Dynamic Salt Generation and Placement for Secure Password Storing

Sirapat Boonkrong and Chaowalit Somboonpattanakit

Abstract—**Cryptographic hash functions such as MD5 and SHA-1 are the most popular functions used for storing passwords. The main problem is that they were not designed to serve such purpose. Therefore, using them for storing passwords has generated a vulnerability. An attack using a rainbow table is possible. In order to counter this type of attack, a salt value has been introduced. However, attaching a salt value to a password is still found not to be enough. This research, therefore, proposes a method that helps generate and place a salt value into a password dynamically. After the implementation and mathematical analysis, the results show that if our method is applied, passwords will become more tolerant to the attack, which makes it more difficult to compromise.**

Index Terms—**authentication, dynamic, password, salt, secure.**

I. Introduction

AUTHENTICATION is considered a very important security mechanism for any IT systems. It is used to ensure that only authorised persons are allowed to enter and use the systems. It can also be used to prevent such attacks as information forgery [1] and phishing attack [2].

There are four common methods of authentication nowadays. They are "Something You Know", "Something You Have", "Something You Are" and "Something You Produce". Even though the "Something You Are" method such as biometric [3] is gaining popularity, this paper focuses on the "Something You Know" method of authentication, or commonly known as a *password*, and how to make it more secure.

Password is the most popular method of authentication today. It is the main method used for verifying the authorisation of a user before entering an IT system [4], [5]. Due to its importance, many have come up with ways to help make passwords more secure. For example, some organisations only allow the use of a password for a limited amount of time, while some organisations try to get their employees to generate passwords that are difficult to crack.

Unfortunately, many people still put convenience before security. In other words, many still use insecure passwords and many even use the same password to access all their accounts. It is, therefore, the job of the systems or service providers to find a way to secure those passwords. Without any secure mechanism for storing or keeping passwords, it is possible that an adversary can get hold of the passwords and use them in any wrongdoings.

One popular method that people have used to keep passwords more secure is cryptographic hash functions, such as MD5 [6] and SHA-1 [7]. That is, instead of storing password in the database in the plaintext format, their hash values are stored. However, the problem with these functions is that they were designed to be very fast to serve their main usage in message integrity checking. The fastness of MD5 and SHA-1 is considered the main nemesis of passwords, because it allows a rainbow table attack [8] and a dictionary attack [9] to be carried out. With today's technology, using a graphic processing unit or a GPU, more than one hundred million MD5 hashes can be computed per second [5]. This implies that a password that consists of one to six characters can easily be cracked, but longer passwords will take more time.

This is why a rainbow table has been created. A rainbow table, first pioneered by [8], is a large database containing pre-computed hash values of possible plaintext patterns [10]. This means that if an adversary gets hold of a password database that contains hash values of the passwords, the adversary will only need to compare those hash values with the ones in the rainbow table to know the corresponding plaintext. This implies that keeping passwords using just cryptographic hash functions is not enough. A well-published example is that of LinkedIn [11], when an attacker got hold of 6.5 million SHA-1 hash values of the passwords in the social network's database. The hash values were run through the rainbow table and many corresponding plaintext passwords were found. LinkedIn had to later come out and inform their users to change passwords, while they needed to find a better way to store them.

The chosen method was to introduce a random value known as a salt, to which a password would be appended before hashing. This is considered a more secure method. However, we will show in this research that this cannot withstand an attack, either. The main objective of this research is, therefore, to find an algorithm that helps instill a salt value into a password dynamically, rather than attaching them together. The way the salt value is used will be unique to each password. As a result, a more secure password storing technique can be achieved.

The rest of the paper is organised as follows. Section 2 contains background knowledge and existing methods for storing passwords and the analysis of each method. Section 3 explains our proposed algorithm for inserting a salt value into a password. Section 4 shows the speed, security and mathematical analyses of the proposed method. Section 5 concludes the paper.

A. Background Knowledge and Related Work

This section provides an overview of necessary knowledge on password storing methods as well as existing research on the subject of passwords. The section begins with a brief description of cryptographic hash functions, which

Manuscript received April 14, 2015; revised July 7, 2015; accepted September 2, 2015.

S. Boonkrong and C. Somboonpattanakit are with the Department of Data Communication and Networking, Faculty of Information Technology, King Mongkut's University of Technology North Bangkok, Thailand.
e-mail: sirapat.b@it.kmutnb.ac.th

are commonly used for storing passwords today. We then describe existing methods for storing passwords and give analyses on them. The existing research on passwords and topics related to passwords are also provided.

B. Cryptographic hash function

A cryptographic hash function is a mathematical function that is used to digest messages. That is, it takes any message as its input and outputs a value known as a hash value [12]. The size of the output or message digest depends on which algorithm is used. The common sizes include 128 bits, 160 bits and 256 bits.

One very important property of cryptographic hash function is the one-wayness property. That means it is easy to compute a result in one direction, but very difficult or impossible to compute a value in the reverse direction. In other words, given a value x and a function $f()$, it is easy to compute the value $f(x)$. However, knowing the value $f(x)$ and the function $f()$, it is very difficult to find the value x [13].

The second property of cryptographic hash function is the collision resistance property. A collision occurs when two different messages x and y are input to the same cryptographic hash function $h()$, and the exact same hash value is obtained as a result. Therefore, collision resistance means that it is very difficult for any cryptographic hash function to produce a collision.

Collisions are directly related to the security of hash functions and password storing. If it were easy to find two different messages that produced the same hash value, then an attacker would not need to know what the real password was. He or she only had to find another password that could produce the same hash value as the original password. The password would then be compromised.

The two most popular cryptographic hash functions that are applied to password storing are Message Digest 5 or MD5 [6] and Secure Hash Algorithm 1 or SHA-1 [7]. Both algorithms can take any message as an input and follow similar processes when producing a hash values. The main difference between the two is the size of the output. That is, MD5 produces a hash value that is 128 bits long whereas SHA-1 produces a hash value that is 160 bits long.

C. Methods for password storing

Before proposing an improved method for storing passwords, it is necessary to take a look at the existing methods [14], [15], [16], [10] for doing such thing first. Let us now go through and analyse each method in turn.

1) Plaintext passwords: The most simple way that a password can be stored in a database is in its plaintext form. This means that usernames and passwords are stored in the database in a human-readable format. For example, if a password is `password`, it will also be stored in the password database as `password`. When a user logs into the system, he or she enters his or her username and password, the system will then check them against the database to see whether or not they match. If so, the user will be allowed to enter the system.

This is considered the worst method to store a password, in security context. It is because if an adversary were able to access the password database, all users' passwords would immediately be seen by the attacker. Hence, all passwords would be compromised.

2) Encrypted passwords: One way to reduce the risk of passwords being exposed, encryption has been adopted as a possible solution. Encryption is a function that uses a secret key to transform plaintext to ciphertext. This means that even if an attacker were able to access the password database, he or she would not be able to see the passwords in their plaintext form. Only the ciphertext passwords would be seen.

This may appear secure at first glimpse. However, there is a problem with this method. The problem is that the secret key or the key that is used to encrypt and decrypt passwords is usually stored in the same database as the passwords. This means that if the password database were compromised, the attacker would also be able to get hold of the key. Hence, he or she could use it to decrypt all the ciphertext passwords to obtain the passwords in their plaintext format. Therefore, this method is also considered insecure.

3) Hashed passwords: Hashing a password is when a password is taken as an input into a cryptographic hash function such as MD5 or SHA-1. It is then processed and scrambled into an output that appears to be some random value. For example, using the MD5 algorithm to process that input password, `password`, we obtain its corresponding hash value as `5f4dcc3b5aa765d61d8327deb882cf99`. The SHA-1 hash value of the same password is `5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8`.

This method is often used in password systems because only the hash values of the passwords are stored in the database, rather than the passwords in their plaintext format. When a user logs into a system, the hash value of the entered password is computed and compared with the value stored in the database. If both values are exactly the same, the system assumes that the password is correct, and the user will be allowed to enter the system. If they do not match, the user will have to give another attempt to log in.

Using a hash function to store passwords appears to be a secure method for storing passwords. This is because even if an attacker has an access to the password database, he or she will not be able to see the password in the plaintext format. However, in recent years, there have been a number of high profile cases where large social networks had seen their users' passwords leaked to the public [11]. This is possible due to an attack or technology known as a rainbow table [8], [17] which is essentially a pre-computed hash values of possible plaintext. This means that if hash values of passwords are leaked, the attacker will only need to look up the rainbow table to find the corresponding plaintext passwords.

Having said that, not all possible passwords or plaintext can be found in the rainbow table yet. If hash values are not found in the rainbow table, it means that they are the hash values of long and complex passwords or plaintext that have not been pre-computed. However, the size of the rainbow table is growing all the time, and more passwords will eventually be found.

4) Multiple iterations hashed passwords: This method is similar to the password hashing method explained earlier. The difference is that rather than hashing a password once, the resultant hash value is then hashed again either by the same hash function or by a different hash function.

Even though this method makes it more difficult for the attacker to compromise the passwords, a database similar to the rainbow table can still be generated once the number of hash iterations are known.

5) Salted passwords: Due to the problem of the rainbow table, there needed to be a way to ease the concern. This was why a salt value has been introduced [14], [18]. A salt is a random string of letters or numbers that is added to the beginning or the end of a password, before hashing. In other words, instead of hashing just a password $h(password)$, we compute $h(salt||password)$. In this method, a different salt value is used for each password and is also stored in the same password database, in clear text. That means if the salt value and its position are known, it is still possible for an attacker to use the rainbow table to find the plaintext password.

This method has now been adopted by many Web sites and organisations in order to secure users' passwords. However, the placement of the salt value has been either at the beginning or the end of a password. We feel that the position of the salt value can affect the strength of password storing. Therefore, it is necessary to find a method to place the salt value in such a way that stored passwords become harder to crack and the efficiency is not noticeably reduced. This is the main objective of this research.

D. Related work

It should be re-emphasised here that the main problem studied in this paper is to find a way to securely store passwords. Apart from the basic knowledge stated in previous sections, we will take a look at what other researchers have done to improve the security of passwords and password storing.

First of all, it has to be stated that MD5 [6], invented in 1991, is a cryptographic hash function that has been used in many security applications. Of course, it has also been a popular function used to hash passwords before storing. This was the case until in 2005 when Wang and Hongbo [19] proposed a modular differential technique that could be applied to successfully find a collision in the MD5 algorithm. This has caused the research community to find an improvement in the existing hash function so that they can be used to make password storing more secure.

An example is the work of Chawdhury and Habib [20], who proposed that, on a networked system, the six reserved bits in the reserved field of the TCP header were to be used. They suggested that a random prime number was to be generated by a client machine and saved in the reserved field. The number would then be used as a part of the hashing process together with the password to obtain the resultant hashed password. The server would follow the same process to check whether or not the received password was correct. Thanawat and Boonkrong [21] also proposed a similar idea, but a random key was used instead of a random prime number. By carrying out these proposed methods, the papers claimed that it would make it harder for an adversary to attack using the rainbow table.

In 2012, Zheng and Jin [13] adjusted the way MD5 worked so that it would take longer to process. They argued that by increasing the processing time, the size of the rainbow table would grow less rapidly. Hence, hash values and their corresponding plaintext would be harder to find. One of the methods to achieve this was to place a random value called salt either in front of, at the back or in between a password before hashing it. The second way was to transform the plaintext password by using matrix and logical operations such as XOR before hashing it. The third method was to add some "information interference" to the plaintext password. For example, the username and the current time could be added to the password before inputting them into a hash function. It was claimed that any of the three proposed methods could make password storing more secure.

The work related to the above became an interest in the analysis of the placement of a salt value [22]. The analysis confirmed that using a salt value as a prefix or a suffix of a password increased the security of the password. However, with repeated experiments, an attacker could find a fixed point of the salt placement. Once the position of the salt was found, the security would be drastically reduced by, again, the use of the rainbow table.

It can be seen that rainbow table is the main source of attack. In 2013 [4], there was an extensive study on using rainbow table to compromise hashed passwords. For hashing with the MD5 algorithm, it was found that to construct a rainbow table for alpha-numeric passwords that were one to seven characters long, it would take approximately five days. Moreover, it only took around three and a half days to carry out cryptanalysis on all possible hashed passwords in the rainbow table.

Let us turn our attention to the work related to the strength of the actual password. Ma *et al.* [23] studied the relationship between the randomness and the quality of passwords. It was found that the randomness was not the factor affecting the strength or the quality of the passwords. This was because in order to evaluate the quality of passwords, it would be better to look at the actual time taken to crack them. Therefore, Ma *et al.* proposed a password complexity index that helped measure the password quality. As a result, they suggested that a strong password must contain at least eight characters, which consist of at least three special characters and numerical values should also be present. This idea will be applied in the design process of our proposed method for securely storing password later on.

It can be seen from our studies that there has been no work done on the quality of salt values and their positions within a password. Therefore, it is our intention to propose an algorithm to generate salt values and place them at appropriate positions so that the passwords become more tolerant to an attack via rainbow table.

II. Proposed Algorithm

This research investigates and designs a method for storing passwords securely, which consists of three factors. They are the quality of passwords, the selection of suitable salt values and a way to place the salt values into passwords. We discuss each factor in turn.

A. Adjusting the quality of passwords

Consider a scenario where a user registers his or her password, it is important to inspect the quality of the entered password before actually storing it. We suggest that this is done in order to ensure that it is not too easy to be compromised.

Ma *et al.* [23] came up with a password quality index, which requires that a strong password should contain at least eight characters, three of which should be special characters. There should be some numbers in the password, too. The criteria can be summarised in Table I. This is what we will use in the next step.

TABLE I
CRITERIA FOR STRONG PASSWORD

Criterion	**Property**
Number of Special Characters	At least 3 characters
Number of Numerical Values	At least 1 number
Number of Letters	At least 1 letter
Length of Password	At least 8 characters

The method for adjusting the quality of the password is practically our proposed salt generation process. That is, for each password being inspected, a different salt value will be generated. Which salt value is generated depends of the quality and components of each password. For example, if the original password lacks special characters, our algorithm will randomly generate and add special characters. If the password lacks numbers, our method will randomly generate and add numbers. Also the proposed method will ensure that the size of the password is at least of the required size.

After the inspection of the starting password, the existing number of especial characters, numeric values and letters will be known. A set of values will then be chosen from Table II in such a way that when combining it with the existing password, it will satisfy the criteria specified in Table I.

B. Finding suitable salt sizes

As mentioned, the work of Ma *et al.* found that the quality of a password could be measured by the time taken to crack it. Furthermore, they concluded that a harder to crack password should consist of at least eight characters that contained three or more special characters. A better password should contain numbers as well. These suggestions and combinations of characters will be used in our design experiments. The characters that will be used are shown in Table II.

TABLE II
PASSWORD CHARACTER SETS

Type of Characters	**Characters**
Special Character	!@#$%*∧&()-_+=˜ ‘’[]{}|\:;”<>,.?/
Number	0123456789
Alphabet	abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWXYZ

The next step is to generate a rainbow table using an Intel Core i7 CPU with the speed of 2.20 GHz. This is done so that the length of the passwords that will be tolerant to the rainbow table attack can be determined. The software that was used to obtain the rainbow table was *winrtgen* version 2.9.3. It was also used to calculate the time taken to complete the rainbow table with different password lengths. Moreover, the time taken to generate a rainbow table of the same password sizes using a graphical processing unit (GPU), extracted from [24], is compared with that of the CPU. Table III shows the time taken to generate a rainbow table using the stated CPU and the GPU.

TABLE III
TIME TAKEN TO GENERATE A RAINBOW TABLE

Password Length (Letters)	**CPU Computation Time**	**GPU Computation Time**
1 - 6	38.240 Days	47 Seconds
1 - 7	11.478 Years	1.14 Hours
1 - 8	1241.76 Years	465 Days
1 - 9	134069 Years	108 Years
1 - 10	$3.108830907 * 10^{33}$ Years	$2.527479675 * 10^{29}$ Years

The next step was to consider the time taken to compute hash values of inputs of different sizes. The MD5 algorithm was used in our experiment with the inputs taken from [25], which is the Web site containing ten thousand of the most common passwords. Each password was hashed one thousand times. The average time taken to complete one hash was calculated. The results are shown in Table IV.

TABLE IV
TIME TAKEN TO HASH DIFFERENT SIZES OF INPUTS

Input Size (Bits)	**No. of Printable Characters**	**Computation Time (ms)**
0	0	0.002707
64	8	0.002934
80	10	0.002945
128	16	0.002959
256	32	0.002982
512	64	0.003102
1024	128	0.003279

The result in Table IV can be depicted in the graph shown in Figure 1.

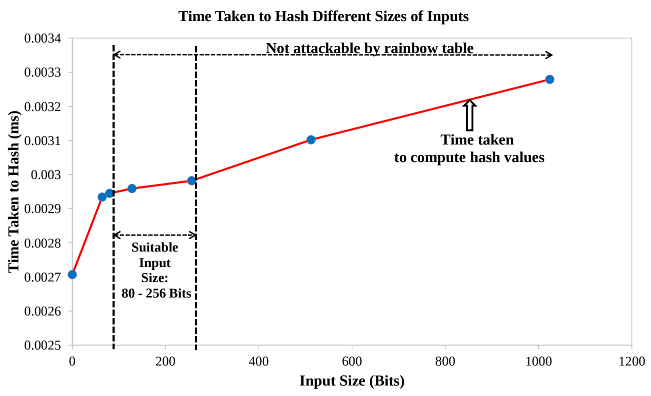

Fig. 1. Time Taken to Hash Different Sizes of Inputs

The analysis of the results in Figure 1, Table IV, together with the Table III, provides us with an idea of suitable salt or input sizes. In other words, it can be seen in Table III that if the input is nine characters long or larger, then it is very difficult to generate a complete rainbow table or the table with all possible plaintext and hash value pairs. Moreover, the result in Table IV shows that the time taken to hash any inputs that are ten to thirty-two characters long took approximately the same amount of time. Anything longer that thirty-two characters would take longer. Therefore, we claim here that the suitable sizes of a salt value is between 80 and 256 bits or 10 and 32 characters.

Now that the suitable sizes of a salt are known, a salt value can be generated for each password. All we have to do now is to find a way to integrate the salt value with the original password in such a way that the resultant password will satisfy the requirements in Table I.

C. Finding a pattern to place salt

This is the step where a pattern for placing the chosen salt value is generated. Bear in mind that the pattern is dynamic since it depends on the starting password and the chosen salt value in the previous steps. That means a different password and different salt value will have a different placement pattern. Our aim is the find a pattern so that when placing the salt into the password, the password will become stronger and harder to attack. Here, we propose an algorithm to find such placement pattern as follows.

First of all, the starting password is used as an input to a hash function to obtain its hash value. For this particular paper, MD5 is used. Secondly, the starting password and its hash value in converted into binary. In order to make it easier to understand, let us give a simple example.

Suppose our starting password is `password`. The password is input into MD5 to obtain the hash value as `0xD41D8CD9`. The password and the hash value are converted into binary as follows:

TABLE V
PASSWORD AND ITS HASH VALUE

Password	01110000 01100001 01110011 01110011 01110111 01101111 01110010 01100100
Hash Value	01100100 00110100 00110001 01100100 00111000 01100011 01100100 00111001

The third step is to XOR the binary values of the starting password and its own hash value. Here, we propose that only the least significant bit (rightmost bit) of each byte is used in the XOR operation.

Continue with our example, the binary values in Table V are XORed with one another. As stated, only the least significant bit of each resulting byte will be used as the placement pattern. Using the example above, the value `01011001` is obtained. This is our salt placement pattern. The next question is how this placement pattern is going to be used.

D. Salt placement rules

In order to answer the question set in the previous section, we need to come up with a number of rules so that the most suitable rule can be selected. The rules for applying the salt placement pattern are as follows.

Rule 1: If the bit value of the pattern is 0, no salt is to be placed into the password at that position. If the bit value of the pattern is 1, one character of salt is placed into the password at that position. If we run out of bit values of the pattern and there are still unused salt characters, append the rest of the salt characters to the end of the password.

Rule 2: If the bit value of the pattern is 0, no salt is to be placed into the password at that position. If there are two consecutive 0 bits in the placement pattern, two salt characters are to be placed into the password at the position. If the bit value of the pattern is 1, one character of salt is placed into the password at that position. If we run out of bit values of the pattern and there are still unused salt characters, append the rest of the salt characters to the end of the password.

Rule 3: If the bit value of the pattern is 0, no salt is to be placed into the password at that position. If there are two consecutive 0 bits in the placement pattern, one salt character is added to the front of the password. If the bit value of the pattern is 1, one character of salt is placed into the password at that position. If there are two consecutive 1 bits in the placement pattern, one salt character is added to the end of the password. If we run out of bit values of the pattern and there are still unused salt characters, append the rest of the salt characters to the end of the password.

Rule 4: If the bit value of the pattern is 0, one salt character is added to the front of the password. If the bit value of the pattern is 1, one salt character is added to the end of the password. If we run out of bit values of the pattern and there are still unused salt characters, append the rest of the salt characters to the end of the password.

Rule 5: If the bit value of the pattern is 0, the final or rightmost salt character is chosen to be placed into the password. If the bit value of the pattern is 1, the first or leftmost salt character is chosen to be placed into the password. If we run out of bit values of the pattern and there are still unused salt characters, append the rest of the salt characters to the end of the password.

It can be seen that we have avoided putting the salt in front of, in between and in the back of the password altogether. This is because having it in those position is considered in secure [22].

The next step is to find the best rule in terms of speed. The speed was measured by taking each of the ten thousand passwords from [25], and put it into a hash function, MD5 in this case, one hundred times. The average time was then calculated. The results from the experiment are shown in Table VI.

TABLE VI
SPEED OF SALT PLACEMENT RULES

Rule	Time Taken (ms)
1	2.49
2	2.48
3	2.50
4	2.50
5	2.50

Table VI shows that Rule 2 is the fastest salt placement rule as it only took 2.48 ms to complete the salt placement process. This is followed by Rule 1, which took 2.49 ms to complete. Rules 3, 4 and 5 took the same amount of time at 2.50 ms. Therefore, out of all the five rules, Rule 2 was selected as the salt placement pattern, because it provided the highest speed. The security of this rule will be analysed later on in the paper.

E. Resultant Algorithm

We will now summarise the algorithm for dynamic salt generation and placement that we believe to provide a solution towards a more secure password storing method. The proposed algorithm can be described as follows.

A user enters a password to be registered on a system in plaintext format. The quality of the entered password is evaluated against the criteria suggested by Ma *et al.* [23] A salt value is then chosen for the password in such a way that its size is appropriate for that particular password. Note that each password will be provided with a different salt value. The objective of this step is to ensure that when combining the password with the chosen salt value, we will end up with a stronger password before hashing or storing it.

Once an appropriate salt value is obtained, a salt placement pattern will be computed. This is done by XORing the original password with its hash value. The placement pattern will be the least significant bit of each resulting byte.

The next step is to insert the salt value into the password. This is done in accordance to the salt placement pattern and our chosen rule. The rule used in our proposed algorithm states that: If the bit value of the pattern is 0, no salt is to be placed into the password at that position. If there are two consecutive 0 bits in the placement pattern, two salt characters are to be placed into the password at the position. If the bit value of the pattern is 1, one character of salt is placed into the password at that position. If we run out of bit values of the pattern and there are still unused salt characters, append the rest of the salt characters to the end of the password.

What will be achieved after this stage is what we believe to be a stronger password, which will then be input to a one-way hash function. The resultant hash value is the value stored in the system's password database.

The proposed algorithm and an example can be summarised in Figure 2.

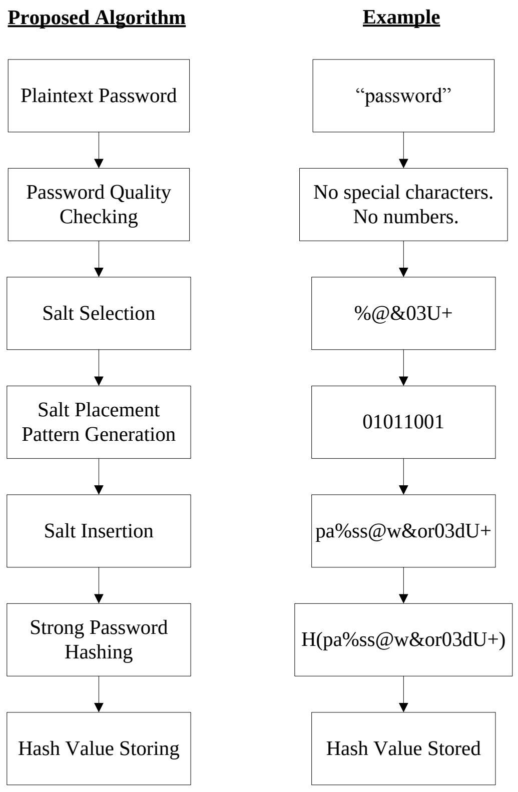

Fig. 2. Proposed Algorithm and an Example

It can be seen in Figure 2 that the starting password is `password`, whose quality is checked against the Ma *et al.*'s criteria [23]. It turns out that the original password does not contain any special character or number, which means that it does not meet the strong password criteria. A salt value, `%@&03U+`, is then chosen in such a way that when combining with the password, a stronger password is achieved. The salt placement is calculated in the next step by XORing `password` with its hash value. The least significant bit of every byte from the result is picked out to obtain `01011001`, which is the salt placement pattern. Let us now explain with this example how to insert the salt into the password using the proposed method.

We begin by looking at the the first character of the password, `password`, and the first bit of the placement pattern, which is `0`. According to the salt placement rule, when the placement pattern is 0, no salt is to be added at this position, so we move to the second character of the password and the second bit of the placement pattern.

The second bit of the placement pattern is `1`, which, according to the rule, means a salt value will be added at this position. We, therefore, take the first character of the salt value (we use the first character since no salt character has been used before) and place it behind the second character of the password to obtain `pa%ssword`.

We now move to the third character of the original password and the third bit of the placement pattern. The placement pattern now has the value of `0`, which means there is no need to add a salt character at this position.

The next character and the next bit of the placement pattern are now considered. The placement pattern is now `1`, which means one character of the salt is to be placed at this position to obtain `pa%ss@word`.

The fifth character of the original password and the fifth bit of the placement pattern are now looked at. The placement pattern has the value of `1`. This means one character of the salt value will be placed behind the fifth character of the password, which now becomes `pa%ss@w&ord`.

The next placement pattern is `0`, so we move to the next password character and do not insert any salt value. However, the next bit of the placement pattern also has the value of `0`. This means that we are seeing two consecutive zero bits, which means we have to insert two salt characters here at this position. Now we have `pa%ss@w&or03d`.

We are now at the final character of the original password and the final bit of the placement pattern. The last placement pattern bit is `1`, which means inserting one character of the salt value at this position. `pa%ss@w&or03dU` is now obtained.

At this stage, we have run out of salt placement bits, but there is one salt character left. The rule states that what we have to do now is to append the remaining salt character to the end of the current password. We, therefore, have `pa%ss@w&or03dU+` as the result.

Using the proposed rule to insert the salt value into the password, we obtain `pa%ss@w&or03dU+`. This password now meets the strong password criteria. It can now be hashed and stored in the database.

For the proposed algorithm, even if a rainbow table is used in an attack, a hash value is visible and a salt value is known by an adversary, the way that the salt value is inserted into the password is still unknown. Moreover, each password will be associated with a different salt value. The way the salt value is inserted into each password is also different. This makes it more difficult to compromise the password via the use of the pre-computed rainbow table.

In addition, by the principle of Kerckhoffs [26], even if an adversary gets hold of the source code, there is no way that the salt placement pattern will be known. This is because the pattern is determined by the original plaintext password. Therefore, the only way to figure out the placement pattern is by knowing the original password, which is not stored anywhere and hence cannot be known by the attacker.

III. Analysis of the Algorithm

In this section, we explain the analyses of the proposed algorithm based on two folds. The first is the speed of the algorithm compared with that of other existing password storing methods. The second is the attack tolerance.

Before going into the detail of the analyses and results, it has to be stated that we developed our test application using PHP, because all basic one-way hash functions, MD5 and SHA-1, were available. Apache was used to run a service for a Web server to run the PHP script. The database used for storing hash values and salt values was MySQL, while PHPMyAdmin was used to manage it.

A. Speed Analysis

The experiment was done by using MD5 [6] and SHA-1 [7] with several password storing methods. They were static salt (salt value never changes for any passwords), dynamic salt with different salt sizes (salt value changes for every password but the position was fixed) and our own algorithm.

The passwords used in this experiment were taken from a collection of ten thousand passwords from [25]. Each password was used as an input into each password storing method and run one hundred times before the average speed was found for each method. The results are shown in Table VII.

TABLE VII
Time Taken to Run Each Method

Method	Salt Length (Bits)	Time Taken using MD5 (ms)	Time Taken using SHA-1 (ms)
Static Salt	256	2.04	2.39
Dynamic Salt	64	2.07	2.47
Dynamic Salt	256	2.09	2.49
Dynamic Salt	512	2.52	2.59
Dynamic Salt	1024	2.56	2.64
Dynamic Salt	80 - 256	2.43	2.53
Proposed Algorithm	80 - 256	2.48	2.56

It can be seen that it took 2.48 ms and 2.56 ms to complete our proposed algorithm when using with MD5 and SHA-1 respectively. This means that it is slower than the static salt method and the dynamic salt methods with 64-bit and 256-bit salt, but faster than the dynamic salt method with 512-bit and 1024-bit salt. The time taken to run our algorithm is also comparable to the dynamic salt method with variable salt sizes.

Even though our proposed algorithm is not the fastest, we believe that its speed still is acceptable. This is because the difference of 0.4 ms from the fastest method is not really noticeable to users as stated in [27] that 0.1 second is the limit for users before feeling any latency.

B. Security Analysis

The main aim of this research is to design and develop an algorithm for securely storing passwords by using our dynamic salt generation and placement method together with a one-way hash function, where the word 'dynamic' means that the salt value and its placement pattern will always be different for every password. Therefore, the security aspect of the proposed method will need to be tested.

For the attack tolerance experiment, it was assumed that an attacker was able to access the password database [4], [11], which contained usernames, salt values and hash values of passwords. The job of the attacker was to find the plaintext passwords from the available information.

In this research, we used an application called *hash-cat* [28], which is a popular application used for password recovery and compromising passwords. Furthermore, a list of passwords used in our analyses was divided into two categories, which were weak passwords and strong passwords. The list of weak passwords was compiled by [29], and was the list of the worst passwords in 2013. The list of stronger passwords was made in such a way that they contained special characters and/or numbers as suggested by [23]. Both lists of passwords are shown in Table VIII.

TABLE VIII
Weak and Strong Passwords

Weak Password	Strong Password
123456	charles&jun!0r@
password	p@$$w0rd1234
12345678	Qw3rty1234
qwerty	Willi@md@ll@s
abc123	Il0vey0u7777
123456789	p@ssw0rd1@dmin
111111	london@midnight%
1234567	J3llyFish
iloveyou	A11B1ack$!
adobe123	jA(kBauer
123123	wroaps9ds
admin	.Doct0rH0use.
1234567890	@damS@ndler
letmein	princess@diamond!
photoshop	ILov3MyPi@no
1234	Jul1eLovesK3v1n
monkey	I34tcarr0ts
shadow	cookie%peanut@
sunshine	m0nkEyil0vey0u
12345	sh@d0wp@$$word
password1	$un$hInE
princess	Pr1ncE$$
azerty	s0p0ov1ot0
trustno1	@zErty1234
000000	Tru$tn01

The following methods were used to compare and evaluate the security level of our proposed method. The first method was the no salt method, which computed a hash value directly from a plaintext password.

The second method was the multiple iterations method. In this method, a plaintext password was hashed more than once. That is, the output from the hash function would be hashed again multiple times. In this research, the passwords were hashed three times before the results were stored on the database.

The third method was called the fixed salt method. In this method, a salt value that was 256 bits long was added to each password before being hashed by a one-way hash function. The same salt value was used for all passwords. Hence, the name fixed salt. Note that the salt value was also stored in the same database as the hash values of the passwords.

The fourth method was the dynamic salt method. This method generated a new salt value for every password. The salt value would then be concatenated to the password before it was hashed by a one-way hash function. Note that the salt value for each password was also stored on the password database.

The fifth method was our proposed method. That is, a new salt value was generated for every password. The salt position in the password was also changed for every password as a result of the placement pattern computed from the original password. Note that the salt value for each password was also stored on the same database as the hash values of the passwords.

The next step was to simulate attack scenarios. It was assumed that an attacker was able to access the password database which stored usernames, salt values and hash values of the passwords. These made up all the information available to the attacker. The aim of the attack was to turn the hash values into their corresponding plaintext passwords.

In this paper the *hashcat* application [28] was used as an attack tool. The experiment was run by using hashcat to attack both weak passwords and strong passwords presented in Table VIII. We also divided our experiment into four main attack scenarios that can be described as follows.

1) Scenario 1: Scenario 1 was to attack weak passwords that were stored by applying the five methods. MD5 was used as the one-way hash function. Table IX shows the results of the first attack scenario.

TABLE IX
ATTACK RESULTS OF SCENARIO 1

Password Storing Method	Attack Success Rate
No Salt	92%
Multiple Iterations	92%
Fixed Salt	92%
Dynamic Salt	92%
Proposed Algorithm	0%

It can be seen that when using *hashcat* to attack 25 weak passwords that were stored using the no salt, multiple iterations, fixed salt and dynamic salt methods with MD5, 23 passwords or 92% of all the passwords were compromised. However, all the weak passwords that were stored by our proposed method were left uncompromised.

2) Scenario 2: Scenario 2 was to attack strong passwords that were stored by applying the five methods. MD5 was used as the one-way hash function. Table X shows the results of the second attack scenario.

Table X shows that even if strong passwords were chosen, 24% of all the passwords or 6 out of 25 strong passwords were still compromised when using the no salt, multiple iterations, fixed salt and dynamic salt methods with MD5. However, no passwords were compromised when stored by using our proposed algorithm.

TABLE X
ATTACK RESULTS OF SCENARIO 2

Password Storing Method	Attack Success Rate
No Salt	24%
Multiple Iterations	24%
Fixed Salt	24%
Dynamic Salt	24%
Proposed Algorithm	0%

3) Scenario 3: Scenario 3 was to attack weak passwords that were stored by applying the five methods. SHA-1 was used as the one-way hash function. The attack results are shown in Table XI.

TABLE XI
ATTACK RESULTS OF SCENARIO 3

Password Storing Method	Attack Success Rate
No Salt	92%
Multiple Iterations	92%
Fixed Salt	92%
Dynamic Salt	92%
Proposed Algorithm	0%

It can be seen that when using hashcat to attack 25 weak passwords that were stored using the no salt, multiple iterations, fixed salt and dynamic salt methods with SHA-1, 23 passwords or 92% of all the passwords were compromised. However, no plaintext passwords were found when they were stored by using our proposed method.

4) Scenario 4: Scenario 4 was to attack strong passwords that were stored by applying the five methods. SHA-1 was used as the one-way hash function. Table XII shows the attack results of this scenario.

TABLE XII
ATTACK RESULTS OF SCENARIO 4

Password Storing Method	Attack Success Rate
No Salt	24%
Multiple Iterations	24%
Fixed Salt	24%
Dynamic Salt	24%
Proposed Algorithm	0%

Table XII shows that *hashcat* was able to compromise 24% of the passwords or 6 out of 25 strong passwords when they were stored using the no salt, multiple iterations, fixed salt and dynamic salt methods with SHA-1. However, *hashcat* was not able to compromise any of the stored passwords when our proposed method was applied.

On the whole, it can be seen that weak passwords were easy to crack when using the existing password storing methods. With stronger passwords stored by existing techniques, they were not as easy to crack although 24% of the corresponding plaintext of the stored passwords were found. However, when our proposed password storing method was applied to both weak and strong passwords, the *hashcat* application was not able to compromise any of them. Thus, the successful attack rate on our algorithm was 0%. Therefore, we can claim, based on the experiment results, that the proposed method can reduce the risk of passwords being compromised. In other words, the method can provide a more secure function for storing passwords.

It has to be noted that we are aware that stronger cryptographic hash functions such as SHA-256 do exist. However, they were not considered in this paper. This was because our objective is that we would like to illustrate that even though a weaker hash function was used, our proposed salt placement method would still provide a strong password storing method. Hence, there would be no need for a more computationally expensive hash function. Having said that, any cryptographic hash function can be used in conjunction with our proposed salt generation and placement method.

C. Mathematical Analysis

This section turns an attention to the mathematical analysis of the proposed password storing method. We applied a mathematical analysis on passwords using the method provided by Fites and Kratz [30]. The authors proposed a method for analysing and calculating the probability of a password being cracked as follows.

L =	length of time a password is valid
G =	number of password guesses possible in one second
A =	number of possible characters in each password position
M =	password length
P =	password space, which can be calculated by $P = M^A$

For our proposed method, the following values will be used. First of all, the length of time a password is valid L is taken to be 30 days or 2,592,000 seconds [31]. Secondly, the number of password guesses possible in one second G is suggested to be 100,000,000 guesses [32], using an ordinary desktop computer. Thirdly, the number of possible characters in each position A is 93, which comes from the number of uppercase letters, lower case letters, numbers and special characters. Fourthly, the password length M is the average number of characters of weak passwords from Table VIII plus the smallest suitable number of salt characters computed earlier in Section II. Therefore, in this case M is $7+10 = 17$. This means that $P = 17^{93}$.

Fites and Kratz stated in their paper that the probability or likelihood N that a password can be cracked is

$$N = (L * G)/P \tag{1}$$

Therefore, the likelihood that a password will be cracked using our proposed mechanism is $N = (2,592,000 * 100,000,000) * 17^{93}$ which is approximately $9.591 * 10^{-101}$. The number implies that there is a very low chance that a password stored by using the proposed method will be cracked within the space of thirty days.

IV. Conclusion

Password is one of the most used methods of authentication. Although it is very convenient to carry out authentication using passwords, there is one thing that needs to be considered. That is, how to store them securely, which is the main focus of this paper.

We first studied and learned that existing methods of password storing were not as secure as they should be. The existing password storing methods studied included the no salt, multiple iterations, fixed salt and dynamic salt methods. All of them were vulnerable to an attack.

The main aim of this paper was, therefore, to find a more secure way to store passwords. Our proposed method begins by checking the quality of the original password before randomly generating a salt value suitable for the password. This is the first contribution of the paper. An experiment was run and we found that in order to withstand a rainbow table attack, the size of the salt value must be between 80 and 256 bits or 10 and 32 characters.

Once the appropriate salt value has been found, the salt placement pattern is computed. This is the second contribution of this paper. The placement pattern is the result of XORing the original password and its hash value. The salt value is then inserted into the password according to the placement pattern. Next, this combination of salt value and password is to be hashed by a one-way hash function. The resultant hash value is the value to be stored in the password database.

It can be seen that, by using our proposed technique, even if an adversary gets hold of the hash value and the salt, it is very difficult that the password will be compromised. This is because the way the salt is placed in the password will never be known unless the plaintext password is known.

The analyses of the proposed method was done in two folds. The first was the speed analysis. It was found that the time taken to complete the execution of our method was approximately 2.50 ms, which was slower than the fixed salt method and dynamic salt method with small salt. However, our method performed better than the dynamic salt method with large salt value.

The second analysis was the security analysis. This was done by using both weak and strong passwords together with one-way hash functions. They were then attacked by using the *hashcat* application. The results showed that *hashcat* was not able to crack any of the passwords stored by using our method at all. Finally, a mathematical analysis of the proposed method was carried out. The analysis showed that using the proposed method, the likelihood that a password would be compromised by either guessing or cracking was very small. Therefore, it can be claimed that our proposed algorithm can help store passwords more securely than any other existing methods.

Acknowledgement

A special thank will have to go to the Faculty of Information Technology, King Mongkut's University of Technology North Bangkok and Rajamangala University of Technology Phra Nakorn for providing necessary facility and equipment for this research.

References

[1] M. F. Hashmi, A. R. Hambarde, and A. G. Keskar, "Robust image authentication based on hmm and svm classifiers," *Engineering Letters*, vol. 22, no. 4, pp. 183–193, 2014.

[2] P. A. Wang, "Online phishing in the eyes of online shoppers," *IAENG International Journal of Computer Science*, vol. 38, no. 4, pp. 378–383, 2011.

[3] A. A. Fathima, S. Vasuhi, N. T. N. Babu, Vaidehi, and T. M. Treesa, "Fusion framework for multimodal biometric person authentication system," *IAENG International Journal of Computer Science*, vol. 41, no. 1, pp. 18–31, 2014.

[4] H. Kumar, S. Kumar, R. Joseph, D. Kumar, S. Singh, A. Kumar, and P. Kumar, "Rainbow table to crack password using md5 hashing algorithm," in *Proceedings of IEEE Conference on Information & Communication Technologies (ICT)*, Jeju Island, 2013, pp. 433–439.

[5] A. P. Ratna, P. D. Purnamasari, A. Shaugi, and A. Salman, “Analysis and comparison of md5 and sha-1 algorithm implementation in simple-o authentication based security system,” in *Proceedings of IEEE International Conference on QiR (Quality in Research)*, Yogyakarta, 2013, pp. 99–104.

[6] R. Rivest, “The md5 message-digest algorithm: Rfc1321,” Internet Engineering Task Force, United States, 1992.

[7] D. Eastlake, 3rd and P. Jones, “Us secure hash algorithm 1 (sha1): Rfc3174,” Internet Engineering Task Force, United States, 2001.

[8] P. Oechslin, “Making a faster cryptanalytic time-memory trade-off,” in *Proceedings of Advances in Cryptology - CRYPTO 2003, 23^{rd} Annual International Cryptology Conference*, Santa Barbara, California, USA, 2003, pp. 617–630.

[9] A. Narayanan and V. Shmatikov, “Fast dictionary attacks on passwords using time-space tradeoff,” in *Proceedings of the 12^{th} ACM Conference on Computer and Communications Security*, ser. CCS ’05, 2005, pp. 364–372.

[10] K. Theocharoulis, I. Papaefstathiou, and C. Manifavas, “Implementing rainbow tables in high-end fpgas for super-fast password cracking,” in *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, 2010, pp. 145–150.

[11] L. Whitney, “Millions of linkedin passwords reportedly leaked online,” $http\ :\ //news.cnet.com/8301-1009_3-57448079-83/millions-of-linkedin-passwords-reportedly-leaked-online/$, accessed in November 2013.

[12] P. Li, Y. Sui, and H. Yang, “The parallel computation in one-way hash function designing,” in *Proceedings of International Conference on Computer, Mechatronics, Control and Electronic Engineering (CMCE)*, vol. 1, 2010, pp. 189–192.

[13] X. Zheng and J. Jin, “Research for the application and safety of md5 algorithm in password authentication,” in *Proceedings of the 9^{th} International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2012, pp. 2216–2219.

[14] R. Morris and K. Thompson, “Password security: A case history,” *Commun. ACM*, vol. 22, no. 11, pp. 594–597, Nov. 1979.

[15] P. Gauravaram, “Security analysis of salt$||$password hashes,” in *Proceedings of International Conference on Advanced Computer Science Applications and Technologies (ACSAT)*, 2012, pp. 25–30.

[16] S. Boonkrong, “Security of passwords,” *Journal of Information Technology*, vol. 8, no. 2, pp. 112–117, July–December 2012.

[17] M. Jorgensen, “Distributed rainbow table project,” https://www.freerainbowtables.com/en/tables2/, accessed in December 2013.

[18] D. Klein, “Foiling the cracker: A survey of, and improvements to, password security,” in *Proceedings of the United Kingdom Unix User’s Group*, London, England, 1990.

[19] X. Wang and Y. Hongbo, “How to break md5 and other hash functions,” in *Advances in Cryptology - EUROCRYPT*, 2005.

[20] M. Chawdhury and A. Habib, “Security enhancement of md5 hashed passwords by using the unused bits of tcp header,” in *Proceedings of the 11^{th} International Conference on Computer and Information Technology*, 2008.

[21] T. Matuamphan and S. Boonkrong, “An authentication system using md5 with random key,” Tech. Rep., March 2011.

[22] S. K. Sood, A. K. Sarje, and K. Singh, “Cryptanalysis of password authentication schemes: Current status and key issues,” in *Proceedings of International Conference on Methods and Models in Computer Science (ICM2CS 2009)*, 2009, pp. 1–7.

[23] W. Ma, J. Campbell, D. Tran, and D. Kleeman, “Password entropy and password quality,” in *Proceedings of the 4^{th} International Conference on Network and System Security (NSS)*, 2010, pp. 583–587.

[24] J. Atwood, “Speed hashing,” http://blog.codinghorror.com/speed-hashing/, accessed in December 2013.

[25] B. Mark, “10,000 most common passwords list,” https://xato.net/passwords/more-top-worst-passwords/, accessed in January 2014.

[26] F. A. P. Petitcolas, “Kerckhoffs’ principle.” in *Encyclopedia of Cryptography and Security (2nd Ed.)*, H. C. A. van Tilborg and S. Jajodia, Eds. Springer, p. 675.

[27] R. Miller, “Response time in man-computer conversational transactions,” in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS ’68 (Fall, part I), 1968, pp. 267–277.

[28] Kali, “How to crack passwords using hashcat - the visual guide,” http://uwnthesis.wordpress.com/2013/08/07/kali-how-to-crack-passwords-using-hashcat/, accessed in January 2014.

[29] D. Kevin, “The 2013 list of worst passwords,” http://splashdata.com/press/worstpasswords2013.htm, accessed in January 2014.

[30] P. E. Fites and M. P. J. Kratz, *Information Systems Security: A Practitioner’s Reference*. New York, USA: Van Nostrand Reinhold Co., 1993.

[31] G. Spafford, “Security myths and passwords,” USA, Tech. Rep., 2006.

[32] Openwall Community Wiki, “John the ripper benchmarks,” http://openwall.info/wiki/john/benchmarks, accessed in July 2015.

Sirapat Boonkrong is an assistant professor and an associate dean of academic affairs and research at the Faculty of Information Technology, King Mongkut’s University of Technology North Bangkok (KMUTNB), Thailand. He received his B.Sc. and Ph.D. in Computer Science from the Department of Computer Science at the University of Bath, UK. His main area of research is information and network security. He is currently a full-time lecturer at the Faculty of Information Technology, KMUTNB and is also supervising several Ph.D. students all of whom are in the field of information and network security.

Chaowalit Somboonpattanakit received his B.Sc. in Information Systems from Rajamangala University of Technology Phra Nakorn, at which he is now a computer and network specialist. He is currently studying for his M.Sc. in Information Technology at the Faculty of Information Technology, King Mongkut’s University of Technology North Bangkok (KMUTNB), Thailand. Chaowalit has also got Cisco CCNA and several Microsoft certificates to his name.