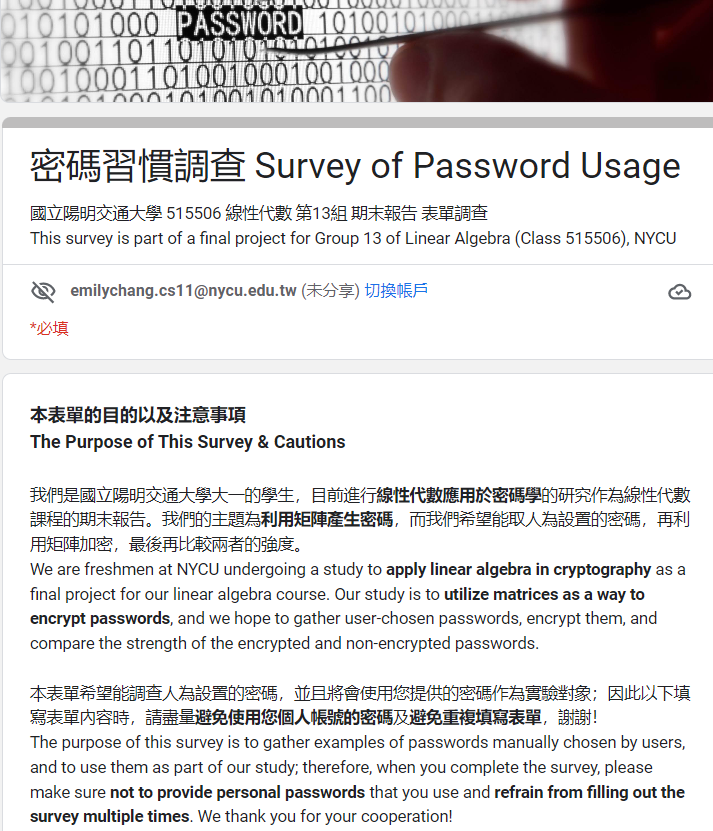


Figure 1.



Figure 2.

### The UIC Password Strength Test

1. About the Developer:

The UIC Password Strength Test is developed by the Academic Computing and Communications Center at University of Illinois at Chicago.

1. How the test works:

The test assigns points for addition and deduction to evaluate the complexity of a hypothetical password. The individual points are then added up to a total score, which is presented by a bar on top of the screen (as shown below in Fig 3).

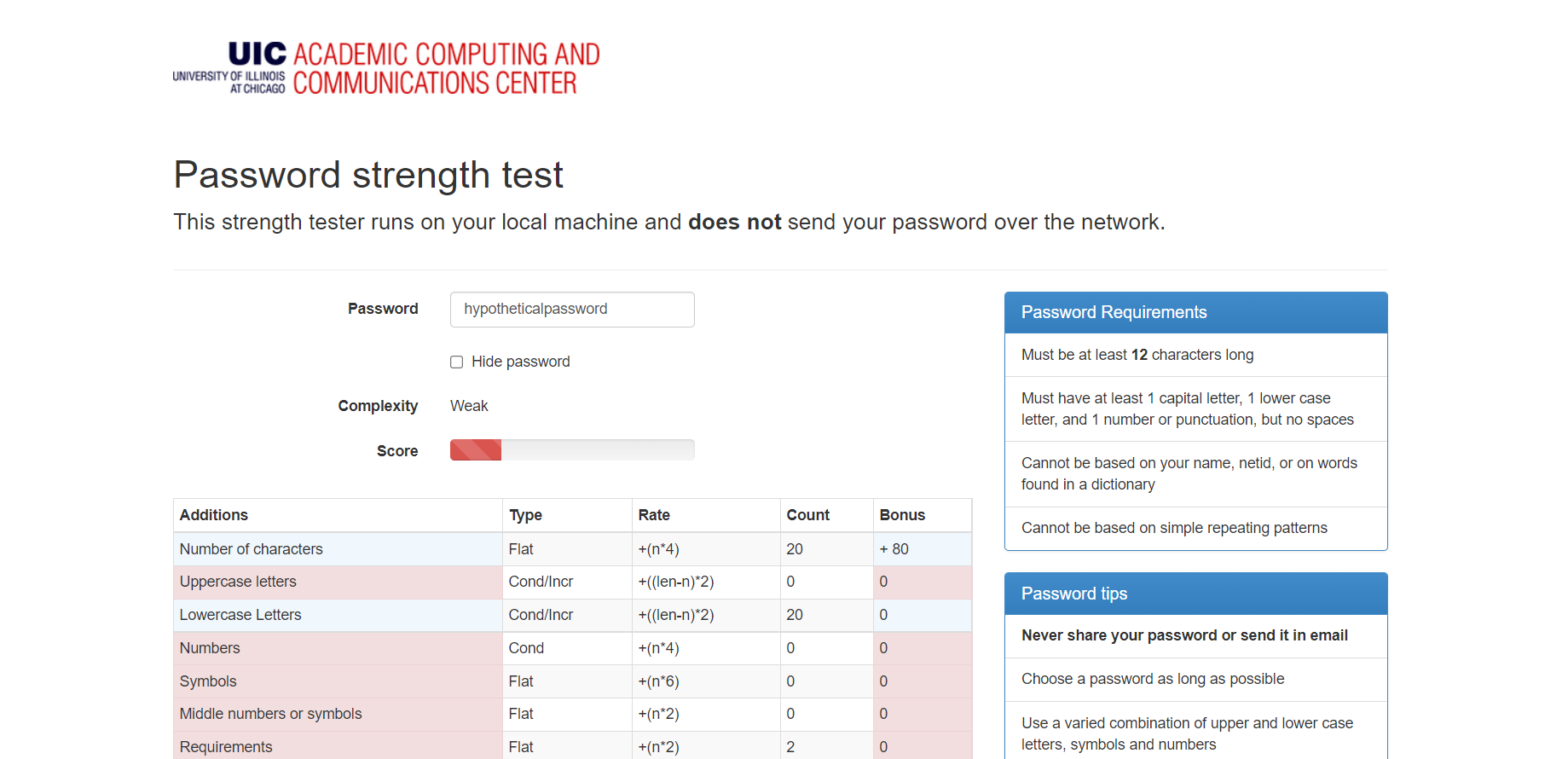


Fig 3.

The horizontal row at the top of the graph for evaluation in Fig 3 stands for how the points are added to/deducted from the overall score. *Type* in the second column corresponds to *Rate* in the third column, which indicates how the points are added/deducted and the used formula. *Count* in the fourth column stands for the occurrences for each criteria met (which are listed in the first column).

1. Points for Addition:
2. Number of Characters

The numbers of characters are counted, and for each character, a point is added to the overall score.

1. Uppercase/Lowercase Letters

For each uppercase/lowercase letter, the points increase by twice the total amount of points minus the number of occurrences of uppercase/lowercase letters.

1. Numbers

For each number, points are added by four times the amount of numbers used.

1. Symbols

*Symbols* refers to special characters such as the $, %, &, \*, etc. For each of these symbols, points are added by six times the amount of symbols used.

1. Middle Number or Symbols

For each number or symbol used in the middle of a password, the points are added by two times the occurrence of such use. (For clarification, a number or symbol used in the middle of a password means that the number or symbol is not the beginning character nor the ending character.)

1. Requirements

For each met requirement in the list in the top right corner of Fig 3, the points are added twice for each requirement met.

1. Points for Deduction:
   1. Letters/Numbers only  
      If a password is comprised of either only letters or only numbers, the test deducts a point for each letter/number used.
   2. Repeat Characters (case insensitive)  
      If a character is repeated, such as *abca*(*a* being the repeated character in this case), points are deducted. As this criteria is case insensitive, lowercase and uppercase letters are treated as the same letter, such as *A* and *a* in the hypothetical password *abcA*. However, how many points this particular criteria deducts is unknown to us, as it is not shown. Although it states that the method of deducting can be found in the source code, we are unable to access it.
   3. Consecutive Uppercase Letters/Lowercase Letters/Numbers  
      For each instance of uppercase/lowercase letters or numbers used consecutively, points are deducted twice for each occurrence. An example of a password with consecutive uppercase letters would be *ZXcvbn*, where Z and X form an instance of consecutive uppercase letters.
   4. Sequential Numbers/Letters/Symbols (used in a sequence longer than three characters)  
      For each case of sequential characters used in a password, three times the occurrences of the mentioned case of points would be deducted. An example of this would be a hypothetical password of *password123*, where 3 points would be deducted due to the usage of *123*.

### Detailed Results from Strength Test

1. Social Account:
2. Non-encrypted:

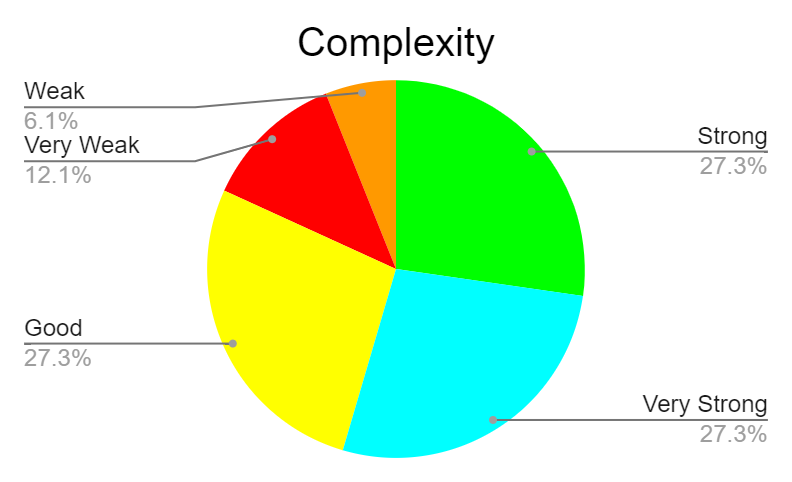


Figure 4.

1. Encrypted:

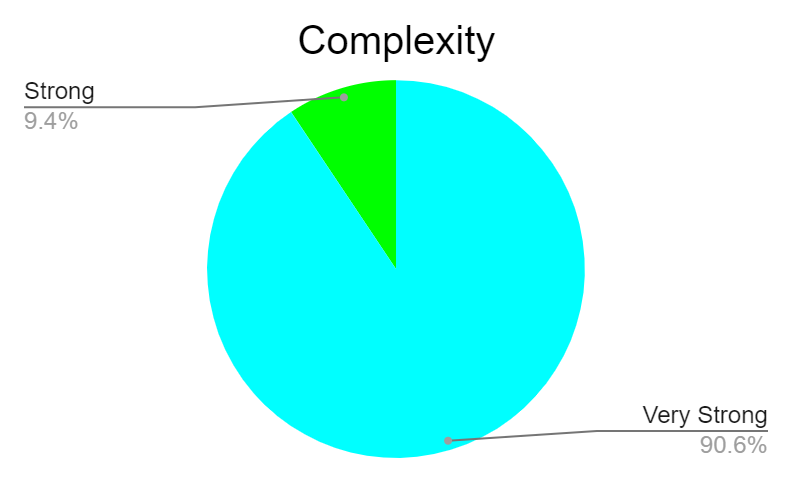


Figure5.

1. Average increment of password complexity:

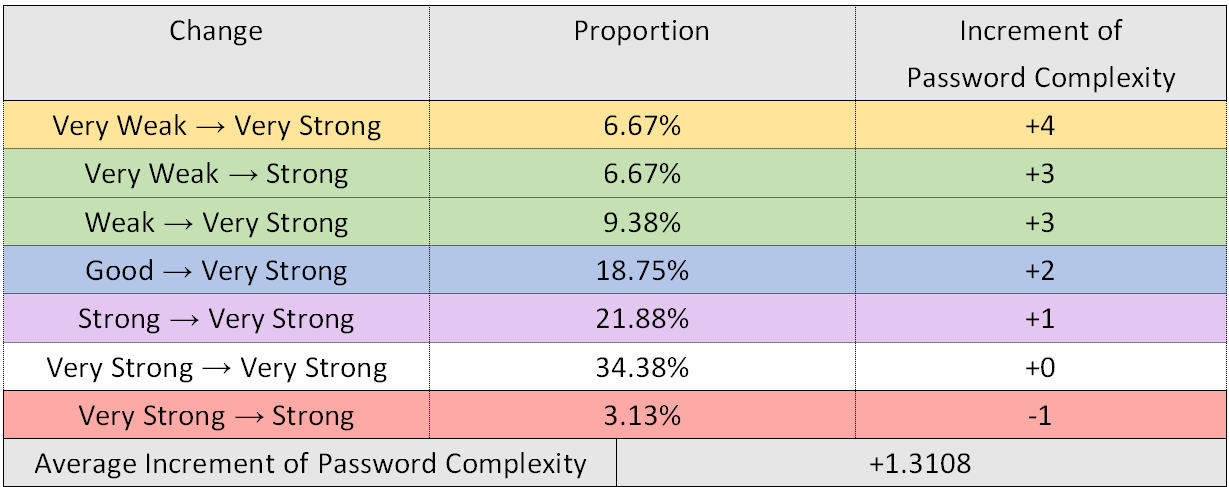


Table 1.

1. E-Mail:
2. Non-encrypted:

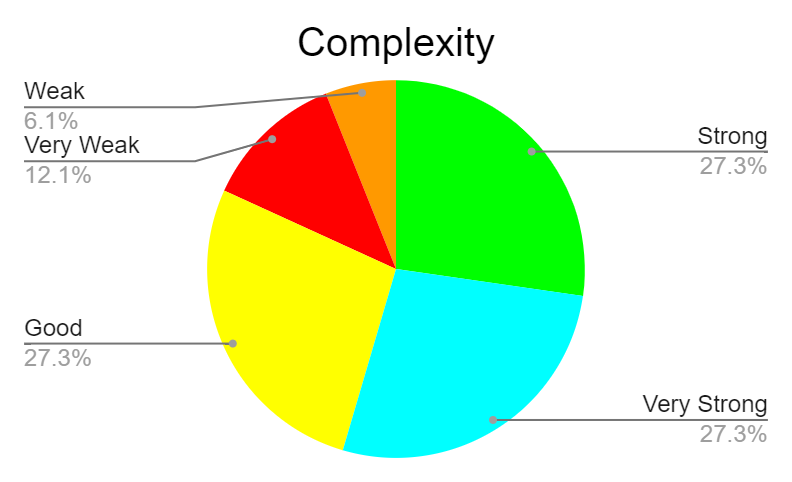


Figure 6.

1. Encrypted:

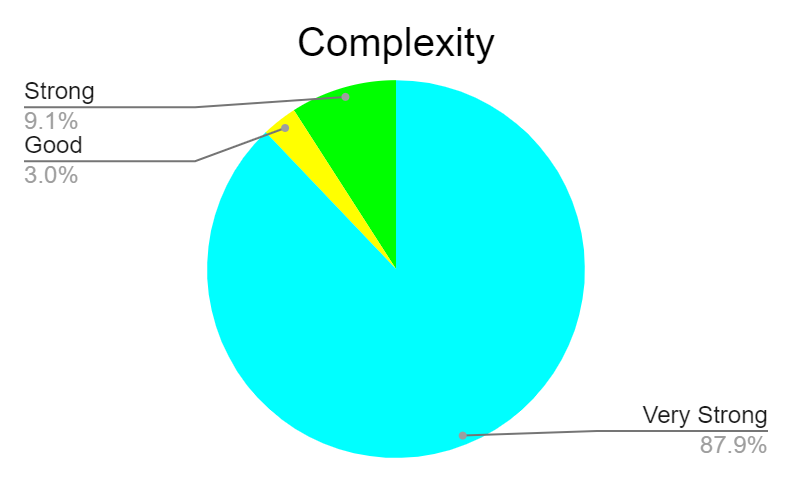


Figure 7.

1. Average increment of password complexity:

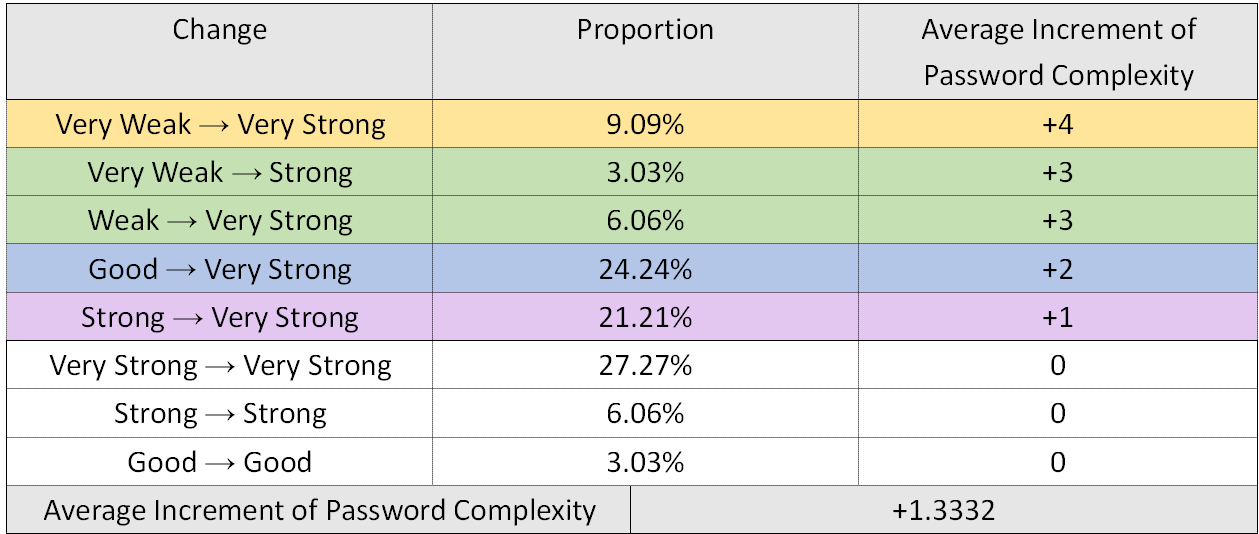


Table 2.

1. Phone:
2. Non-encrypted:

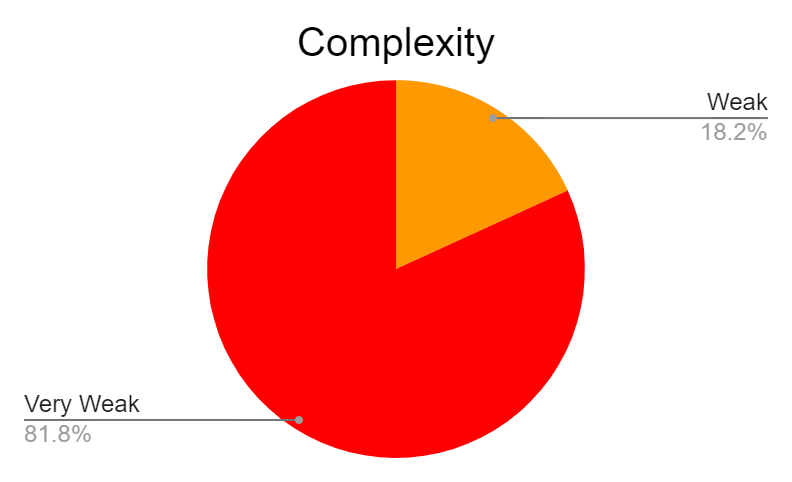


Figure 8.

1. Encrypted:

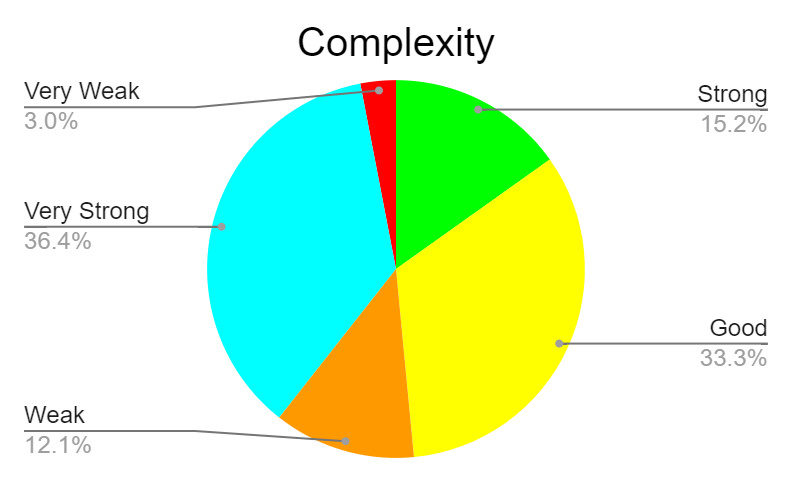


Figure 9.

1. Average increment of password complexity:

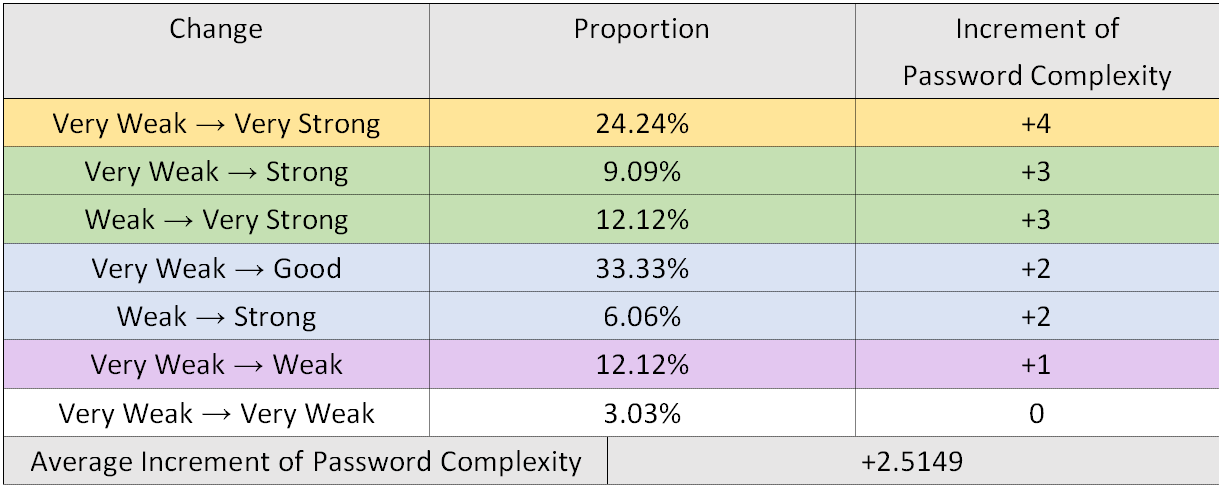


Table 3.

### Validation

We used a research paper (Figure 10.) of CyLab Usable Privacy and Security Laboratory from Carnegie Mellon University to validate our proposed methods.

1. Background of the Research Paper:
   1. Title: Design and Evaluation of a Data-Driven Password Meter
   2. Authors: CyLab Usable Privacy and Security Laboratory from CMU
   3. Published: 02 May 2017
2. What we used in the research paper for validation?

The authors mentioned the following four points to illustrate why manually generated passwords are vulnerable. Users sometimes make predictable passwords.

1. Manually-generated passwords are easily related to words and phrases
2. specific characters are in predictable locations
3. Some users use keyboard patterns like “asdfghjkl” as their passwords
4. Users frequently reuse passwords

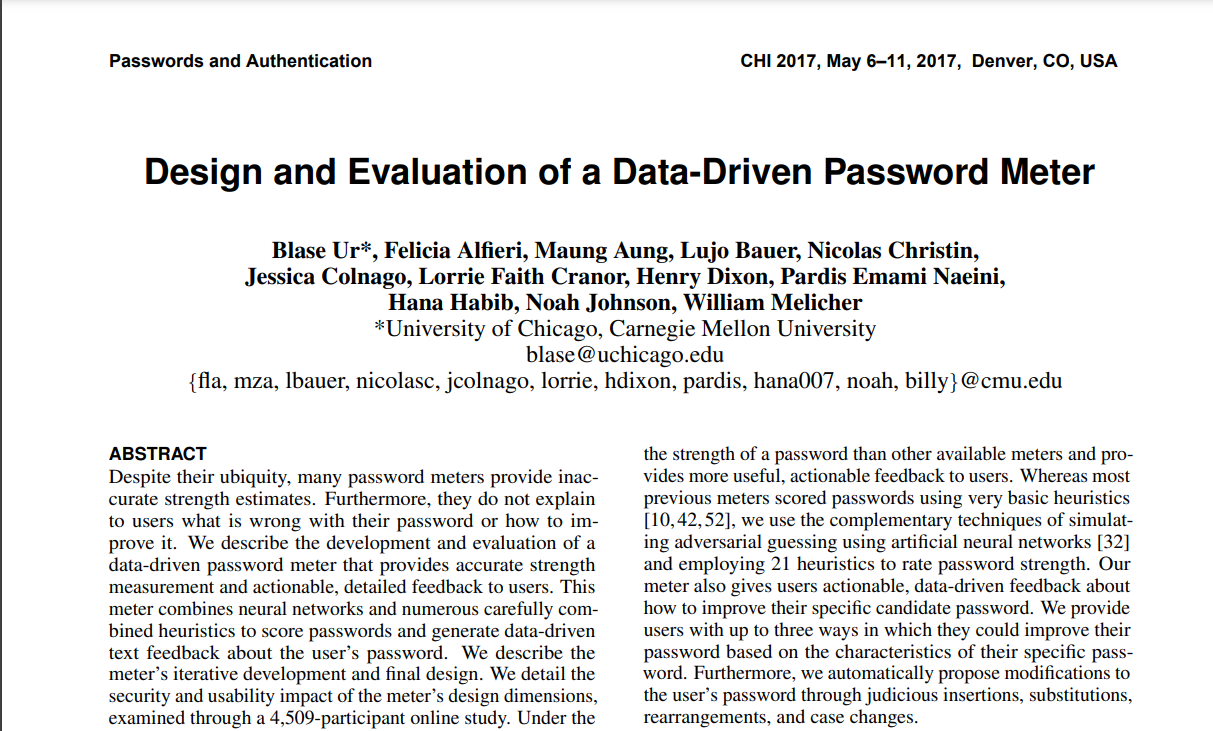


Figure 10.

### Other Attempted Evaluation Methods

1. **Cain and Abel**

This is a tool used for cracking passwords. It provides several features to crack passwords. At first, we focused on three password-cracking methods, which are the brute-force attack, dictionary attack, and the Rainbow Table Attack. We wanted to use these methods to crack the passwords both before and after encryption and measure the time taken.

1. **Brute-force Attack**

A brute-force attack uses trial-and-error and to work through all possible combinations in order to crack passwords. However, our passwords consist of upper and lowercase letters, Arabic numerals, and special characters, which led to too many combinations having to be run through and the time it takes is too long to measure.

1. **Dictionary Attack**

The dictionary attack uses a predefined list of words, which each has a hash value[[1]](#footnote-1). If the hash of a password in the dictionary matches that of the yet-to-be-determined password, then the password is successfully identified by the attacker.

1. **Rainbow Table Attack**

A Rainbow Table attack is a password cracking method that uses a special table (a “rainbow table”) , which stores all the possible hashes of passwords of fixed length and components, to crack the passwords. However, there are two reasons that we gave up this method.

First, it is too strong that it cannot distinguish the cracking time between original and new passwords. Second, because our passwords consist of upper and lowercase letters, Arabic numerals, and special characters, it takes memory up to TBs to store the complete rainbow table.

Figure 11. shows we were trying to download a smaller rainbow table to use.

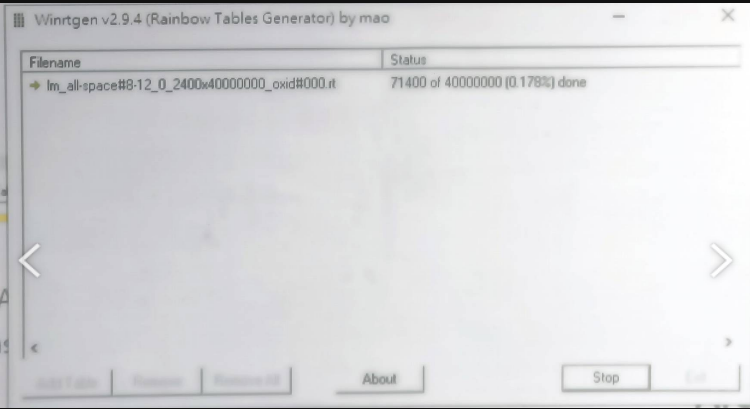
****

Figure 11.

### Source Code

(For an easier experience viewing the code, please visit [this website](https://replit.com/@annguo1106/password#main.c))

|  |
| --- |
| #include <stdio.h> #include <stdlib.h> #include <string.h>  // find seed for random int find\_seed(int len, int seed, char \*account, char \*key) {  for (int i = 0; i < len; i++)  for (int j = 0; j < 63; j++)  if (account[i] == key[j])  seed += j;  return seed; }  int main(void) {  char web\_key[] = "PwFeSiV2cCh7Ju8AnEsDaXd0HoTm9LrWtBb3Q5fUgRyKvZkI1xYl4MzNjOqG6p ";  // 63 -> this is provided from a special string   char account[100000]; // the web name you want to submit  char general\_password[100000]; // the password which you usually used   // scanf your web name and password  printf("Please enter your web's name : ");  fgets(account, sizeof(account), stdin);  printf("Please enter your general password : ");  fgets(general\_password, sizeof(general\_password), stdin);   int acc\_len = strlen(account), password\_len = strlen(general\_password);   // generate random number w -> used for deciding row numbers for matrix A and  // generating matrix A  int seed = 0;  seed = find\_seed(acc\_len, seed, account, web\_key);  srand(seed);  int w = rand() % 999 + 2;   int A\_row = (w % password\_len) + password\_len; // row numbers for matrix A   // allocate memory for matrix A -> is used for encode our passwords  int \*\*matrix = (int \*\*)calloc(A\_row, sizeof(int \*));  for (int i = 0; i < A\_row; i++)  matrix[i] = (int \*)calloc(password\_len, sizeof(int));   // generate matrix A using web\_key  int tmp = 0, tempcount = 0;  for (int i = 0; i < A\_row; i++) {  for (int j = 0; j < password\_len; j++) {  matrix[i][j] = ((int)web\_key[tmp]) + (2 \* w + 5) + (tempcount / 63);  tmp++;  tempcount++;  tmp %= 63;  }  }   //  for (int i = 0; i < A\_row; i++) {  for (int j = 0; j < password\_len; j++) {  printf("%d ", matrix[i][j]);  }  printf("\n");  }  // transpose general password  int number\_general\_password[password\_len][1];  for (int i = 0; i < password\_len; i++)  number\_general\_password[i][0] = (int)general\_password[i];   int new\_password[A\_row][1];   // encoding  for (int i = 0; i < A\_row; i++)  new\_password[i][0] = 0;  for (int i = 0; i < A\_row; i++) {  for (int j = 0; j < password\_len; j++)  new\_password[i][0] +=  (matrix[i][j] % 10000) \* (number\_general\_password[j][0] % 10000);  }   int number[10000];  int numberfornew[10000];   // combine every three numbers in a group and transport it to an ASCII code  int j = 0;  for (int i = 0; i < A\_row; i++) {  while (new\_password[i][0] > 0) {  number[j] = new\_password[i][0] % 10;  new\_password[i][0] /= 10;  j++;  }  }  if (j % 3 == 1) {  number[j] = 0;  j++;  } else if (j % 3 == 2) {  number[j] = 0;  number[j + 1] = 0;  j += 2;  }  int numcase = 0;  for (int i = 0; i < j; i += 3) {  numberfornew[numcase] =  number[i] \* 100 + number[i + 1] \* 10 + number[i + 2];  numberfornew[numcase] = (numberfornew[numcase] % 89) + 33;  numcase++;  }   // eliminate characters that can't be used in password  // characters can be used in password: 33 35~37 47~57(number) 64~90(capital)  // 97~122(lower case)  int index[] = {  33, 35, 36, 37,  64}; // ASCII code of special characters that can be used in password  for (int i = 0; i < j; i++) {  for (int k = 0; k < 5; k++) {  if (numberfornew[i] == 34)  numberfornew[i] = index[numberfornew[i] % 5];  else if (numberfornew[i] > 37 && numberfornew[i] < 47)  numberfornew[i] = index[numberfornew[i] % 5];  else if (numberfornew[i] > 57 && numberfornew[i] < 64)  numberfornew[i] = index[numberfornew[i] % 5];  else if (numberfornew[i] > 90 && numberfornew[i] < 97)  numberfornew[i] = index[numberfornew[i] % 5];  }  }   // transport ASCII code into corresponded character  char lastpassword[10000];  for (int i = 0; i < numcase; i++) {  lastpassword[i] = (char)numberfornew[i];  }  // printf encoded password  printf("New password : ");  if (strlen(lastpassword) < 10)  printf("%10s", lastpassword);  else  for (int i = 0; i < password\_len; i++)  printf("%c", lastpassword[i]);  printf("\n");  return 0; } |

|  |
| --- |
|  |

### Author Contributions

|  |  |  |
| --- | --- | --- |
| Name | Student ID | Contributions |
| 蔡承倢 | 111550119 | Code Implementation |
| 郭芷安 | 111550139 |
| 陳姸沂 | 111950031 |
| 顏名柔 | 111550075 | Evaluation of Program |
| 張芷瑜 | 111550131 |
| 李欣穎 | 111950020 |

1. a hash value refers to an encrypted output with a fixed length which is produced by an unique, mathematical function known as a hash function [↑](#footnote-ref-1)